

# Práctica 5



*DSIC*

DEPARTAMENT DE SISTEMES  
INFORMÀTICS I COMPUTACIÓ

**IPC - DSIC**

**Prácticas**



# Objetivos

En esta práctica, vamos a ver los siguientes puntos:

- ➔ **Formularios**
  - ◆ **TextField**
  - ◆ **Date & Time Picker**
  - ◆ **RadioButton**
- ➔ **Diálogos modales**
- ➔ **CustomPainter**
- ➔ **Orientación y diseños responsive**
- ➔ **Ejercicio entregable**

# Formularios

---

# Formularios

Cuando un usuario ha de introducir información a través de una app/web, lo habitual es que rellene un formulario

- Introducir su nombre, edad, número de teléfono, correo electrónico, intereses...

Contact Form

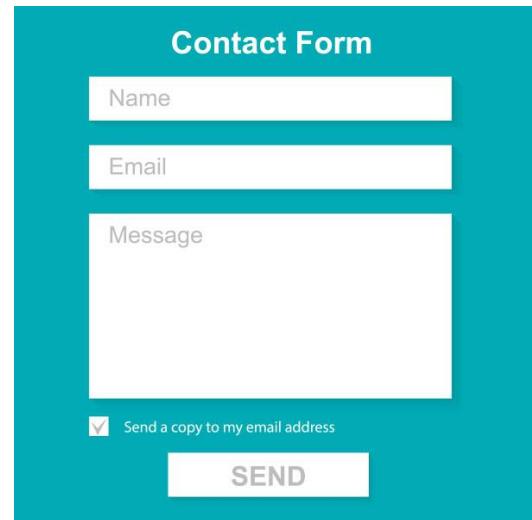
Name

Email

Message

Send a copy to my email address

SEND



# Formularios

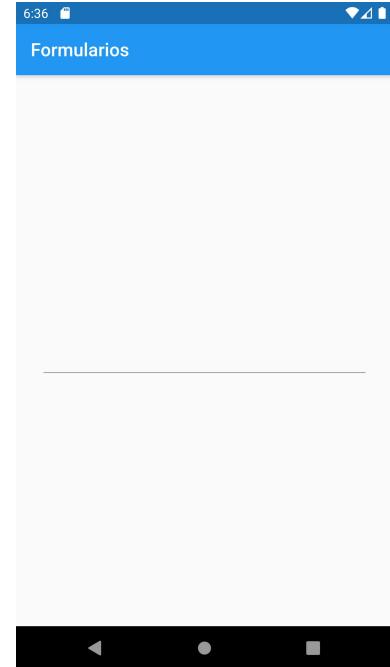
TextField



# Widget: TextField ([doc](#))

- Un **TextField** permite al usuario introducir información en forma de texto
  - ◆ Nombres, teléfonos, passwords, fechas, emails...

```
return Center(  
  child: Container(  
    margin: EdgeInsets.symmetric(horizontal: 30),  
    child: TextField(), // Container  
  ); // Center
```



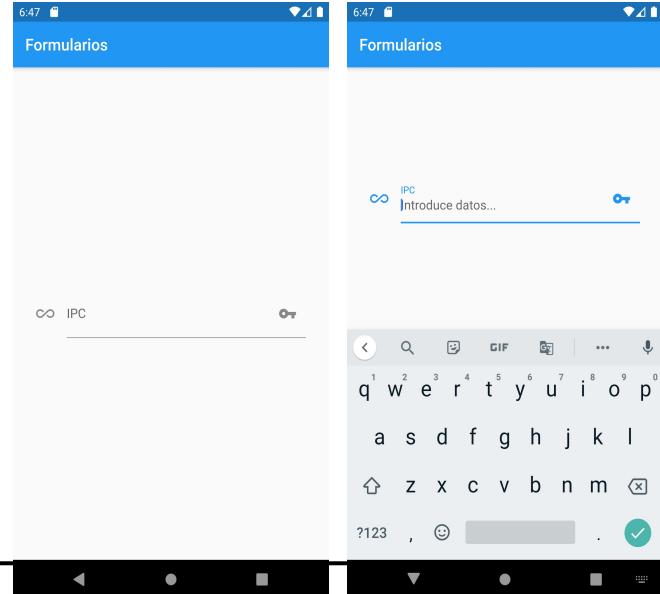
# Widget: TextField ([doc](#))

- En el momento en que se pulse en el formulario, el teclado software del dispositivo móvil aparecerá
- Algunos de sus campos más interesantes a la hora de llamar al constructor son:
  - ◆ **keyboardType** Alteran el teclado según el tipo de información a introducir (passwords, email, fechas...)
  - ◆ **decoration** Permite alterar el aspecto para mejorar el aspecto del campo de texto
  - ◆ **controller** Nos permite establecer un objeto para recuperar la información que haya introducido el usuario

# Widget: TextField ([doc](#))

→ Ejemplo con InputDecoration ([doc](#))

```
@override  
Widget build(BuildContext context) {  
    return Center(  
        child: Container(  
            margin: EdgeInsets.symmetric(horizontal: 30),  
            child: TextField(  
                keyboardType: TextInputType.name,  
                decoration: InputDecoration(  
                    icon: Icon(Icons.all_inclusive),  
                    labelText: "IPC",  
                    suffixIcon: Icon(Icons.vpn_key),  
                    hintText: "Introduce datos...",  
                ), // InputDecoration  
            ), // TextField, Container  
    ); // Center
```

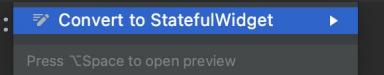


# Widget: TextField ([doc](#))

- Si queremos recuperar la información que ha introducido el usuario en el campo de texto, necesitamos definir un controller (de tipo TextEditingController ([doc](#)))
- Algunos usos de este objeto son:
  - ◆ Establecer un valor inicial en el TextField
  - ◆ Recuperar información almacenada
  - ◆ Estar a la escucha (*listener*) de cambios en el campo de texto

# Widget: TextField ([doc](#))

- 1º Creamos nuestro StatelessWidget con el formulario, y lo convertimos a StatefulWidget

```
class MyTextFieldContainer extends StatelessWidget {  
  const MyTextFieldContainer({Key? key}) : super(key: key);  
  
  @override  
  Widget build(BuildContext context) {  
    return Padding(  
      padding: const EdgeInsets.symmetric(horizontal: 10),  
      child: Column(  
        mainAxisAlignment: MainAxisAlignment.center,  
        children: const [  
          TextField(  
            decoration:  
              InputDecoration(icon: Icon(Icons.arrow_forward_ios_outlined)),  
          ), // TextField  
          SizedBox(  
            height: 20,  
          ), // SizedBox  
          Text("Result")  
        ],  
      ),  
    );  
  }  
}  
  
class MyTextFieldContainer extends StatelessWidget {  
  const MyTextFieldContainer({Key? key}) : super(key: key);  
    
  
```

# Widget: TextField [\(doc\)](#)

- 2º Declaramos un `TextEditingController` (`late`) como miembro de la clase, lo creamos en el `initState()` y lo liberamos en el `dispose()`
- Debemos sobreescribir estos 2 métodos
  - ◆ El método `initState()` se invoca 1 vez, cuando se añade al widget a la pantalla
  - ◆ El método `dispose()` se invoca cuando se elimina el objeto del árbol de widgets

# Widget: TextField ([doc](#))

→ 2º Declaramos un TextEditingController...

```
class _MyTextFieldContainerState extends State<MyTextFieldContainer> {

    late TextEditingController _controller;

    @override
    void initState() {
        super.initState();
        _controller = TextEditingController();
    }

    @override
    void dispose() {
        _controller.dispose();
        super.dispose();
    }
}
```

# Widget: TextField ([doc](#))

- 3º Asignamos ese `TextEditingController` al campo `controller` cuando construimos `TextField`

```
  - TextField(  
    controller: _controller,  
    decoration:  
      const InputDecoration(icon: Icon(Icons.arrow_forward_ios_outlined),  
    ), // TextField
```

# Widget: TextField ([doc](#))

- 4º Podemos escuchar los cambios que se producen en el **TextField** a través del controlador (será necesario un **hot restart** para que el método `initState()` se vuelva a ejecutar)

```
@override  
void initState() {  
    super.initState();  
    _controller = TextEditingController();  
    _controller.addListener(() {  
        debugPrint("Cambio: ${_controller.text}");  
    });  
}
```

# Widget: TextField ([doc](#))

- 5º Con el atributo **text del controller**, podemos modificar el texto que aparece por pantalla cada vez que el usuario introduce algo, sin embargo, no funcionará

```
children: [
  TextField(
    controller: _controller,
    decoration: const InputDecoration(icon: Icon(Icons.arrow_forward_ios)),
    // TextField
    const SizedBox(...), // SizedBox
    Text("Result: ${_controller.text}")
  ],
]
```

# Widget: TextField ([doc](#))

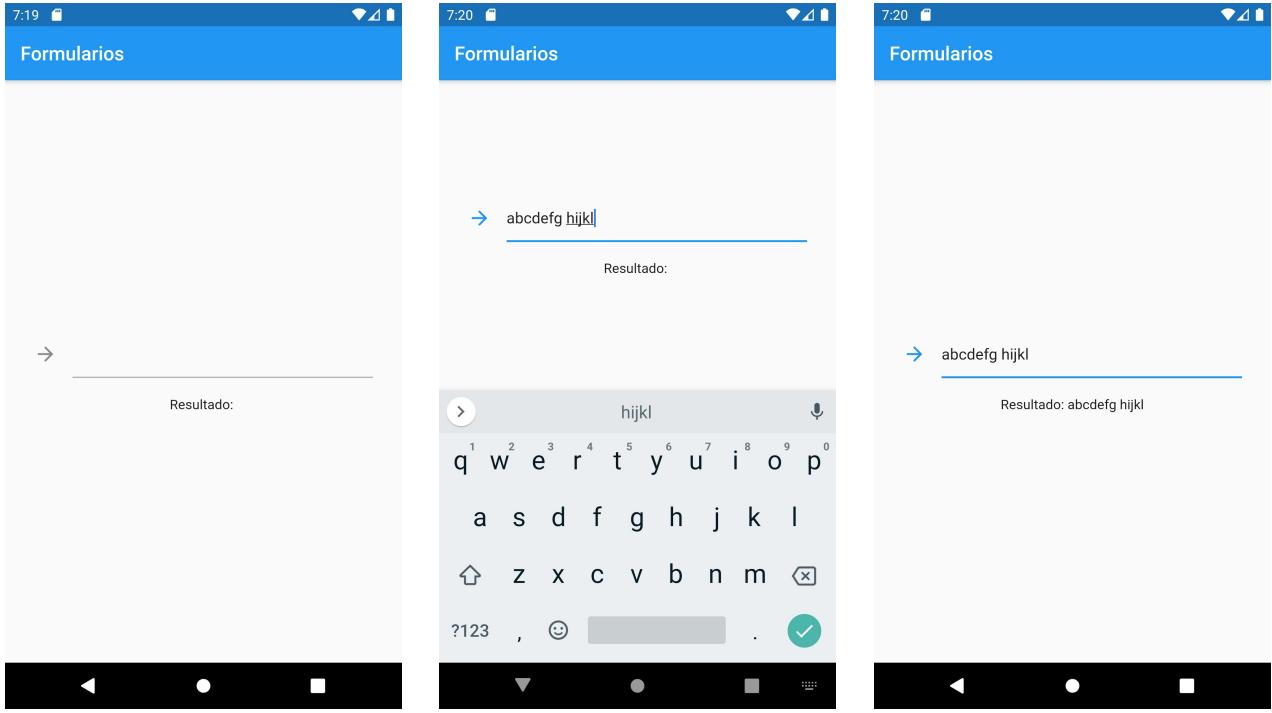
- 6º Para que funcione como esperamos, necesitamos que nuestro widget sea Stateful (ya lo hemos hecho!), ya que esto nos permite redibujarlo cuando detectemos que haya que hacerlo (e.g. cuando el usuario ha introducido un texto nuevo y queremos reflejar ese cambio en pantalla)
  - ◆ Para decirle a Flutter que ha de redibujar los cambios, hemos de llamar al método `setState()` ([doc](#))
  - ◆ Nosotros cambiaremos el texto inferior cuando el usuario acabe de introducir la información en el `TextField`

# Widget: TextField ([doc](#))

- 7º Llamamos al `setState()` => Este método acepta como parámetro una función que podemos dejar vacía
  - ◆ Con esto, el widget se redibujará para actualizar su visualización según el ESTADO

```
TextField(  
    controller: _controller,  
    onEditingComplete: () {  
        setState(() {});  
    },  
    decoration: const InputDecoration()  
, // TextField
```

# Widget: TextField ([doc](#))



# ¿Stateless o Stateful?

- Un widget Stateless es inmutable
  - ◆ Su método `build()` se llama 1 vez
- Un widget Stateful es mutable
  - ◆ Su método `build()` puede llamarse varias veces
  - ◆ Esto redibujará el widget
  - ◆ Para redibujarlo, utilizamos el método `setState()`
  - ◆ Lo usamos cuando hay estado que queramos reflejar en la interfaz
    - e.g. alguna variable que, cuando se vea modificada, el widget debe repintarse

# Formularios

Date & Time picker

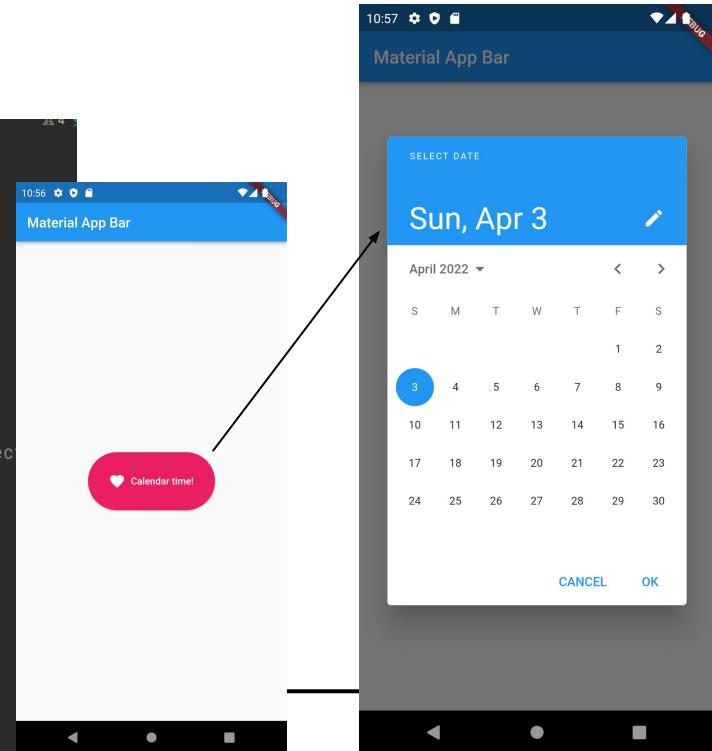
# showDatePicker() ([doc](#))

- A la hora de insertar fechas en cualquier aplicación, lo más sencillo es mostrar un calendario para que el usuario pueda seleccionar una fecha de forma intuitiva
  - ◆ Flutter solventa este problema mostrando un diálogo con el calendario, al cual debemos especificarle:
    - `context`
    - `initialDate` Fecha en la cual se abre el calendario
    - `firstDate` Límite inferior de fechas
    - `lastDate` Límite superior de fechas

# showDatePicker() ([doc](#))

## → Ejemplo

```
@override  
Widget build(BuildContext context) {  
  return Center(  
    child: ElevatedButton.icon(  
      icon: const Icon(Icons.favorite),  
      label: const Text("Calendar time!"),  
      style: ButtonStyle(  
        backgroundColor: MaterialStateProperty.all(Colors.pink),  
        padding: MaterialStateProperty.all(const EdgeInsets.all(30)),  
        shape: MaterialStateProperty.all(RoundedRectangleBorder(  
          borderRadius: BorderRadius.circular(50))), // RoundedRect  
        onPressed: () {  
          showDatePicker(  
            context: context,  
            initialDate: DateTime.now(),  
            firstDate: DateTime(1900),  
            lastDate: DateTime(2099));  
        },  
      ), // ElevatedButton.icon  
    ); // Center
```



# showDatePicker() ([doc](#))

- Como habrás observado, el diálogo dispone de 2 botones en la parte inferior, uno para aceptar (OK) la fecha seleccionada y otro para cancelar (CANCEL) la selección
- Debemos esperarnos (de manera asíncrona) a que el usuario seleccione una de las dos opciones para poder recuperar el valor seleccionado, para lo que disponemos de 2 opciones
  - a. Podemos utilizar `await` en un método asíncrono
  - b. Podemos encadenar `then` en la misma llamada a `showDatePicker()`

# showDatePicker() ([doc](#))

→ Con await

```
void selectDate(BuildContext context) async {
  final value = await showDatePicker(
    context: context,
    initialDate: DateTime.now(),
    firstDate: DateTime(1900),
    lastDate: DateTime(2099));

  if (value != null) {
    debugPrint("OK => ${value.weekday}");
  } else {
    debugPrint("Cancelled!");
  }
}

@Override
Widget build(BuildContext context) {
  return Center(
    child: ElevatedButton.icon(
      icon: const Icon(Icons.favorite),
      label: const Text("Calendar time!"),
      style: ButtonStyle(...), // RoundedRectangleBorder, ButtonStyle
      onPressed: () => selectDate(context),
    ), // ElevatedButton.icon
  ); // Center
}
```



# showDatePicker() ([doc](#))

→ Con then

```
@override
Widget build(BuildContext context) {
  return Center(
    child: ElevatedButton.icon(
      icon: const Icon(Icons.favorite),
      label: const Text("Calendar time!"),
      style: ButtonStyle(...), // RoundedRectangleBorder, ButtonStyle
      onPressed: () {
        showDatePicker(
          context: context,
          initialDate: DateTime.now(),
          firstDate: DateTime(1900),
          lastDate: DateTime(2099))
          .then((pickedValue) {
            if (pickedValue != null) {
              debugPrint("OK => ${pickedValue.weekday}");
            } else {
              debugPrint("Cancelled!");
            }
          });
      },
    ), // ElevatedButton.icon
  ); // Center
}
```

# showDatePicker( ) [\(doc\)](#)

- A la hora de recuperar los datos, podemos seguir la misma estrategia que con el StatefulWidget del TextField
  - ◆ Convertimos la clase de Stateless a Stateful
  - ◆ Creamos un miembro de tipo DateTime [\(doc\)](#) DateTime? \_birthday;
  - ◆ Cuando el usuario seleccione un valor en el calendario, modificamos ese miembro
  - ◆ Invocamos setState( ) para redibujar la interfaz y mostrar la fecha seleccionada
- Se deja al alumno que realice este ejercicio

# Operador ??

- Tras realizar el ejercicio anterior, una vez almacenamos la fecha seleccionada, queremos modificar la llamada a `showDatePicker()` para que la `initialDate` sea, o bien ahora (`now`) o bien la fecha seleccionada
- El operador binario `??` devuelve la expresión de la izquierda si no es `null`, pero si lo es, devuelve la de la derecha

```
var a = null;  
var b = 10;  
  
var c = a ?? b;
```

# Operador ??

## → Ejemplo

```
showDatePicker(  
    context: context,  
    initialDate: _birthday ?? DateTime.now(),  
    firstDate: DateTime(1900),  
    lastDate: DateTime(2099))  
    .then((pickedValue) {  
        if (pickedValue != null) {  
            debugPrint("OK => ${pickedValue.weekday}");  
            _birthday = pickedValue;  
        } else {  
            debugPrint("Cancelled!");  
        }  
    });
```



# Mostrar widgets de forma condicional

- En ocasiones, puede ser interesante mostrar una serie de Widgets u otros en base a una condición
- En este caso, podemos querer mostrar un ícono o un texto dependiendo de si el usuario ha seleccionado alguna fecha
  - ◆ Esto podemos realizarlo utilizando las sentencias condicionales `if-else` o el operador ternario de Dart

# Mostrar widgets de forma condicional

## → Ejemplo

```
return Container(
  width: double.infinity,
  child: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
      RaisedButton(
        color: Colors.lightGreen,
        child: Text("Calendar time!"),
        onPressed: () => selectDate(context),
      ), // RaisedButton
      SizedBox(
        height: 20,
      ), // SizedBox
      if (_birthday != null)
        Text("${_birthday.day} de ${_birthday.month}, ${_birthday.year}")
      else
        Icon(Icons.device_unknown)
    ],
  ), // Column
); // Container
```



# Mostrar widgets de forma condicional

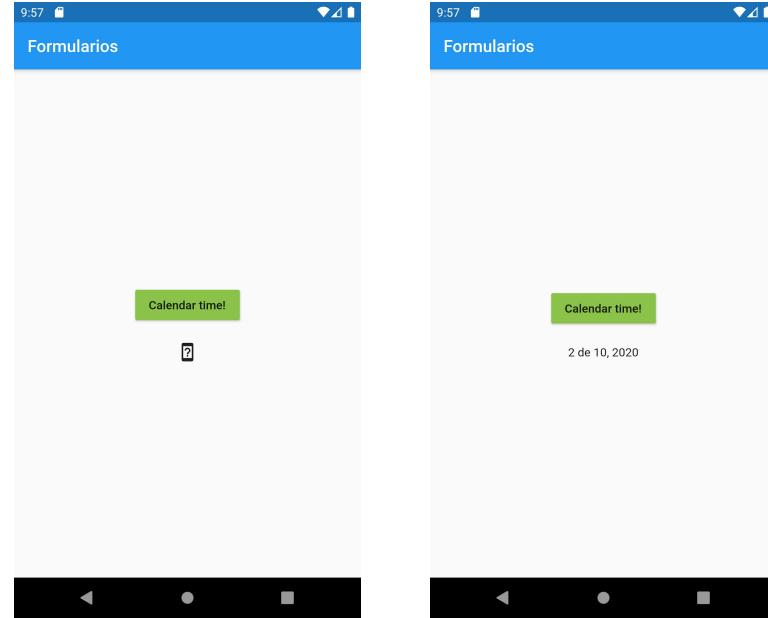
## → Ejemplo

```
return Container(
  width: double.infinity,
  child: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
      RaisedButton(
        color: Colors.lightGreen,
        child: Text("Calendar time!"),
        onPressed: () => selectDate(context),
      ), // RaisedButton
      SizedBox(
        height: 20,
      ), // SizedBox
      _birthday != null
        ? Text("${_birthday.day} de ${_birthday.month}, ${_birthday.year}")
        : Icon(Icons.device_unknown)
    ],
  ), // Column, Container
)
```



# Mostrar widgets de forma condicional

→ Ejemplo



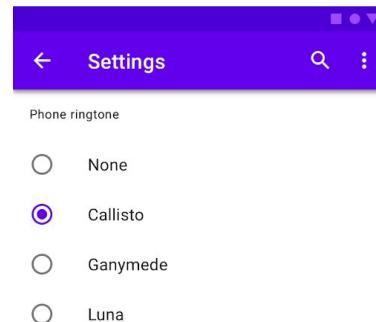
# Formularios

RadioButton



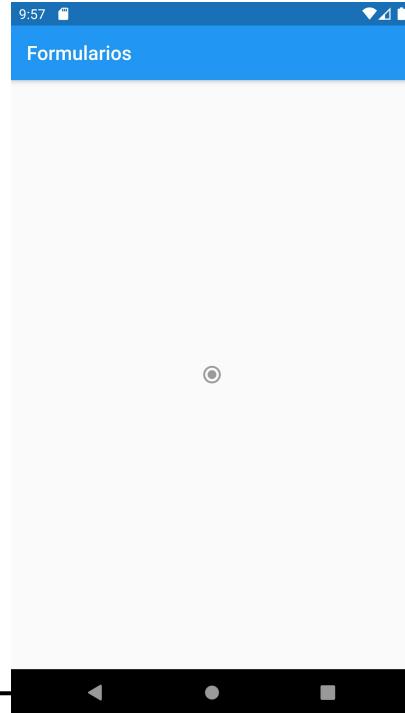
# Widget: Radio [\(doc\)](#)

- Cuando necesitamos que un usuario, en un formulario, introduzca un dato de forma excluyente/única, utilizamos un botón radio (Radio button)
  - ◆ Género (Masculino, Femenino...)
  - ◆ Rango de edad (Entre 0 y 10, entre 11 y 20...)
  - ◆ Casa (Gryffindor, Hufflepuff...)



# Widget: Radio [\(doc\)](#)

- En Flutter disponemos del widget Radio para conseguir este objetivo; sin embargo, por sí solo únicamente muestra un botón circular
- Tendremos que acompañarlo de otros widgets como textos, iconos...



# Widget: Radio ([doc](#))

- Cuando creamos un Radio, debemos establecer los siguientes campos
  - ◆ **groupValue** Objeto donde almacenaremos el valor seleccionado
  - ◆ **onChanged** Función que se ejecutará cuando se cambie el valor (e.g. para actualizar **groupValue**)
  - ◆ **value** Valor que tomará el **groupValue** cuando se pulse en este Radio

# Widget: Radio ([doc](#))

→ Ejemplo, en un StatefulWidget...

```
@override  
Widget build(BuildContext context) {  
  return Column(  
    mainAxisAlignment: MainAxisAlignment.center,  
    children: [  
      Row(  
        mainAxisAlignment: MainAxisAlignment.center,  
        children: const [  
          Expanded(child: Text("Griffindor")),  
          Flexible(  
            flex: 2,  
            child: Radio(  
              groupValue: null,  
              onChanged: null,  
              value: null,  
            ), // Radio  
            ), // Flexible  
          ],  
        ), // Row  
      Row(...), // Row  
      Row(...), // Row  
      Row(...), // Row  
    ],  
  ); // Column
```



# Widget: Radio ([doc](#))

- Vamos a crear un miembro de tipo **String** para almacenar el valor de cada Radio
  - ◆ Lo asignaremos al **groupValue**
  - ◆ En **value** pondremos el valor de cada 'caso' según el valor asociado a cada Radio
  - ◆ En **onChanged**, llamaremos al **setState()** para que se actualice la interfaz, actualizando también el miembro que acabamos de declarar

```
String? _house = "Hufflepuff"; Valor inicial
```

# Widget: Radio [\(doc\)](#)



## → Ejemplo

```
Widget build(BuildContext context) {  
  return Column(  
    mainAxisSize: MainAxisSize.center,  
    children: [  
      Text("You belong to ${_house}"), const SizedBox(height: 40),  
      Row(  
        mainAxisSize: MainAxisSize.center,  
        children: [  
          const Expanded(child: Text("Griffindor")),  
          Flexible(flex: 2,  
            child: Radio(  
              groupValue: _house,  
              onChanged: (String? newHouse) {  
                setState(() {  
                  _house = newHouse;  
                });  
              },  
              value: "Griffindor",  
            ), // Radio  
          ), // Flexible  
        ],  
      ), // Row  
      Row(...), // Row  
      Row(...), // Row
```

You belong to Ravenclaw

Griffindor

Slytherin

Ravenclaw

Hufflepuff

# Otros widgets similares

→ Existen otros widgets similares, como **RadioListTile** ([doc](#)) o **CheckboxListTile** ([doc](#))

Lafayette

Thomas Jefferson



Animate Slowly



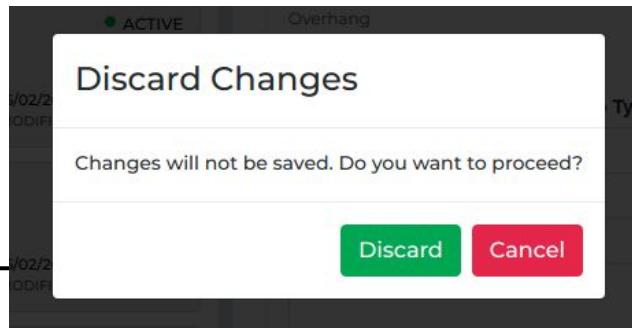
# Diálogos modales

AlertDialog



# Diálogos modales

- En ocasiones, es posible que necesitemos que el usuario realice alguna acción para poder avanzar o retroceder
- Un diálogo modal es una ventana que aparece sobre la interfaz/ventana principal, dejando que se vea lo que hay detrás, pero obligando a que el usuario preste atención a este diálogo
- Para continuar, el usuario deberá interactuar con el modal



# Widget: AlertDialog ([doc](#))

- Flutter nos ofrece la clase AlertDialog para estos casos
- Podemos alterar su contenido cuando llamamos a su constructor, por ejemplo:
  - ◆ title El título del diálogo
  - ◆ content El contenido del diálogo (información a mostrar)
  - ◆ actions El listado de acciones que podrá realizar el usuario para cerrar el diálogo
- Para mostrar un AlertDialog, hay que pasarlo como parámetro al llamar al método showDialog() ([doc](#))

# Widget: AlertDialog ([doc](#))

- En un stateful/less widget, creamos un método que devuelva un AlertDialog

```
AlertDialog _createAlertDialog(BuildContext context) => AlertDialog(  
  title: const Text("Mi modal"),  
  content: Column(  
    mainAxisSize: MainAxisSize.min,  
    children: [  
      const Text("Un texto"),  
      Row(  
        children: const [  
          Icon(Icons.accessible_sharp),  
          Icon(Icons.account_balance)  
        ],  
      ),  
    ],  
  ),  
  actions: [  
    TextButton(  
      onPressed: () => Navigator.of(context).pop(),  
      child: const Text("Cancelar"), // TextButton  
    ),  
    TextButton(  
      onPressed: () => Navigator.of(context).pop(),  
      child: const Text("Aceptar") // TextButton  
    ),  
  ],  
); // AlertDialog
```

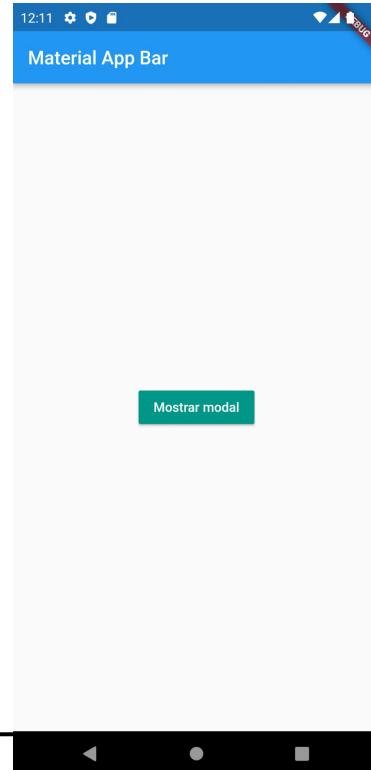
# Widget: AlertDialog ([doc](#))

- Llamamos a `showDialog()` para mostrar el `AlertDialog` que hemos definido

```
@override
Widget build(BuildContext context) {
  return Center(
    child: MaterialButton(
      color: Colors.teal,
      child: const Text(
        "Mostrar modal",
        style: TextStyle(color: Colors.white),
      ),
      onPressed: () => showDialog(
        context: context,
        builder: (context) => _createAlert(context)),
    );
}
```

# Widget: AlertDialog ([doc](#))

→ Ejemplo



# Custom Painter



# ¡Somos artistas!

- En ocasiones, puede que necesitemos dibujar un Widget que Flutter no nos ofrezca, o puede que el 'estilo' en que aparece no sea el que nosotros queramos
- Flutter, a través de la clase `CustomPainter`, pone a nuestra disposición un lienzo (canvas) para que dibujemos nuestro widget como queramos

# Clase: CustomPainter ([doc](#))

- El primer paso es crear una clase que herede de la clase abstracta CustomPainter
- Deberemos sobreescribir los siguientes métodos:
  - ◆ `void paint(canvas, size)`
    - En el definiremos los pinceles y los trazos a dibujar en nuestro canvas
  - ◆ `bool shouldRepaint(oldDelegate)`
    - Devolveremos true

# Clase: CustomPainter [\(doc\)](#)

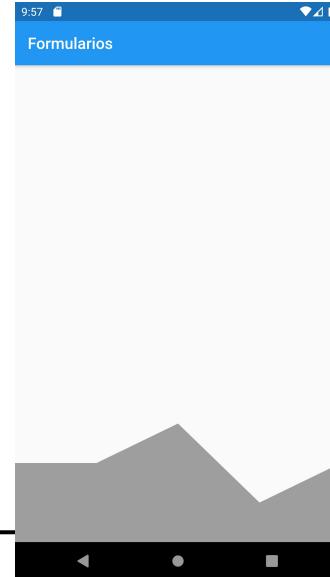
→ Ejemplo

```
class MyFooter extends CustomPainter {  
    @override  
    void paint(Canvas canvas, Size size) {  
        final paint = Paint();  
  
        paint  
            ..color = Colors.grey      Características  
            ..style = PaintingStyle.fill;  del pincel  
  
        final path = Path()  
            ..moveTo(0, size.height - 100)  Características  
            ..lineTo(0, size.height)       del trazo  
            ..lineTo(size.width, size.height)  
            ..lineTo(size.width, size.height - 100)  
            ..lineTo(size.width * 0.75, size.height - 50)  
            ..lineTo(size.width * 0.5, size.height - 150)  
            ..lineTo(size.width * 0.25, size.height - 100);  
  
        canvas.drawPath(path, paint);  
    }  
  
    @override  
    bool shouldRepaint(covariant CustomPainter oldDelegate) {  
        return true;  
    }  
}
```

# Clase: CustomPainter ([doc](#))

- Para pintar ese contenido, necesitamos utilizar el Widget [CustomPaint](#) ([doc](#)), que es capaz de renderizar clases que heredan de [CustomPainter](#)

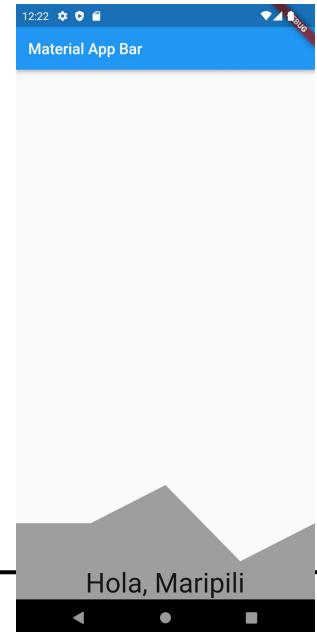
```
class MyFooterContainer extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return SizedBox.expand(  
      child: Container(  
        child: CustomPaint(  
          painter: MyFooter(),  
        ), // CustomPaint  
      ), // Container  
    ); // SizedBox.expand  
  }  
}
```



# Clase: CustomPainter [\(doc\)](#)

- Al crear un **CustomPaint**, podemos utilizar su propiedad **child** para añadir cualquier otro Widget

```
@override  
Widget build(BuildContext context) {  
  return SizedBox.expand(  
    child: CustomPaint(  
      painter: MyFooter(),  
      child: const Align(  
        alignment: Alignment.bottomCenter,  
        child: Text("Hola, Maripili", style: TextStyle(fontSize: 36),),  
      ), // Align  
    ), // CustomPaint  
  ); // SizedBox.expand  
}
```

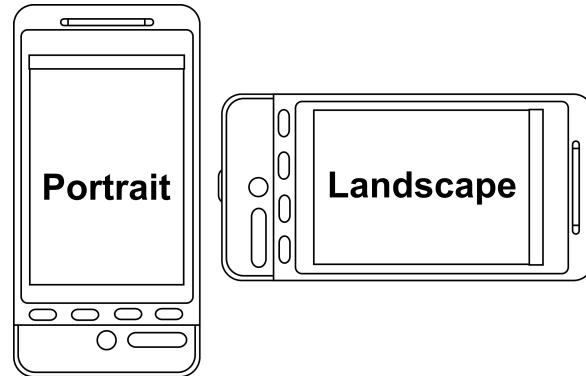


# Orientación y diseños responsive

Cómo trabajar con distintas orientaciones y tamaños de pantalla

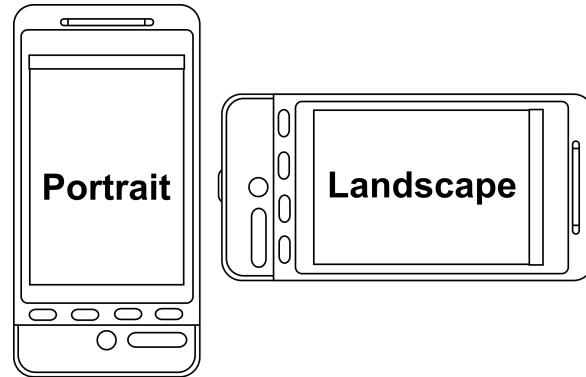
# Detección de la orientación

- Cuando trabajamos con dispositivos móviles, distinguimos entre 2 tipos de orientaciones
  - ◆ Landscape (Horizontal)
  - ◆ Portrait (Vertical)



# Detección de la orientación

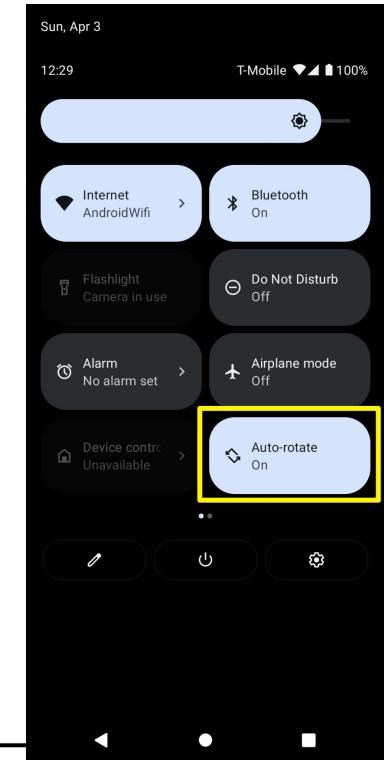
- En Flutter podemos envolver nuestros Widgets en un **OrientationBuilder** ([doc](#)), lo cual nos devolverá la orientación del dispositivo para así poder actuar en consecuencia



# Detección de la orientación

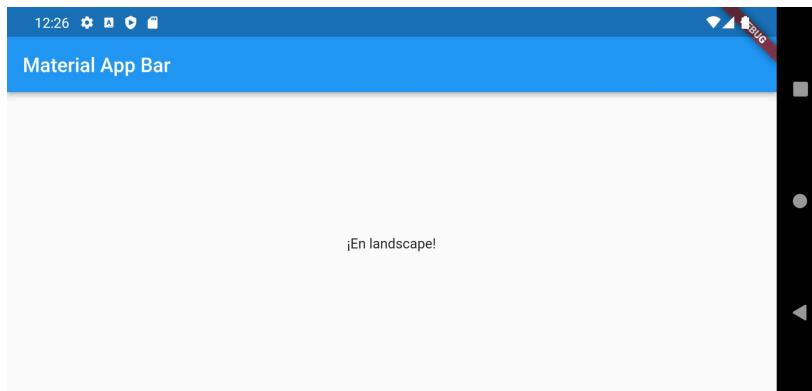
## → Ejemplo

```
@override  
Widget build(BuildContext context) {  
  return Center(  
    child: OrientationBuilder(builder: (context, orientation) {  
      if (orientation == Orientation.landscape) {  
        return const Text("¡En landscape!");  
      } else {  
        return const Text("¡En portrait!");  
      }  
    },), // OrientationBuilder  
); // Center  
}
```



# Detección de la orientación

→ Ejemplo



¡En landscape!



¡En portrait!

# Diseño responsive

- Sin embargo, cuando pensamos en otros dispositivos tales como un PC, la noción de orientación del dispositivo desaparece
- A la hora de desarrollar interfaces responsive para estos dispositivos, la estrategia más habitual de trabajar es comprobar el ancho de la pantalla y actuar en consecuencia
  - ◆ Un navegador web puede redimensionarse, por lo que deberíamos ofrecer diferentes interfaces según el ancho disponible
- Esta estrategia también es válida cuando trabajamos con móviles, dado que el ancho varía según la orientación del dispositivo (de hecho, una estrategia muy común es la de **Mobile-first**, en contraposición a **Desktop-first**)

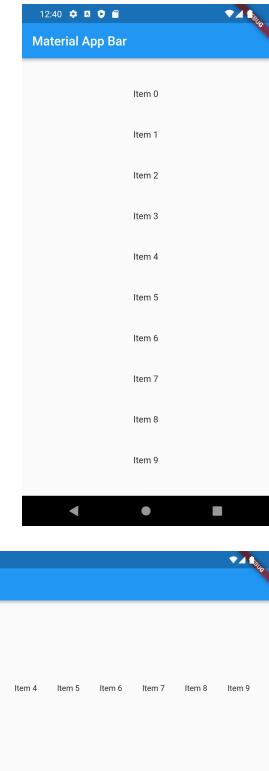
# Clase: MediaQuery ([doc](#))

- A través del objeto **MediaQuery**, podemos obtener información en forma de un objeto tipo **MediaQueryData** ([doc](#)), que nos proporciona información sobre :
  - ◆ Tamaño de la pantalla
  - ◆ Brillo
  - ◆ Densidad de píxeles
  - ◆ Accesibilidad habilitada (VoiceOver, etc)
  - ◆ ...

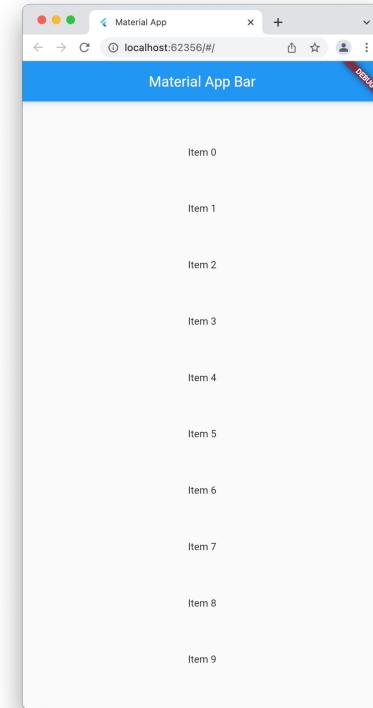
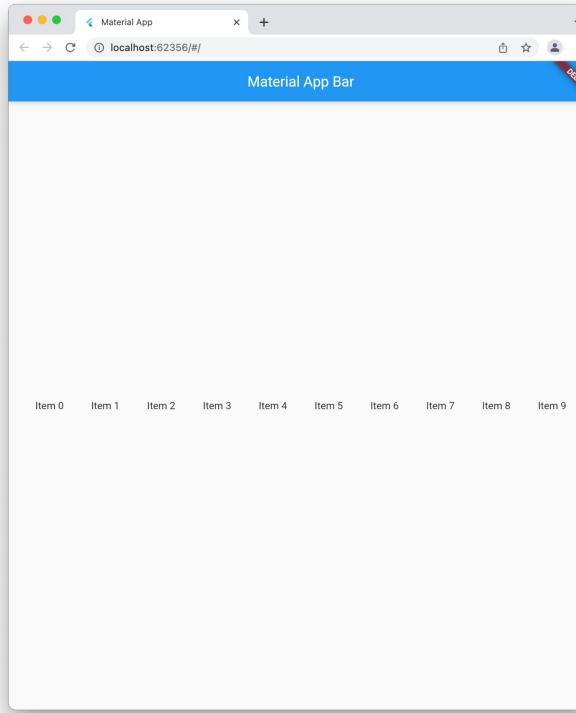
# Detección de la orientación

→ Ejemplo, definiendo un punto de control

```
@override  
Widget build(BuildContext context) {  
  final size = MediaQuery.of(context).size;  
  var items = List.generate(10, (index) => Text("Item $index"));  
  
  Widget distribution;  
  
  if (size.width > 500) {  
    distribution = Row(  
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
      children: items,  
    );  
  } else {  
    distribution = Column(  
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
      children: items,  
    );  
  }  
  
  return SizedBox.expand(child: distribution);  
}
```



# Detección de la orientación



# Ejercicio entregable



# Ejercicio entregable P5

El entregable de esta práctica consiste en desarrollar una aplicación formada por 2 pantallas:

- Una pantalla con un formulario, donde el usuario podrá introducir sus datos personales (formulario) y validarlos (diálogo modal)
- Una segunda pantalla que, recuperando la información de la pantalla previa, muestre una tarjeta personal con los datos del usuario
  - ◆ Esta segunda pantalla deberá comportarse de manera responsive, tanto en vertical como en horizontal

# Ejercicio entregable P5

11:54 Formulario de datos

Nombre  
Luke

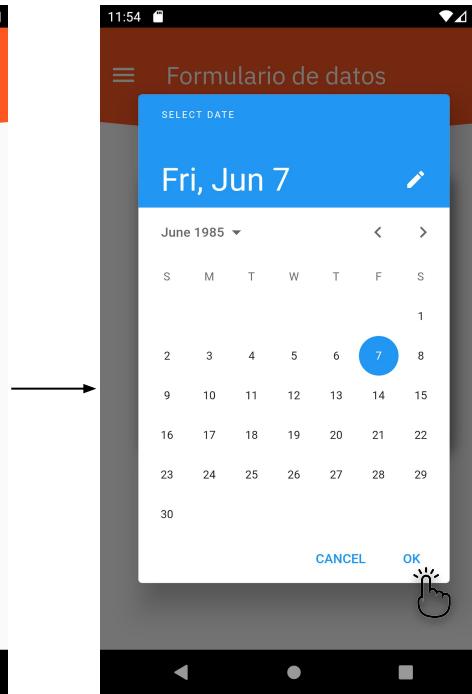
Teléfono  
111222333

Cumpleaños desconocido

Otro  F  M

Género

Confirmar datos



11:58 Formulario de datos

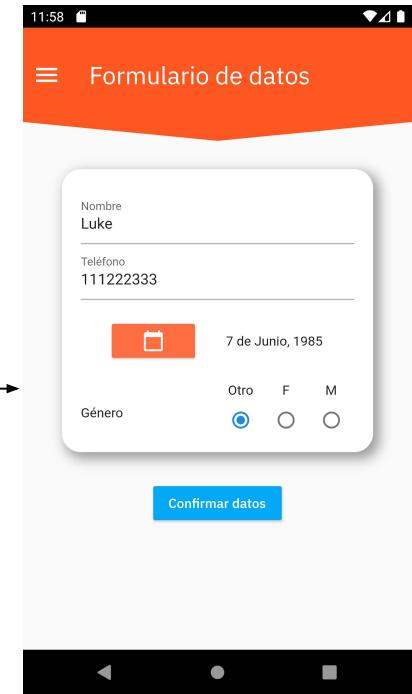
Nombre  
Luke

Teléfono  
111222333

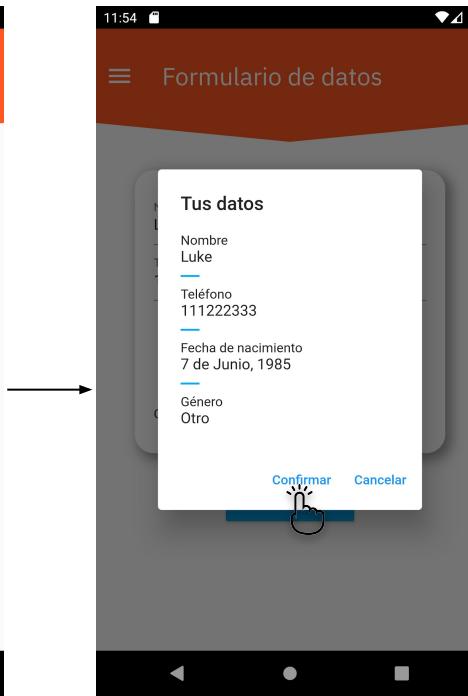
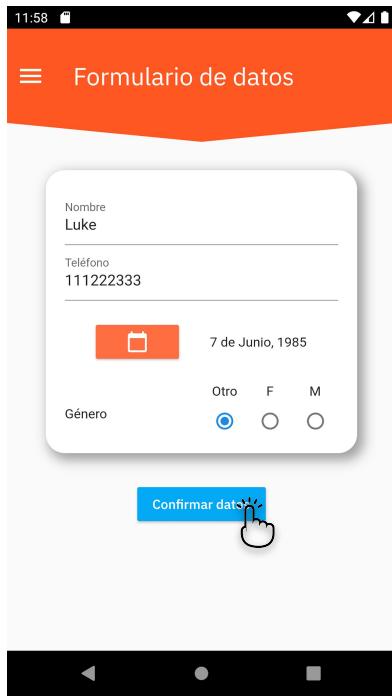
7 de Junio, 1985

Otro  F  M

Confirmar datos



# Ejercicio entregable P5



# Ejercicio entregable P5



# Ejercicio entregable P5

Requisitos de la aplicación:

- 0º A la hora de configurar el proyecto **IMPORTANTE**
  - ◆ Project name ipc\_USUARIO\_p5
  - ◆ Organization epsa.ipc.USUARIO
- Donde **USUARIO** es el nombre de usuario de tu correo de la UPV

# Ejercicio entregable P5

Requisitos de la aplicación:

- **1º Deberás tener 4 ficheros**
  - ◆ **main.dart** Punto de entrada de la app (ya creado)
  - ◆ **form\_page.dart** Página inicial con el formulario
  - ◆ **card\_page.dart** Página con la tarjeta personal
  - ◆ **person\_model.dart** Fichero con el modelo para facilitar el paso de datos de la persona de una pantalla a otra

# Ejercicio entregable P5

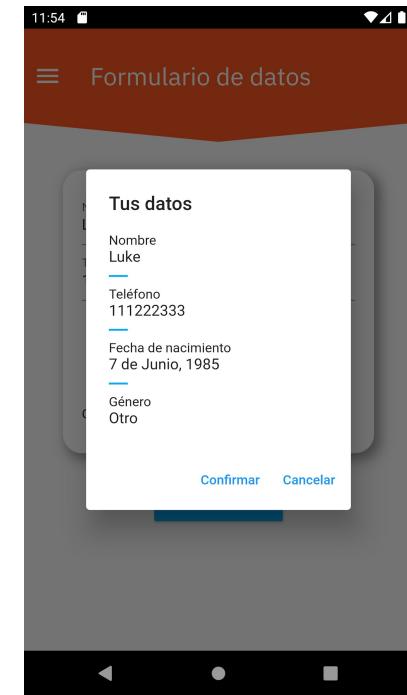
Requisitos de la aplicación:

- 2º La pantalla del formulario deberá recuperar la siguiente información:
  - ◆ Nombre de la persona (TextField)
  - ◆ Teléfono (TextField)
  - ◆ Fecha de nacimiento (Calendario)
  - ◆ Género (u otro dato) (Radio)

# Ejercicio entregable P5

Requisitos de la aplicación:

- **3º Esa misma pantalla mostrará un modal para que el usuario confirme la información introducida, antes de navegar a la siguiente pantalla**



# Ejercicio entregable P5

Requisitos de la aplicación:

- 4º Esa misma pantalla deberá tener una barra de navegación que, opcionalmente, será diseñada con un CustomPainter
  - ◆ El botón del menú de la captura solo es de decoración (sin funcionalidad)



# Ejercicio entregable P5

Requisitos de la aplicación:

- **5º El diseño de la pantalla con la tarjeta del usuario es LIBRE, y deberá ofrecer 2 diseños posibles, dependiendo de la orientación del dispositivo**
  - ◆ Deberá, a parte de la información introducida en el formulario, mostrar una imagen (cuálquiera)
  - ◆ Si deseas simular el diseño proporcionado (imagen circular) puedes utilizar las clases **CircleAvatar ([doc](#))** para la forma circular, y **ClipOval ([doc](#))** para que la imagen no se salga del **CircleAvatar**

# Ejercicio entregable P5

Entregable:

- Se deberán ir haciendo commits a lo largo de la práctica sobre un proyecto de GitLab cuyo nombre será *ipc2021b\_usuarioupv\_p5*
  - ◆ Donde 'usuarioupv' será vuestro nombre de usuario
  - ◆ Deberás añadir a tus profesores de prácticas como miembro del repositorio, con rol de tipo Reporter

Usuario Juanje: @juanje\_upv

Usuario Amando: @amolse

# Ejercicio entregable P5

Entregable:

- Deberás subir en la tarea de Poliforma-T la URL del proyecto en **GitLab** y el **acceso al vídeo** en el que expliques brevemente el código y el funcionamiento en el emulador (3-5 min. máximo)
  - ◆ Si el vídeo es muy pesado, Poliforma-T no te dejará subirlo, pero puedes compartirlo desde Google Drive, Dropbox, subirlo a YouTube o en media.upv.es

# Ejercicio entregable P5

Herramientas de grabación de vídeo:

- [OBS](#)
- Herramientas online como [Apowersoft](#)
- o una reunión en *MS Teams*, compartiendo pantalla