WEEK-6

a) Create custom widgets for specific UI elements.

Aim: To design and implement custom widgets in Flutter for reusable and consistent UI elements, improving code maintainability and enhancing application scalability.

Description:

In Flutter, a **custom widget** allows developers to encapsulate a piece of the user interface into a reusable component. Instead of rewriting the same UI code in multiple places, custom widgets provide a modular approach, making the code cleaner, easier to maintain, and more consistent in design.

By creating custom widgets for specific UI elements—such as buttons, cards, text fields, or icons—we can:

- Improve reusability of code.
- Ensure consistent styling across the application.
- Simplify updates or modifications to the UI.
- Enhance readability and maintainability of the project.

For example, a custom CustomButton widget can be created to standardize button colors, shapes, and behaviors, so any change to the design requires updating only one file instead of multiple screens.

Program:

```
import'package:flutter/material.dart';

void main() => runApp( MaterialApp(
  home:Center( child:Row(
  mainAxisSize:MainAxisSize.min,
        children:[
          CustomIcon(),
          SizedBox(width:8),
          CustomText(),
        ],
      )
    )
  )
);

//custom icon widget class CustomIcon
extends StatelessWidget {
  @override
  Widget build(BuildContext context) { return
  Icon(Icons.favorite,color:Colors.red); }
}

//custom text widget
class CustomText extends StatelessWidget {
```

```
  @override
  Widget build(BuildContext context) { return Text( 'Custom
    Widget!',
    style:TextStyle(fontSize:18,fontWeight:FontWeight.bold),
    );
  }
}
```

OUTPUT:

b) Apply styling using themes and custom styles.

Aim: To implement consistent visual design across a Flutter application by using themes and custom styles, enabling easier UI management and improved user experience.

Description:

Flutter's **Theme** system allows developers to define a unified set of colors, fonts, and visual properties that can be applied throughout the application. Instead of styling each widget individually, themes help ensure consistency, reduce redundancy, and make design updates more efficient.

By applying **custom themes** and **styles**, we can:

- Maintain a consistent look and feel across all screens.
- Centralize style definitions for easy updates.
- Improve readability and maintainability of UI code.
- Support light and dark modes effortlessly.

For example, by defining a ThemeData in the MaterialApp, we can set default properties for buttons, text, and app bars. This eliminates the need to repeat styling code for every widget and ensures the design remains uniform even if the app grows in complexity.

Program:

```
import 'package:flutter/material.dart';

void main() => runApp(
  MaterialApp(    theme:ThemeData(      primaryColor:Colors.deepPurple,
iconTheme:IconThemeData(color:Colors.deepPurple,size:36),
textTheme:TextTheme(
bodyMedium:TextStyle(fontWeight:FontWeight.w600,color:Colors.deepPurple),
      ),
    ),
    home:Center(
child:Row(
       mainAxisSize:MainAxisSize.min,              children:<Widget>[  //
Explicitly specify type argument for clarity
         CustomIcon(),
         SizedBox(width:8),
         CustomText(),
        ],
      ),
    ),
  ),
);

//custom Icon widget
class CustomIcon extends StatelessWidget {
  @override
  Widget    build(BuildContext    context)    {
return
```

```
Icon(Icons.star,color:Theme.of(context).iconTheme.color,size:Theme.of(context).iconTheme.size);
  }
}
//custom text widget
class CustomText extends StatelessWidget { // Corrected: changed 'StatelessWidgets' to 'StatelessWidget'
  @override
  Widget build(BuildContext context) {
    return              Text(
'Styled by Theme!',
      style:Theme.of(context).textTheme.bodyMedium,
    );
  }
}
```

Output: