State management

The *state* of a Flutter app refers to all the objects it uses to display its UI or manage system resources. State management is how we organize our app to most effectively access these objects and share them between different widgets.

1. Using a StatefulWidget

The simplest way to manage state is to use a *StatefulWidget*, which stores state within itself. For example, consider the following widget:

```
class MyCounter extends StatefulWidget {
  const MyCounter({super.key});

  @override
  State<MyCounter> createState() => _MyCounterState();
}

class _MyCounterState extends State<MyCounter> {
  int count = 0;

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Count: $count'),
        TextButton(
          onPressed: () {
            setState(() {
              count++;
            });
          },
          child: Text('Increment'),
        )
      ],
    );
  }
}
```

What this code does:

This code creates a *stateful* widget called *MyCounter* that displays **a number and a button.**

Every time the user taps the "**Increment**" button, **the number increases by 1**.

setState() tells Flutter to **rebuild the UI**, updating the text.

This code illustrates two important concepts when thinking about state management:

Encapsulation

The widget that uses *MyCounter* has no visibility into the underlying *count* variable and no means to access or change it.

Object lifecycle

The *_MyCounterState* object and its *count* variable are created the first time that *MyCounter* is built, and exist until it's removed from the screen.

Class Task:

i) Create a *stateful Flutter widget* that represents a simple social media post. The widget should be named *PostCard* and must display a short caption such as *"This is my first post!"*.

ii) Beneath the caption, include a like button that allows the user to interact with the post. The like button should change its appearance depending on the user's action: when the post is not liked, it should show an outlined heart icon, and when the user taps it, the icon should switch to a filled red heart. Tapping the button again should toggle it back to the unliked state.

iii) Display the like count next to the icon so that the user can clearly see the number of likes change as the user interact with the like button.

2. Sharing state between widgets: Using widget constructors

Since Dart objects are passed by *reference*, it's very common for widgets to define the objects they need to use in their *constructor*.

Any state you pass into a widget's constructor can be used to build its UI

```
class MyCounter extends StatelessWidget {
  final int count;
  const MyCounter({super.key, required this.count});

  @override
  Widget build(BuildContext context) {
    return Text('$count');
  }
}
```

What this widget does

- *MyCounter* is a *StatelessWidget*, meaning it cannot change on its own.
- It receives a value (count) from its parent using a constructor.
- Whatever value is passed into count gets displayed inside a *Text* widget.

This makes it obvious for other users of your widget to know what they need to provide in

```
Column(
  children: [
    MyCounter(
      count: count,
    ),
    MyCounter(
      count: count,
    ),
    TextButton(
      child: Text('Increment'),
      onPressed: () {
        setState(() {
          count++;
        });
      },
    )
  ],
)
```