StatefulWidget instances themselves are immutable and store their mutable state either in separate State objects that are created by the createState method, or in objects to which that State subscribes, for example Stream or ChangeNotifier objects, to which references are stored in final fields on the StatefulWidget itself.

The framework calls createState whenever it inflates a StatefulWidget, which means that multiple State objects might be associated with the same StatefulWidget if that widget has been inserted into the tree in multiple places. Similarly, if a StatefulWidget is removed from the tree and later inserted in to the tree again, the framework will call createState again to create a fresh State object, simplifying the lifecycle of State objects.

A StatefulWidget keeps the same State object when moving from one location in the tree to another if its creator used a GlobalKey for its key. Because a widget with a GlobalKey can be used in at most one location in the tree, a widget that uses a GlobalKey has at most one associated element. The framework takes advantage of this property when moving a widget with a global key from one location in the tree to another by grafting the (unique) subtree associated with that widget from the old location to the new location (instead of recreating the subtree at the new location). The State objects associated with StatefulWidget are grafted along with the rest of the subtree, which means the State object is reused (instead of being recreated) in the new location. However, in order to be eligible for grafting, the widget must be inserted into the new location in the same animation frame in which it was removed from the old location.

Performance considerations

There are two primary categories of StatefulWidgets.

The first is one which allocates resources in State.initState and disposes of them in State.dispose, but which does not depend on InheritedWidgets or call State.setState. Such widgets are commonly used at the root of an application or page, and communicate with subwidgets via ChangeNotifiers, Streams, or other such objects. Stateful widgets following such a pattern are relatively cheap (in terms of CPU and GPU cycles), because they are built once then never update. They can, therefore, have somewhat complicated and deep build methods.

The second category is widgets that use State.setState or depend on InheritedWidgets. These will typically rebuild many times during the application's lifetime, and it is therefore important to minimize the impact of rebuilding such a widget. (They may also use State.initState or State.didChangeDependencies and allocate resources, but the important part is that they rebuild.)

There are several techniques one can use to minimize the impact of rebuilding a stateful widget:

- Push the state to the leaves. For example, if your page has a ticking clock, rather than putting the state at the top of the page and rebuilding the entire page each time the clock ticks, create a dedicated clock widget that only updates itself.
- Minimize the number of nodes transitively created by the build method and any widgets it creates. Ideally, a stateful widget would only create a single widget, and that widget would be a RenderObjectWidget. (Obviously this isn't always practical, but the closer a widget gets to this ideal, the more efficient it will be.)
- If a subtree does not change, cache the widget that represents that subtree and re-use it each time it can be used. It is massively more efficient for a widget to be re-used than for a new (but identically-configured) widget to be created. Factoring out the stateful part into a widget that takes a child argument is a common way of doing this. Another caching strategy consists of assigning a widget to a `final` state variable which can be used in the build method.
- Use `const` widgets where possible. (This is equivalent to caching a widget and re-using it.)
- When trying to create a reusable piece of UI, prefer using a widget rather than a helper method. For example, if there was a function used to build a widget, a State.setState call would require Flutter to entirely rebuild the returned wrapping widget. If a Widget was used instead, Flutter would be able to efficiently re-render only those parts that really need to be updated. Even better, if the created widget is `const`, Flutter would short-circuit most of the rebuild work.
- Avoid changing the depth of any created subtrees or changing the type of any widgets in the subtree. For example, rather than returning either the child or the child wrapped in an IgnorePointer, always wrap the child widget in an IgnorePointer and control the IgnorePointer.ignoring property. This is because changing the depth of the subtree requires rebuilding, laying out, and painting the entire subtree, whereas just changing the property will require the least possible change to the render tree (in the case of IgnorePointer, for example, no layout or repaint is necessary at all).

- If the depth must be changed for some reason, consider wrapping the common parts of the subtrees in widgets that have a GlobalKey that remains consistent for the life of the stateful widget. (The KeyedSubtree widget may be useful for this purpose if no other widget can conveniently be assigned the key.)

This is a skeleton of a stateful widget subclass called `YellowBird`.

In this example. the State has no actual state. State is normally represented as private member fields. Also, normally widgets have more constructor arguments, each of which corresponds to a `final` property.

```
class YellowBird extends StatefulWidget {
  const YellowBird({ Key? key }) : super(key: key);

  @override
  State<YellowBird> createState() => _YellowBirdState();
}

class _YellowBirdState extends State<YellowBird> {
  @override
  Widget build(BuildContext context) {
    return Container(color: const Color(0xFFFFE306));
  }
}
```

This example shows the more generic widget `Bird` which can be given a color and a child, and which has some internal state with a method that can be called to mutate it:

```
class Bird extends StatefulWidget {
  const Bird({
    Key? key,
    this.color = const Color(0xFFFFE306),
    this.child,
  }) : super(key: key);

  final Color color;
  final Widget? child;

  @override
```