StatefulWidget lifecycle

When a Flutter builds a StatefulWidget, it creates a State object. This object is where all the mutable state for that widget is held.

`The concept of state is defined by two things:`

1. The data used by the widget might change.

2. The data *can't* be read synchronously when the widget is built. (All state must be established by the time the build method is called).

`The lifecycle has the following simplified steps:`

- createState()
- mounted == true
- initState()
- didChangeDependencies()
- build()
- didUpdateWidget()
- setState()
- deactivate()
- dispose()
- mounted == false

Why Are StatefulWidget and State Separate Classes?

In one word: performance.

The tldr version is that State objects are long lived, but StatefulWidgets (and all Widget subclasses) are thrown away and rebuilt whenever configuration changes. It's very inexpensive ie cheap for Flutter to rebuild a mutable widget.

As State isn't blown away on every rebuild, it avoids expensive computations, and gets at the States property, getters, setters etc everytime something is rebuilt frame by frame.

Important is that this is what allows Flutter animations to exist. As State isn't thrown away, it can constantly be rebuilding it's Widget in response to data changes, and when required, if any.

createState()

When Flutter is instructed to build a StatefulWidget, it immediately calls createState() . This method *must* exist. A StatefulWidget rarely needs to be more complicated than this.

```
class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => new _MyHomePageState();
}
```

mounted is true

When createState() creates the state class, a buildContext is assigned to that state.

A BuildContext is, overly simplified, the place in the widget tree in which this widget is placed.

All widgets have a bool this.mounted property. It is turns true when the buildContext is assigned. It is an error to call setState() when a widget is unmounted.

This property is useful when a method on your state calls setState() but it isn't clear when or how often that method will be called. Perhaps its being called in response to a stream updating. You can use if (mounted){... to make sure the State exists before calling setState().

initState()

This is the first method called when the widget is created (after the class constructor).

initState() is called **once and only once**. It must also call super.initState() .

This *@override* method is the best time to:

1. Initialize data that relies on the specific BuildContext for the created instance of the widget.
2. Initialize properties that rely on this widgets 'parent' in the tree.
3. Subscribe to Streams, ChangeNotifiers, or any other object that could change the data on this widget.

```
@override
initState() {
  super.initState();
  // Add listeners to this class
  cartItemStream.listen((data) {
    _updateWidget(data);
  });
}
```

didChangeDependencies()

The didChangeDependencies() method is called immediately after initState() on the first time the widget is built.

It will also be called whenever an object that this widget *depends on data from* is called. For example, if it relies on an InheritedWidget, which updates.

build is **always** called after didChangeDependencies is called, so this is rarely needed. However, this method is the first change you have to call BuildContext.inheritFromWidgetOfExactType. This essentially would make this State 'listen' to changes on a Widget it's inheriting data from.

The docs also suggest that it could be useful if you need to do network calls (or any other expensive action) when an InheritedWidget updates.

build()

This method is called often (think fps + render). It is a required, *@override* and must return a Widget.

Remember that in Flutter all gui is a widget with a child or children, even 'Padding', 'Center'.

didUpdateWidget(Widget oldWidget)

didUpdateWidget() is called if the parent widget changes and has to rebuild this widget (because it needs to give it different data), but it's being rebuilt with the same runtimeType, then this method is called.

This is because Flutter is re-using the state, which is long lived. In this case, required is to initialize some data again, as one would in initState().

If the state's build() method relies on a Stream or other object that can change, unsubscribe from the old object and re-subscribe to the new instance in didUpdateWidget().

This method is basically the replacement for 'initState()' if it is expected the Widget associated with the widgets's state to be rebuilt!

Flutter always called build() after this, so any subsequent further calls to setState is redundant.

```
@override
void didUpdateWidget(Widget oldWidget) {
  if (oldWidget.importantProperty != widget.importantProperty) {
    _init();
  }
}
```

setState()

The 'setState()' method is called often from the Flutter framework itself and from the developer.

It is used to notify the framework that "data has changed", and the widget at this build context should be rebuilt.

setState() takes a callback which **cannot be async**. It is for this reason it can be called often as required, because repainting is cheap

```
void updateProfile(String name) {
 setState(() => this.name = name);
}
```

deactivate()

This is rarely used.

'deactivate()' is called when State is removed from the tree, *but it might be reinserted* before the current frame change is finished. This method exists basically because State objects can be moved from one point in a tree to another.

dispose()

'dispose()' is called when the State object is removed, which is permanent.

This method is where to unsubscribe and cancel all animations, streams, etc.

mounted is false

The state object can never remount, and an error is thrown is setState() is called.