LeanCode

Giving context to BuildContext

Plan for today

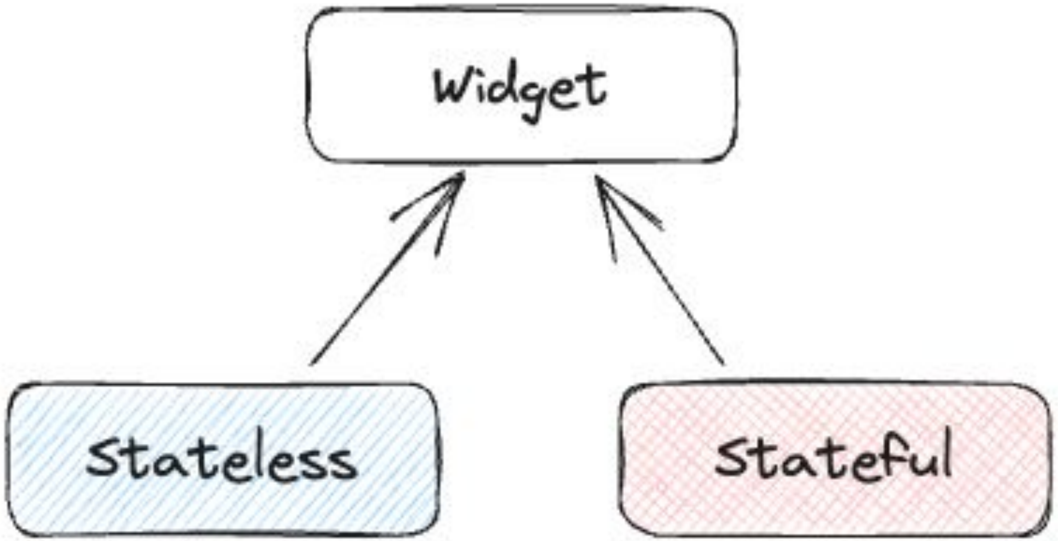
- introduce interactivity
- write some stateful widgets
- keep DRY with inherited widgets
- solve mysteries of BuildContext
- answer how many trees make a forest

The problem:

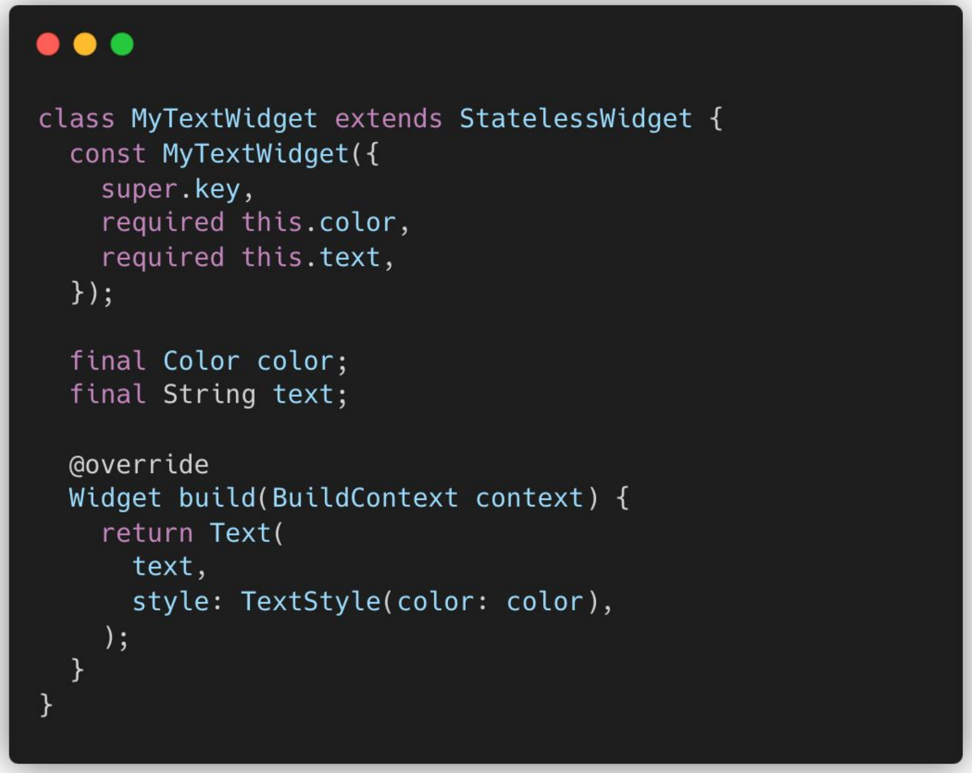
our last app was pretty static

https://flutter-at-mini-labs-w3.web.app/

Enter StatefulWidget


Widget
Stateless
Stateful

Making a widget stateful

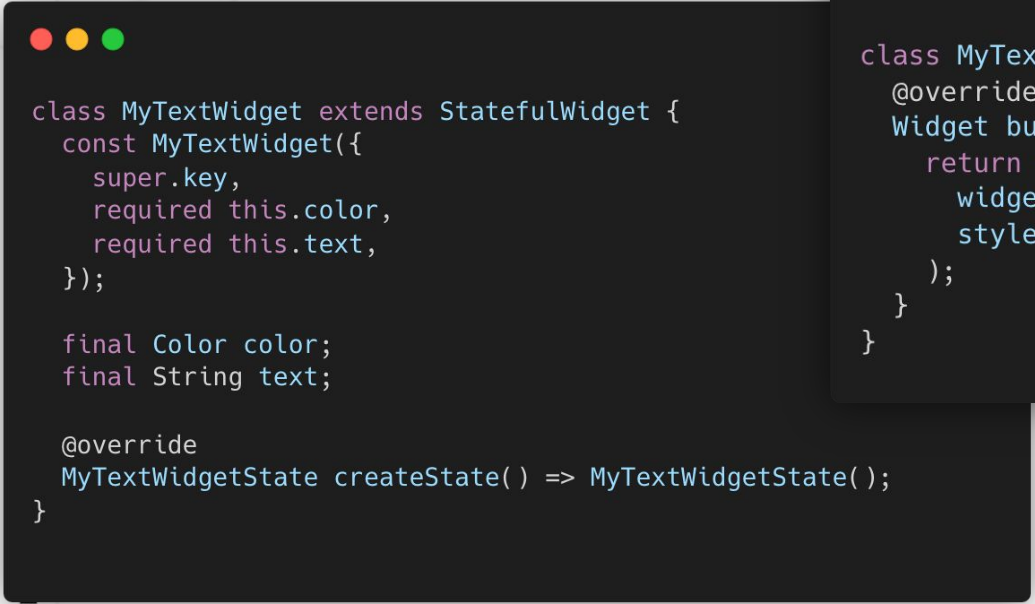


```
class MyTextWidget extends StatelessWidget {
  const MyTextWidget({
    super.key,
    required this.color,
    required this.text,
  });

  final Color color;
  final String text;

  @override
  Widget build(BuildContext context) {
    return Text(
      text,
      style: TextStyle(color: color),
    );
  }
}
```

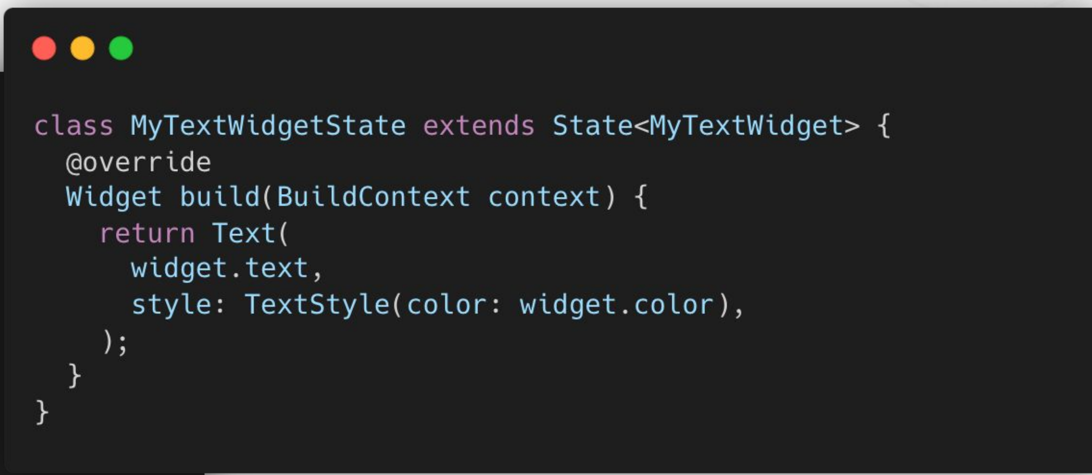
We build digital products.

LeanCode

Making a widget stateful

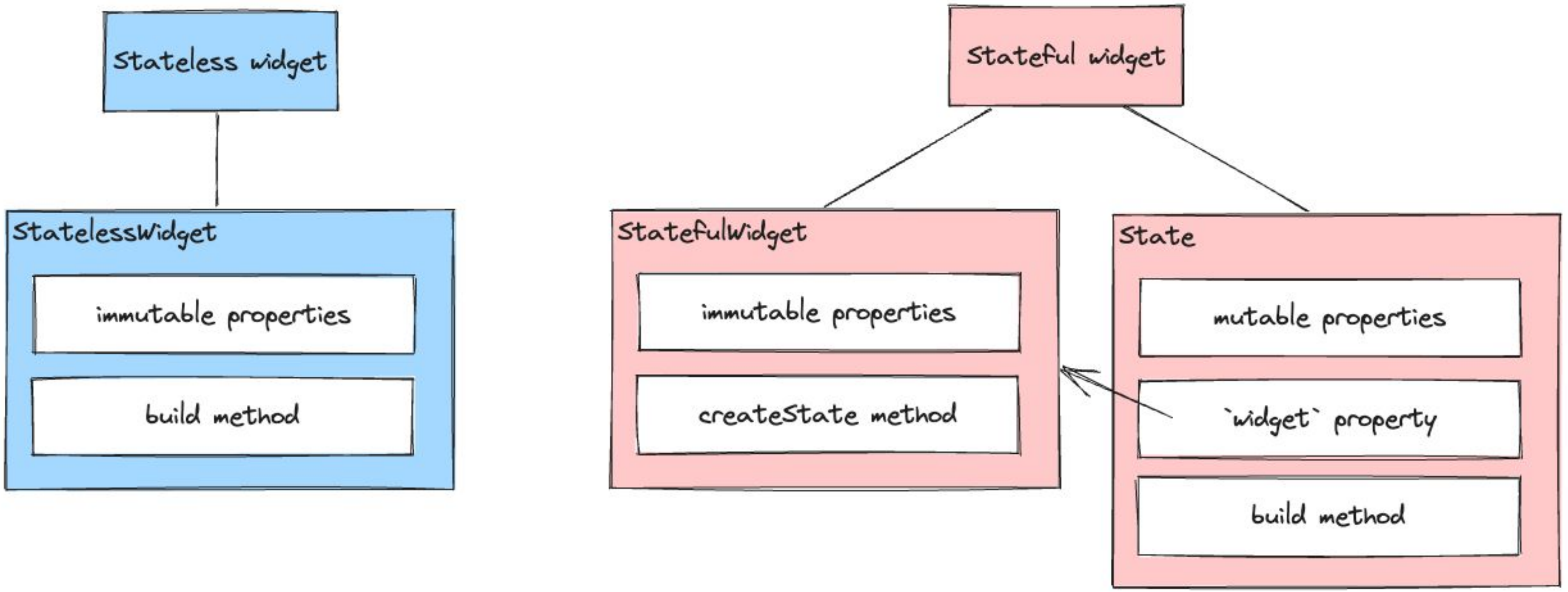


```dart
class MyTextWidget extends StatefulWidget {
  const MyTextWidget({
    super.key,
    required this.color,
    required this.text,
  });

  final Color color;
  final String text;

  @override
  MyTextWidgetState createState() => MyTextWidgetState();
}
```



```dart
class MyTextWidgetState extends State<MyTextWidget> {
  @override
  Widget build(BuildContext context) {
    return Text(
      widget.text,
      style: TextStyle(color: widget.color),
    );
  }
}
```

Making a widget stateful



Making a widget stateful

Note the differences from StatelessWidget:

- stateful widgets consist of **two** classes: StatefulWidget and State
- the build method is in State, not in StatefulWidget
- properties on StatefulWidget are immutable – you can’t change them
- the actual state (mutable properties) reside in State

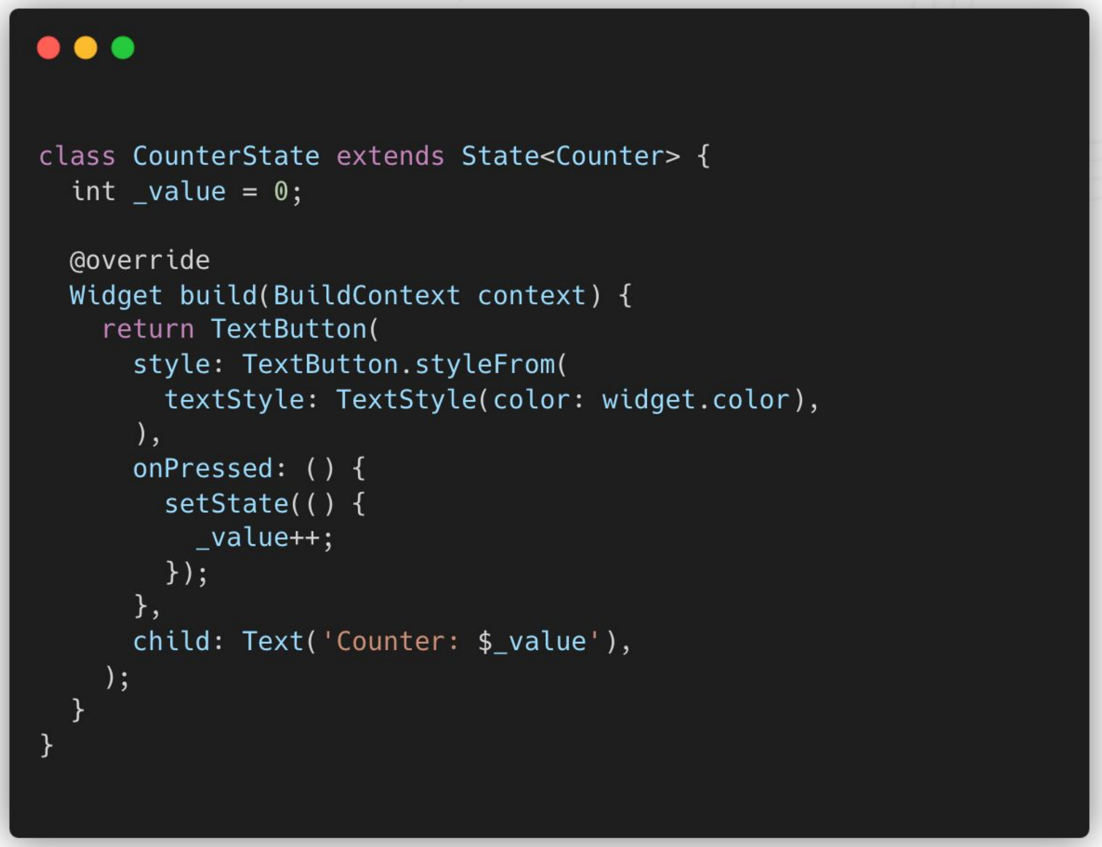
Widgets come and go, State persists

We build digital products.

LeanCode

Changing state

use the setState method

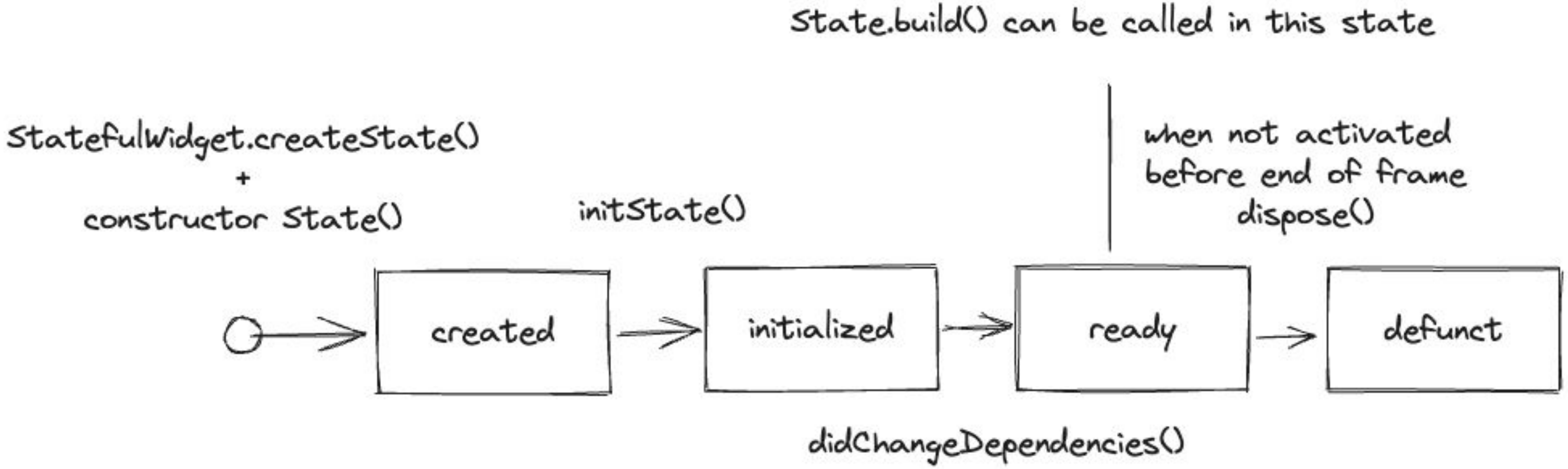


```dart
class CounterState extends State<Counter> {
  int _value = 0;

  @override
  Widget build(BuildContext context) {
    return TextButton(
      style: TextButton.styleFrom(
        textStyle: TextStyle(color: widget.color),
      ),
      onPressed: () {
        setState(() {
          _value++;
        });
      },
      child: Text('Counter: $_value'),
    );
  }
}
```

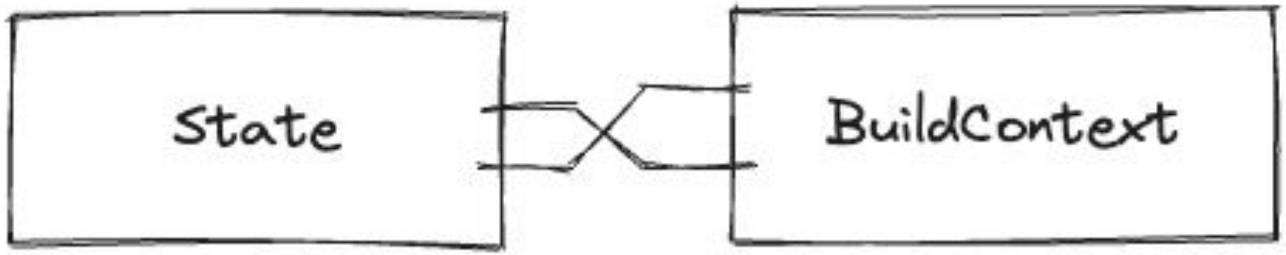
We build digital products.

LeanCode

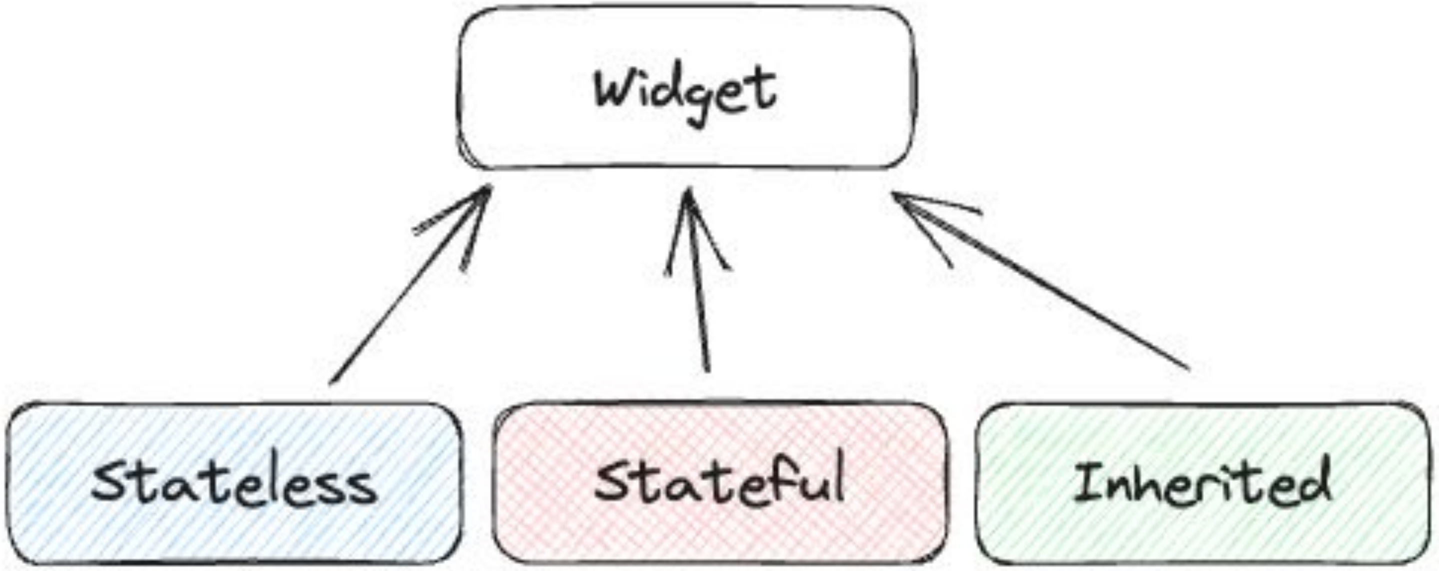
State's lifecycle



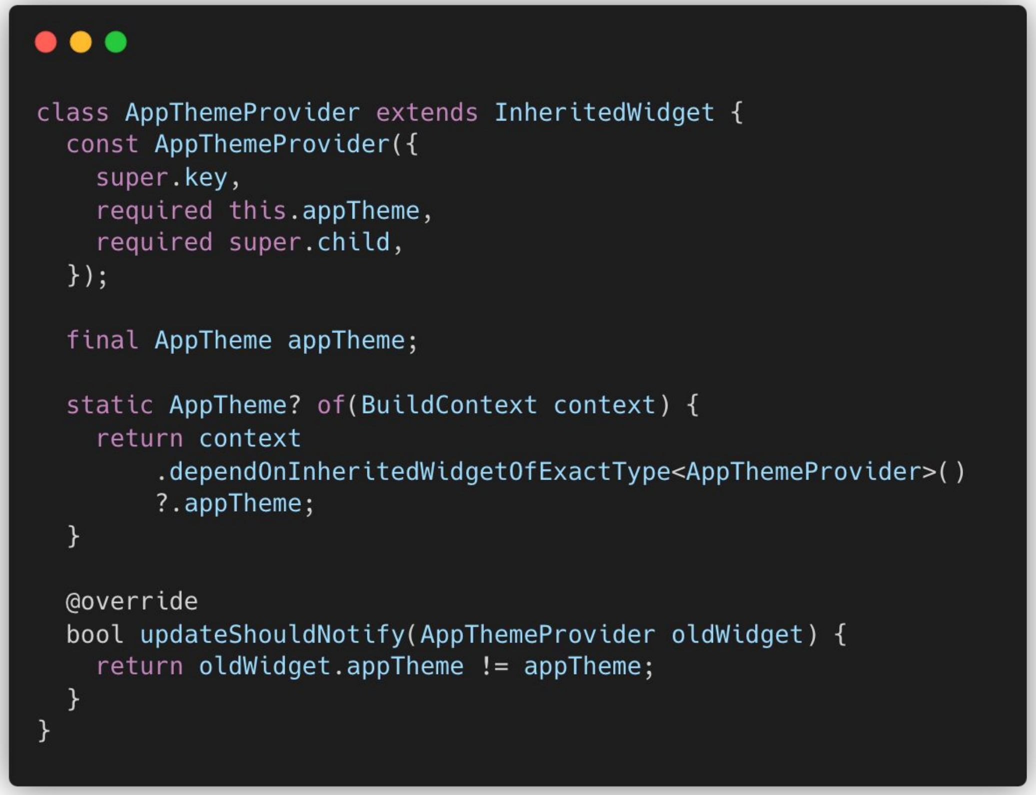
State is bound to a single BuildContext



InheritedWidget

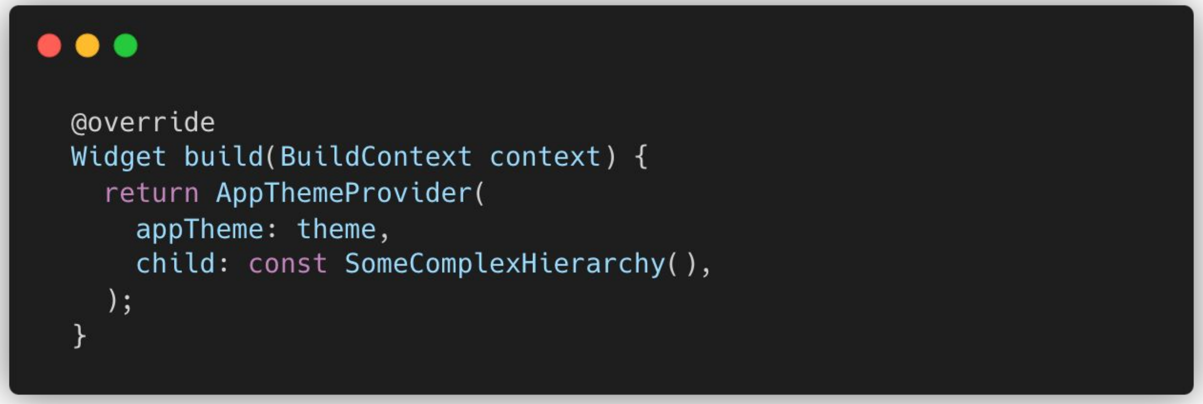

Widget
Stateless
Stateful
Inherited

InheritedWidget



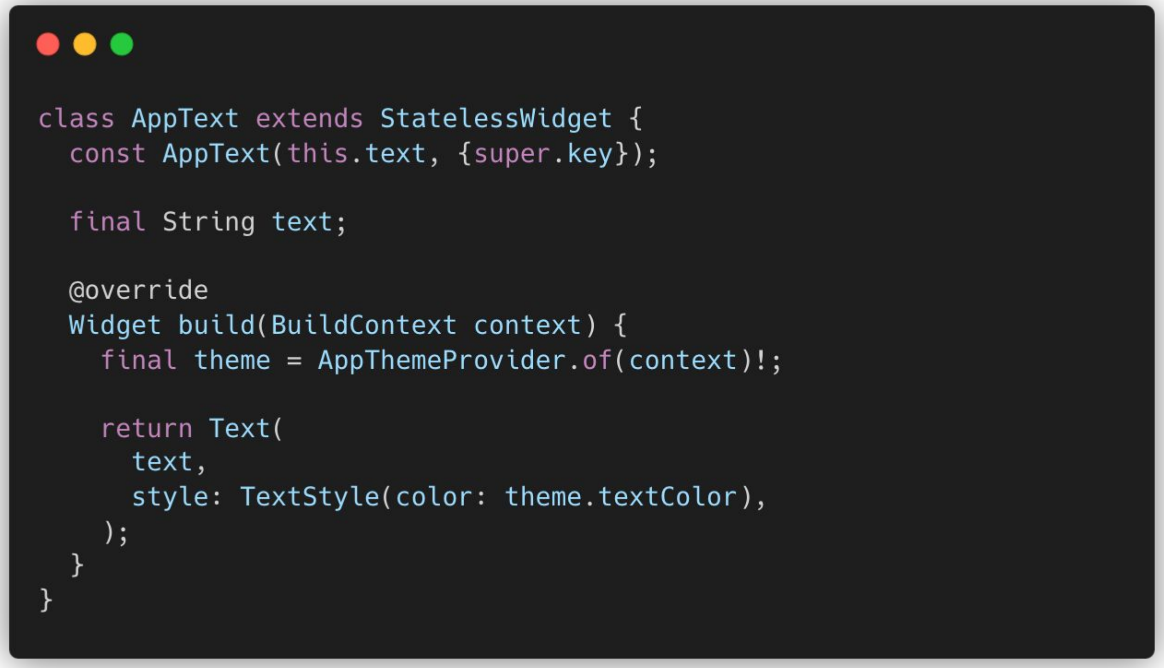
```
class AppThemeProvider extends InheritedWidget {
  const AppThemeProvider({
    super.key,
    required this.appTheme,
    required super.child,
  });

  final AppTheme appTheme;

  static AppTheme? of(BuildContext context) {
    return context
        .dependOnInheritedWidgetOfExactType<AppThemeProvider>()
        ?.appTheme;
  }

  @override
  bool updateShouldNotify(AppThemeProvider oldWidget) {
    return oldWidget.appTheme != appTheme;
  }
}
```

InheritedWidget



```
@override
Widget build(BuildContext context) {
  return AppThemeProvider(
    appTheme: theme,
    child: const SomeComplexHierarchy(),
  );
}
```

InheritedWidget



```dart
class AppText extends StatelessWidget {
  const AppText(this.text, {super.key});

  final String text;

  @override
  Widget build(BuildContext context) {
    final theme = AppThemeProvider.of(context)!;

    return Text(
      text,
      style: TextStyle(color: theme.textColor),
    );
  }
}
```

We build digital products.

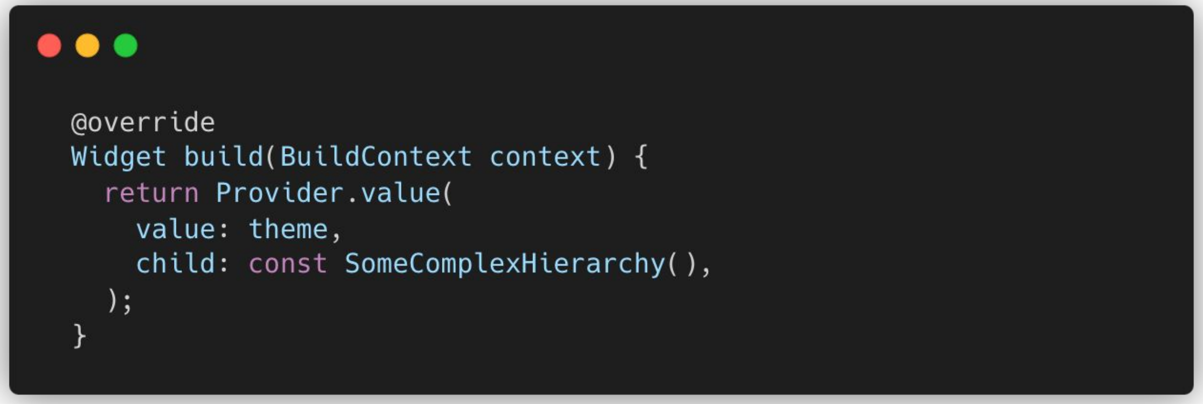
LeanCode

Provider [pub.dev]

InheritedWidget but easier

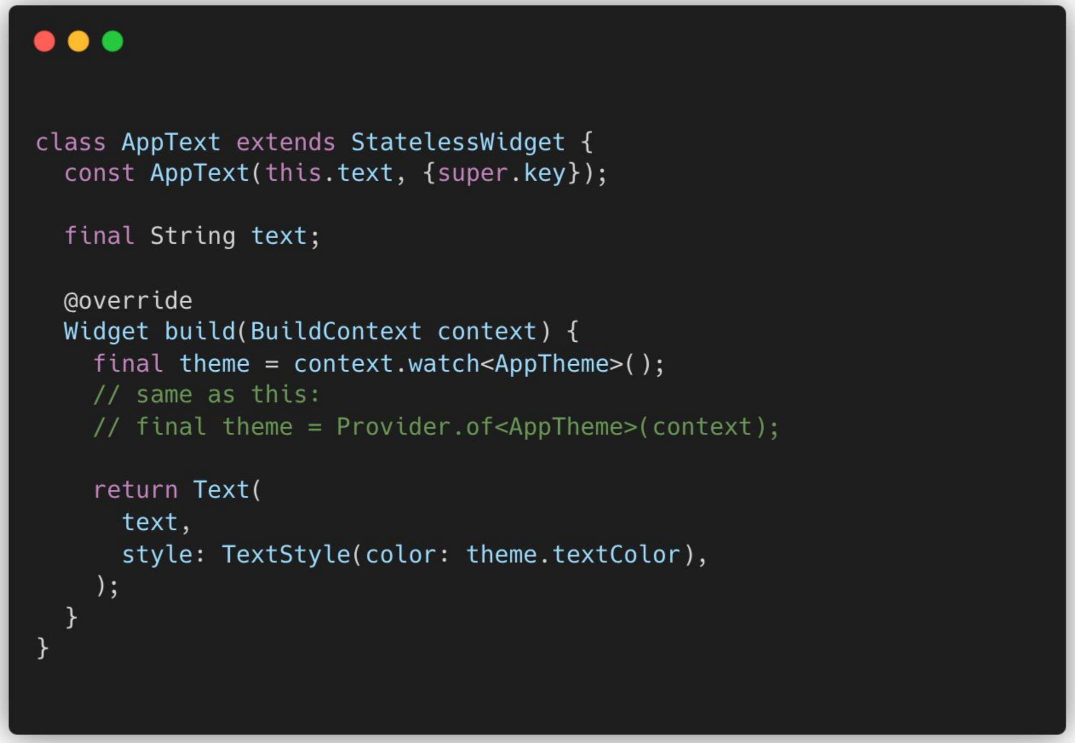
Provider

No need to define custom InheritedWidget
→ use ready-made, generic Provider



```
@override
Widget build(BuildContext context) {
  return Provider.value(
    value: theme,
    child: const SomeComplexHierarchy(),
  );
}
```

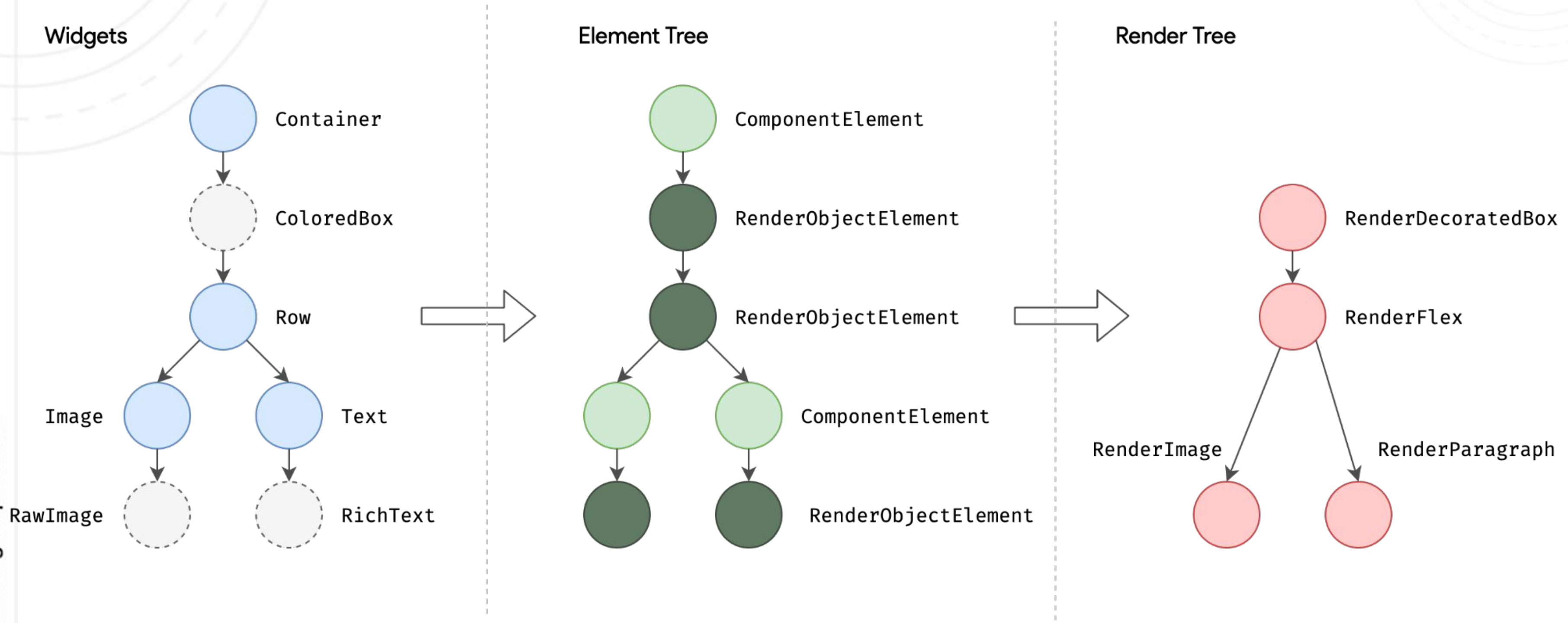
Provider



```dart
class AppText extends StatelessWidget {
  const AppText(this.text, {super.key});

  final String text;

  @override
  Widget build(BuildContext context) {
    final theme = context.watch<AppTheme>();
    // same as this:
    // final theme = Provider.of<AppTheme>(context);

    return Text(
      text,
      style: TextStyle(color: theme.textColor),
    );
  }
}
```

Putting it together

The trees



Source: https://docs.flutter.dev/resources/architectural-overview#layout-and-rendering

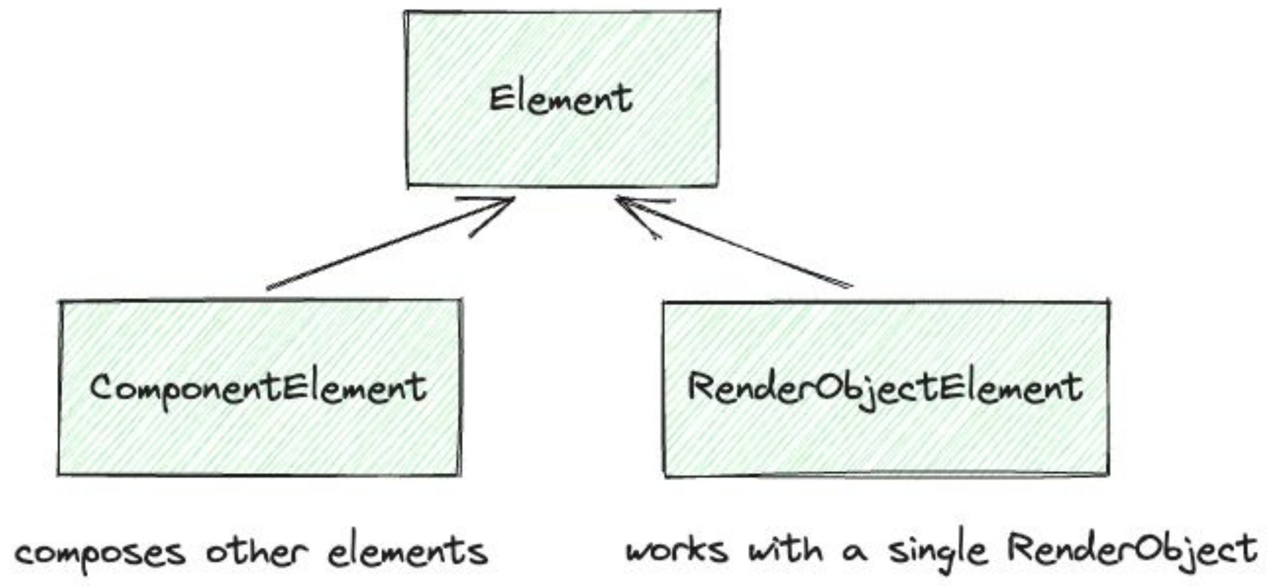
Widgets

- are immutable – never change; always describe the same thing
- are configuration for elements
- are recipes for subtrees, parts of application
- are not associated with any specific part of the tree
- are (almost) pure, plain data objects
- think cheap, lightweight, disposable, ephemeral, impermanent
- one instance can be reused in multiple places; one instance can configure multiple elements

Elements

❝ An instantiation of a Widget at a particular location in the tree.

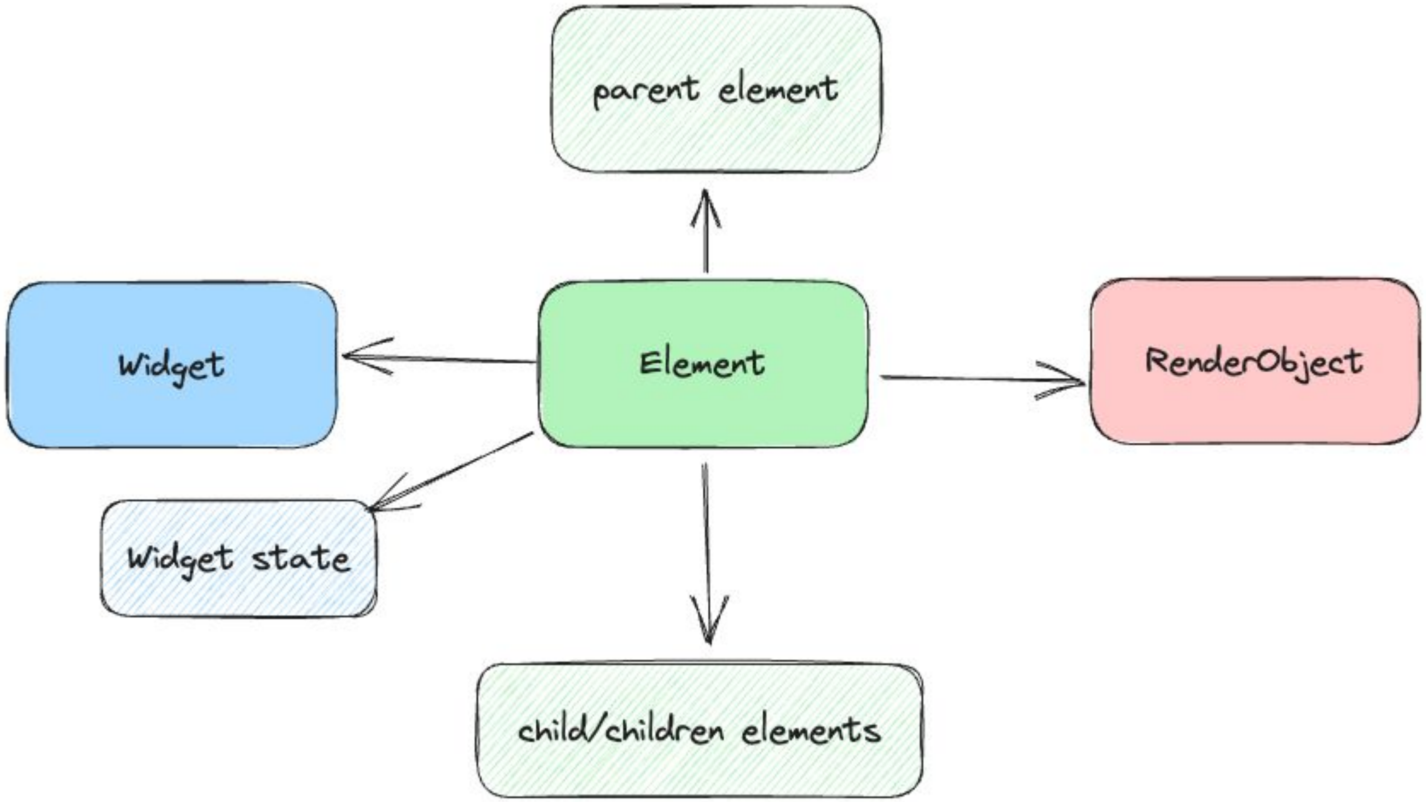
[Element class - widgets library - Dart API]



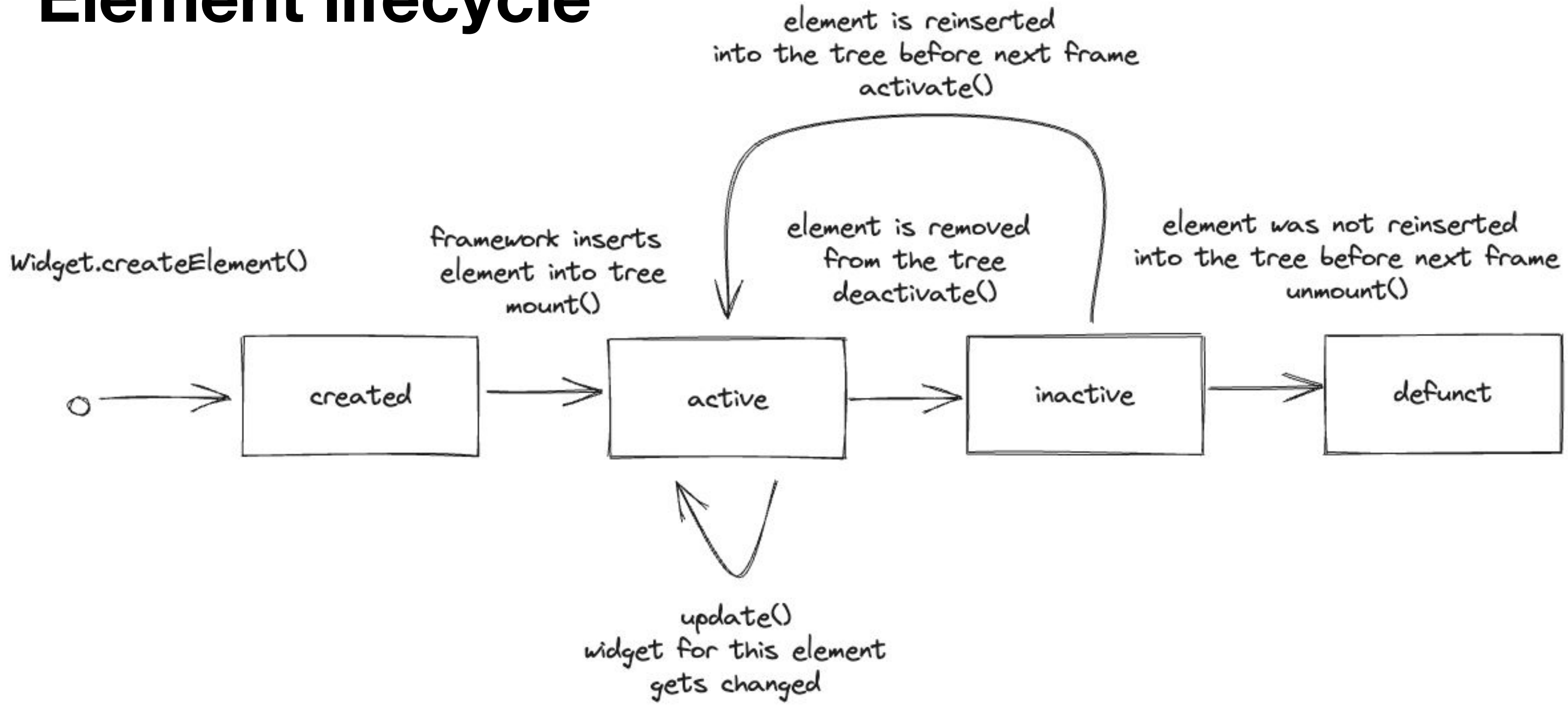
Elements

- are associated with a specific location in the tree
- persistent across rebuilds
- have a lifecycle
- can access ancestors and descendants
- can be reconfigured with a different widget
- keep track of their widgets and their state (if widget is stateful)

Elements


parent element
Widget
Element
RenderObject
Widget state
child/children elements

Element lifecycle



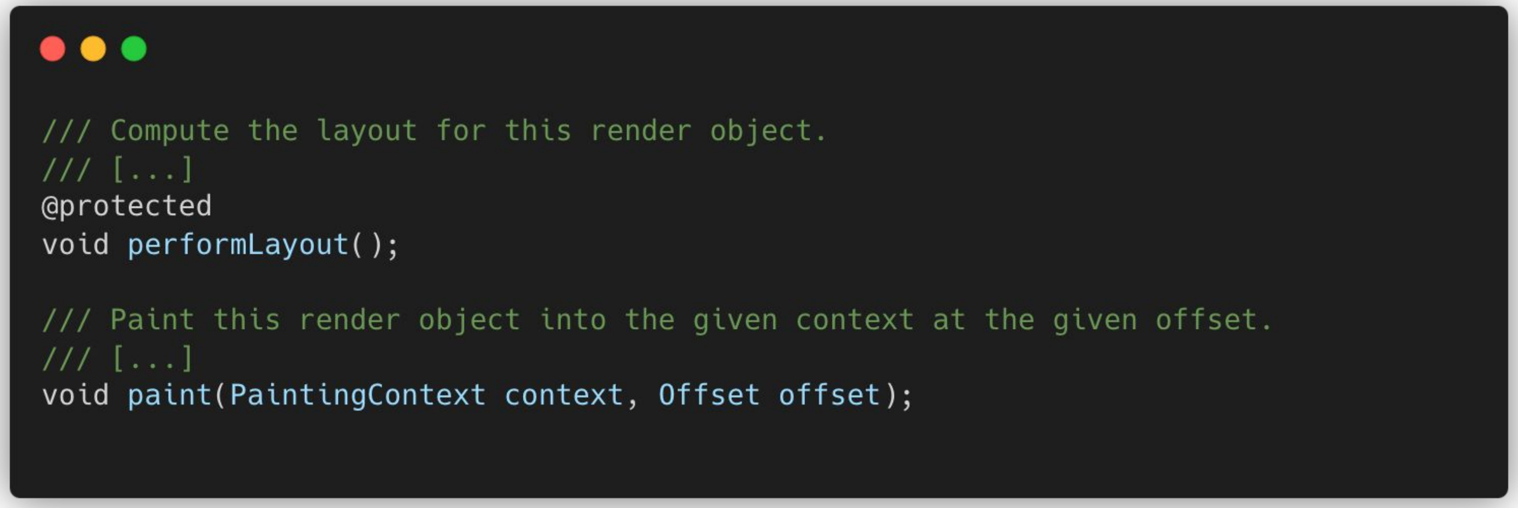
RenderObjects

Responsibilities:

- layout
- painting
- semantics (accessibility)

- operate in terms of a canvas

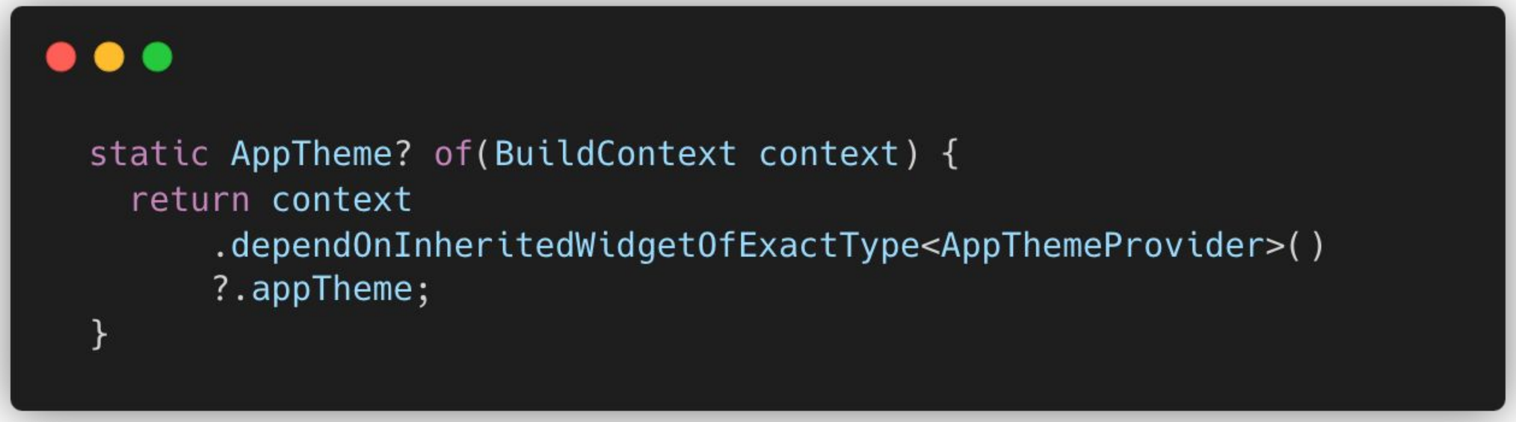
RenderObjects



```
/// Compute the layout for this render object.
/// [...]
@protected
void performLayout();

/// Paint this render object into the given context at the given offset.
/// [...]
void paint(PaintingContext context, Offset offset);
```

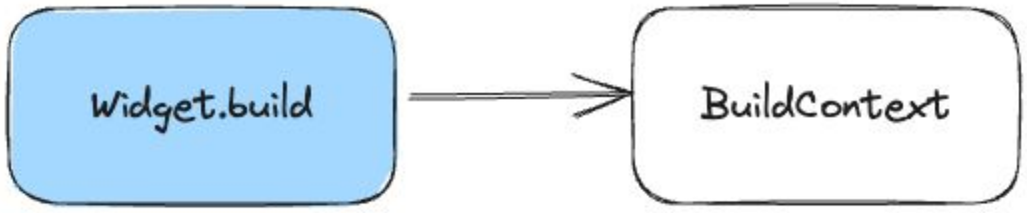
Wait a second

context can obtain something from the tree?

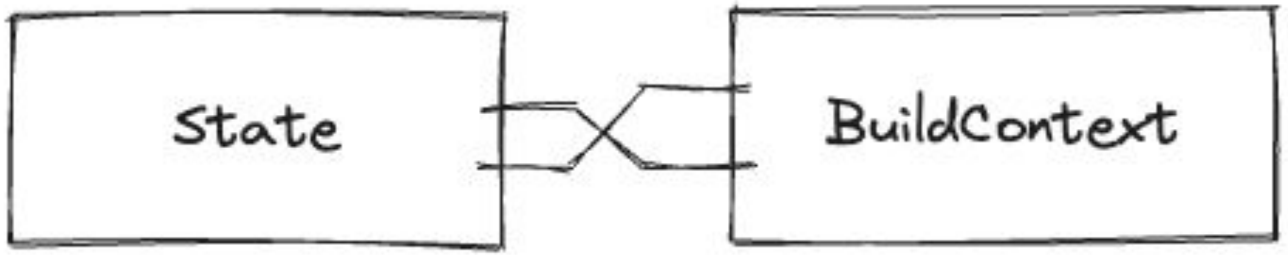


```
static AppTheme? of(BuildContext context) {
  return context
      .dependOnInheritedWidgetOfExactType<AppThemeProvider>()
      ?.appTheme;
}
```

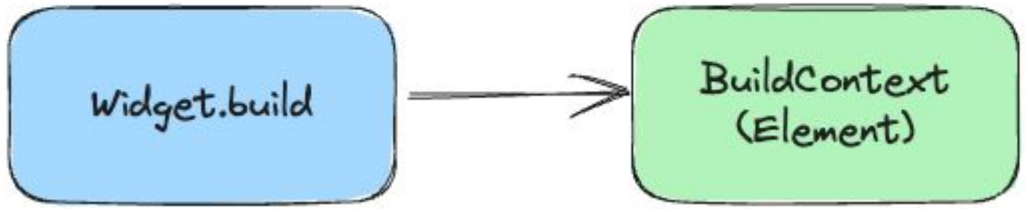
BuildContext



State is bound to a single BuildContext



BuildContext



BuildContext is an Element

Keys

We build digital products.

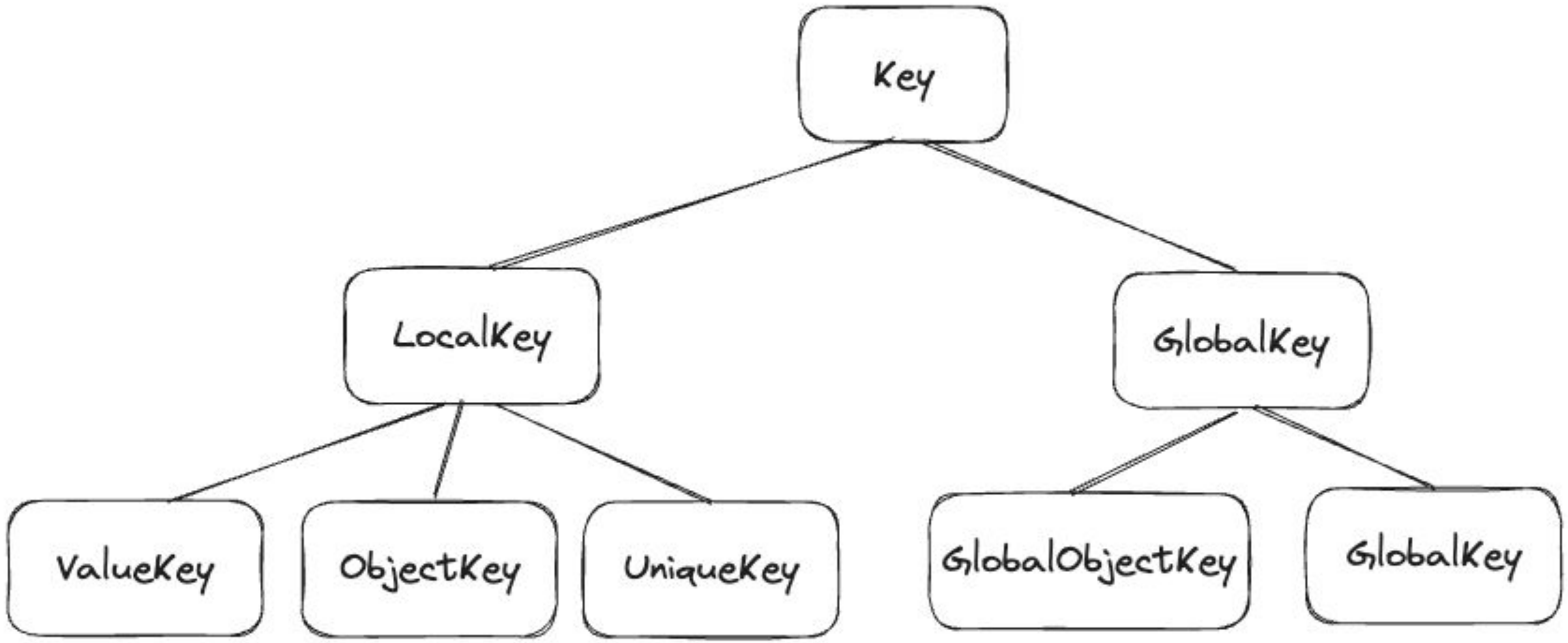
LeanCode

Keys

Useful info here, including 10-min youtube video

Key class - foundation library - Dart API (flutter.dev)

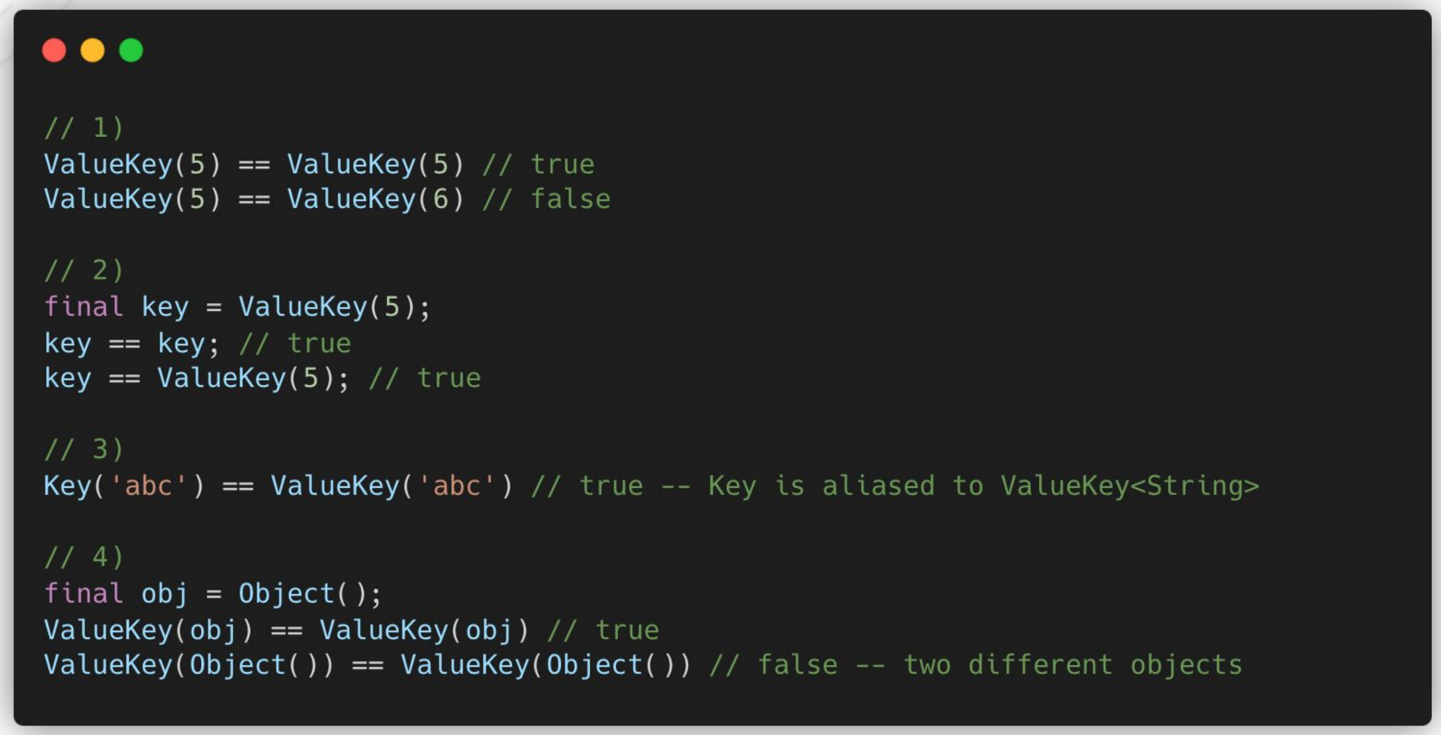
Key types


Key
LocalKey
GlobalKey
ValueKey
ObjectKey
UniqueKey
GlobalObjectKey
GlobalKey

Key types

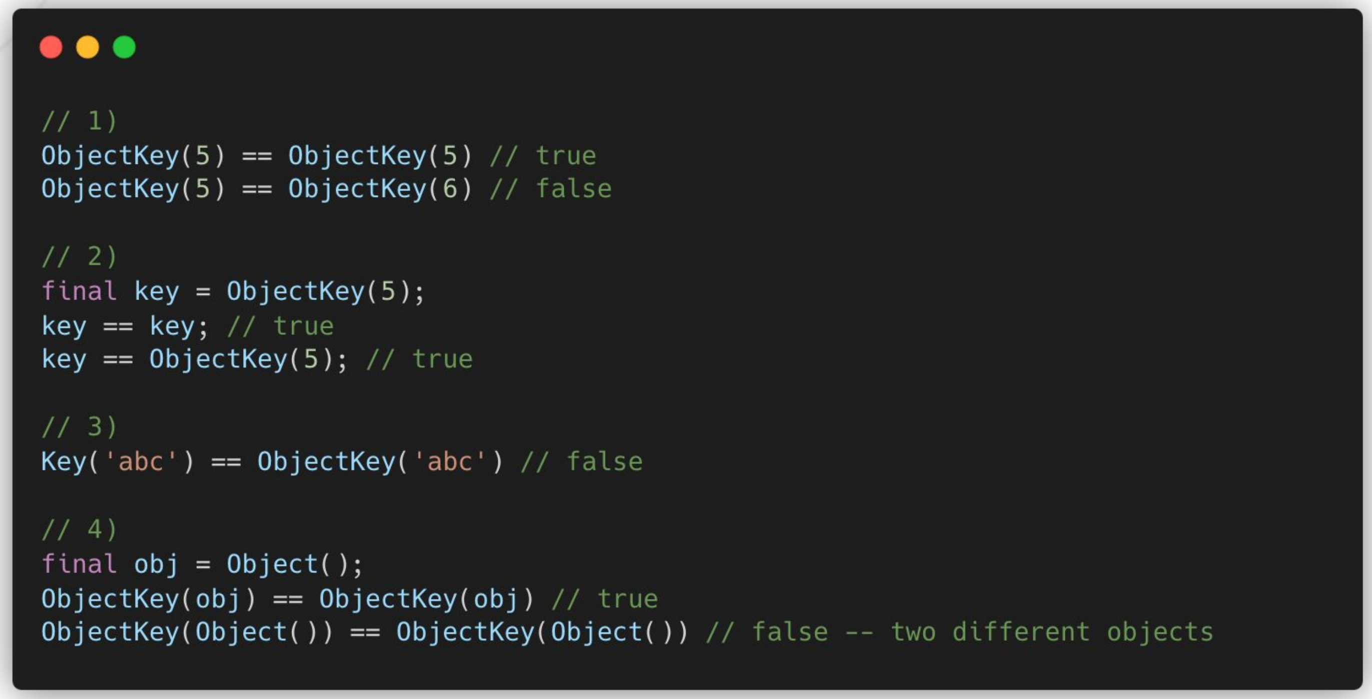
- ValueKey(value) – compares inner values by == (which can be overloaded)
- ObjectKey(value) – compares inner values by `identical` (referential equality)
- UniqueKey() – does not hold any value; compares itself to another key by reference
- Key(string) – shorthand for ValueKey<String>(string)

Key types



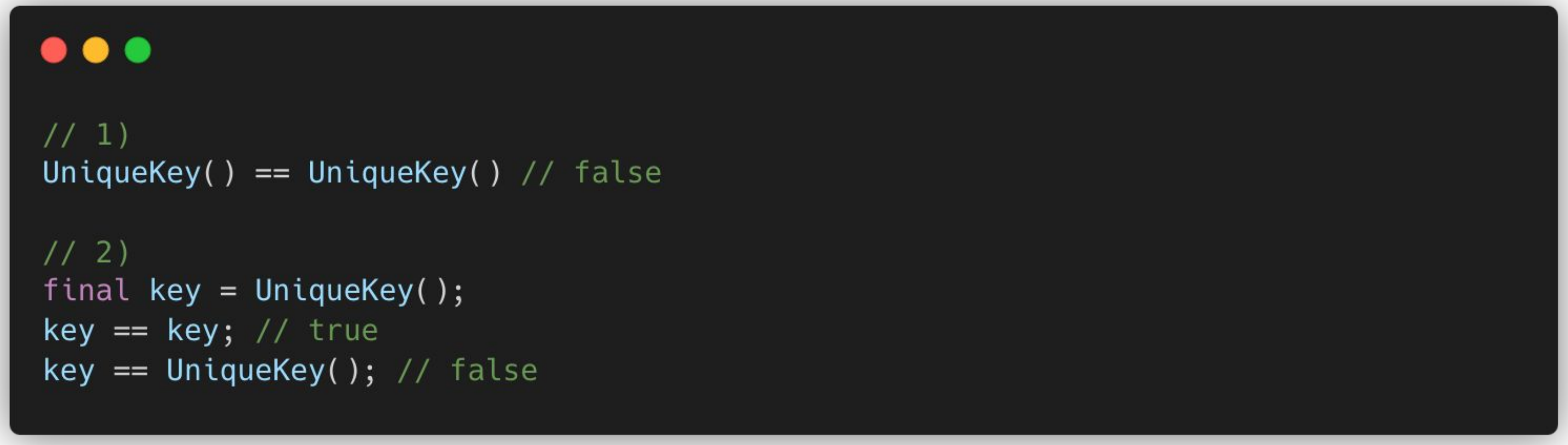
```
// 1)
ValueKey(5) == ValueKey(5) // true
ValueKey(5) == ValueKey(6) // false

// 2)
final key = ValueKey(5);
key == key; // true
key == ValueKey(5); // true

// 3)
Key('abc') == ValueKey('abc') // true -- Key is aliased to ValueKey<String>

// 4)
final obj = Object();
ValueKey(obj) == ValueKey(obj) // true
ValueKey(Object()) == ValueKey(Object()) // false -- two different objects
```

Key types



```
// 1)
ObjectKey(5) == ObjectKey(5) // true
ObjectKey(5) == ObjectKey(6) // false

// 2)
final key = ObjectKey(5);
key == key; // true
key == ObjectKey(5); // true

// 3)
Key('abc') == ObjectKey('abc') // false

// 4)
final obj = Object();
ObjectKey(obj) == ObjectKey(obj) // true
ObjectKey(Object()) == ObjectKey(Object()) // false -- two different objects
```

Key types



```
// 1)
UniqueKey() == UniqueKey() // false

// 2)
final key = UniqueKey();
key == key; // true
key == UniqueKey(); // false
```

Global keys

- GlobalKey() – like UniqueKey() compares key instances by reference
- GlobalObjectKey(obj) – like ObjectKey(obj) compares key types and inner object by reference

Use cases:

- transition widget between screens, maintaining state – Hero transition
- access state from outside via GlobalKey.currentState
- access element from outside via GlobalKey.currentContext

Good practices

We build digital products.

LeanCode

Prefer stateless to stateful

StatelessWidget is:

- cheaper in terms of performance (negligible; might make a difference at scale)
- more concise
- less effort to write
- vscode has quick actions to convert stateless ←→ stateful anyway

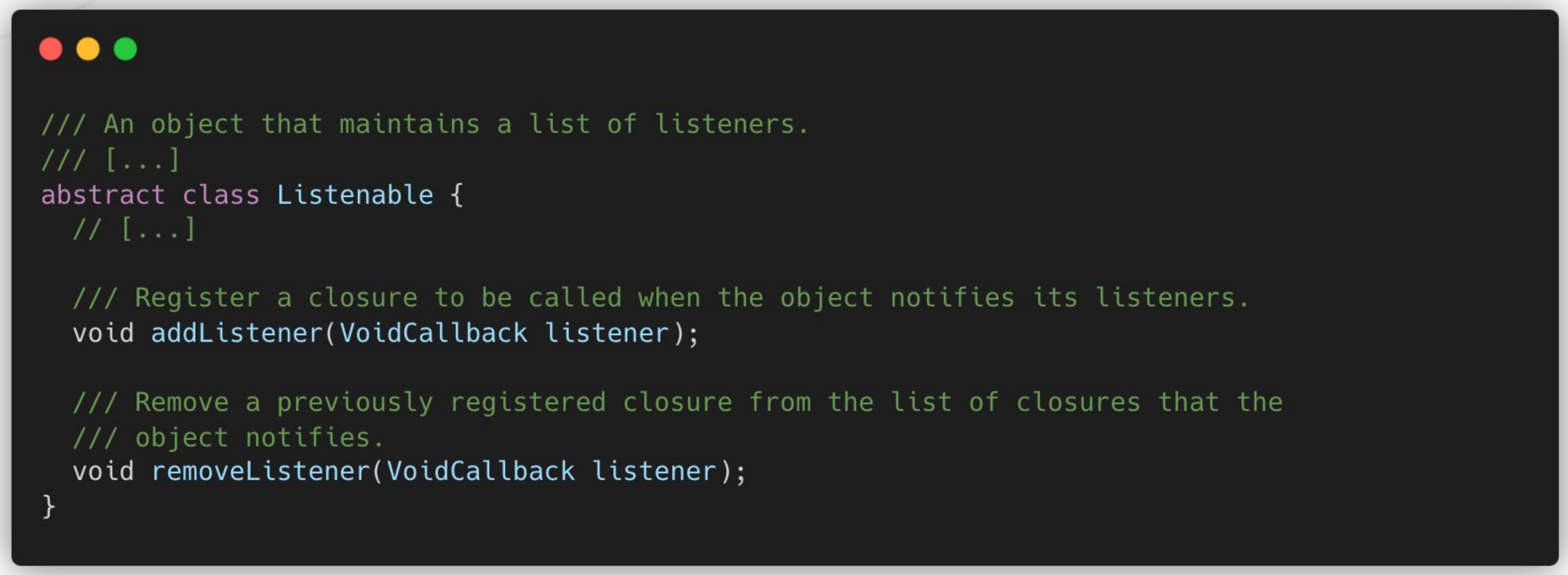
Don’t overuse Provider/InheritedWidget

Passing values implicitly via InheritedWidget or Provider might be confusing to a code reviewer/maintainer/reader.

InheritedWidget forfeits static analysis of passing props directly and might result in runtime errors.

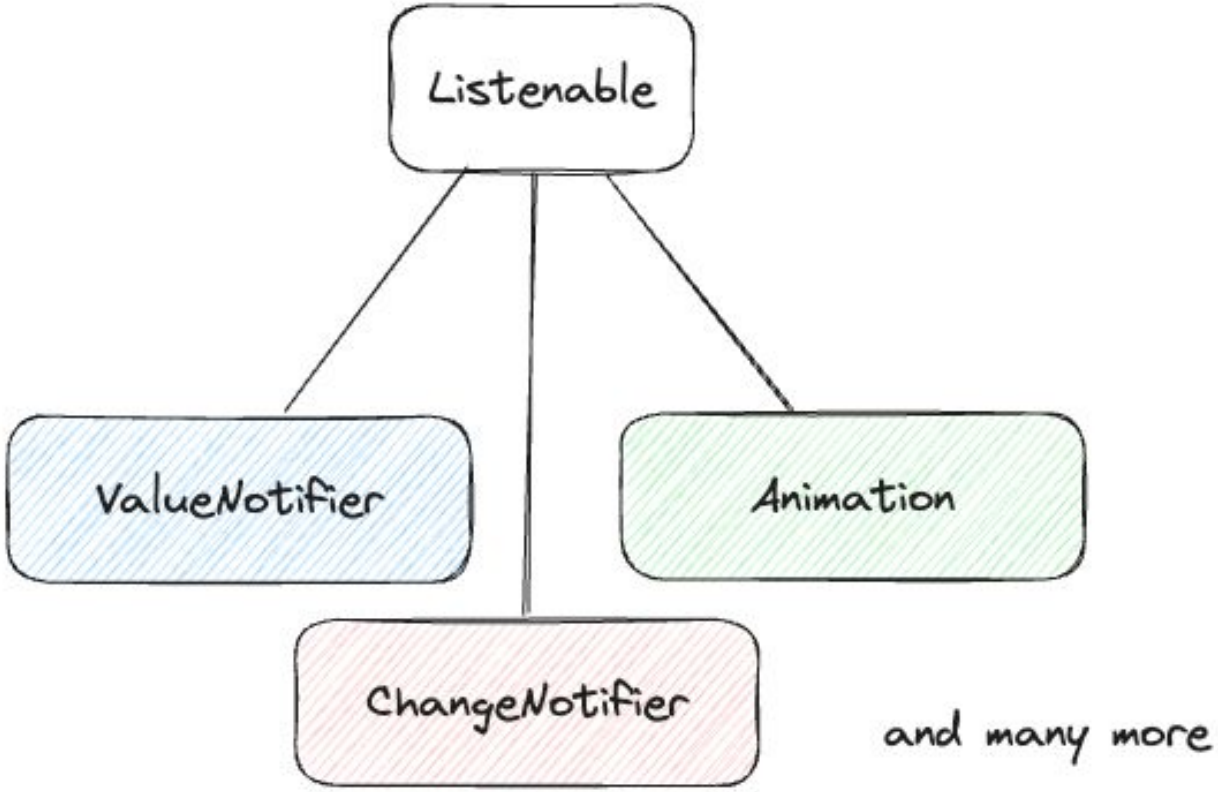
A note on observing state

Listenable

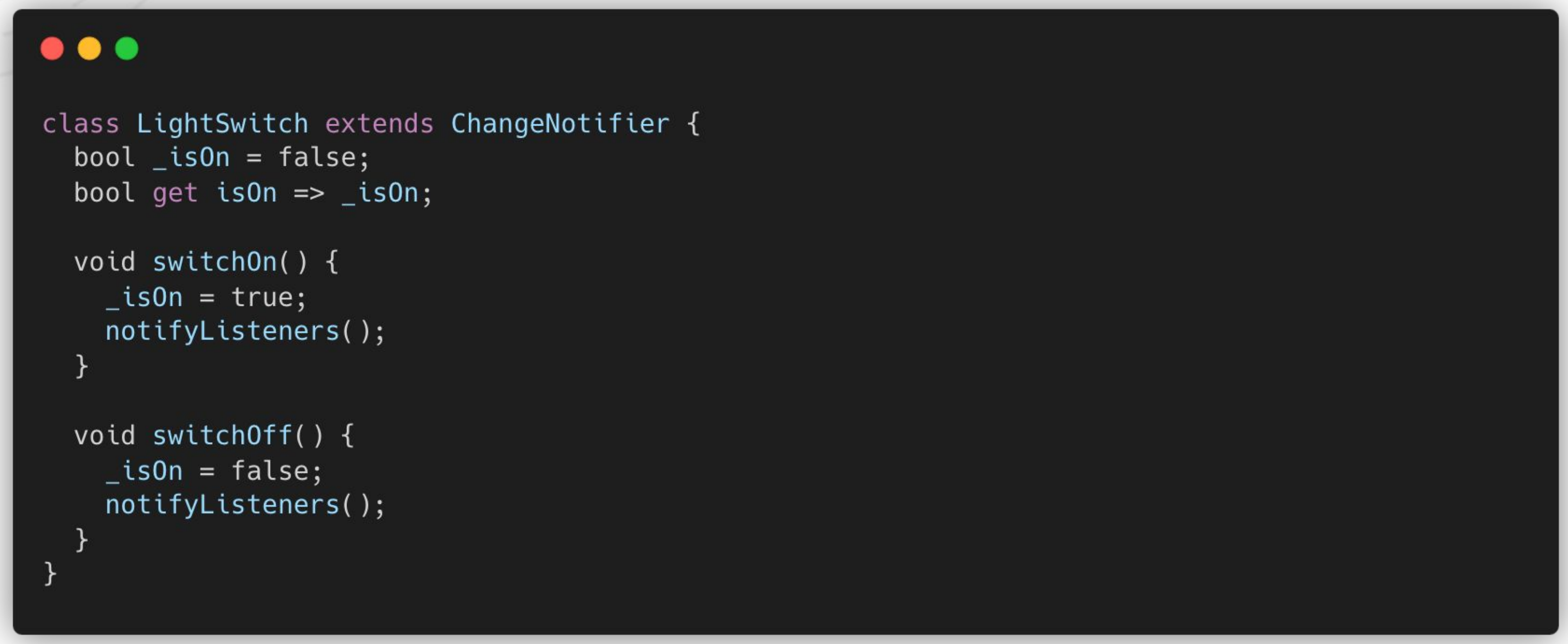


```
/// An object that maintains a list of listeners.
/// [...]
abstract class Listenable {
  // [...]

  /// Register a closure to be called when the object notifies its listeners.
  void addListener(VoidCallback listener);

  /// Remove a previously registered closure from the list of closures that the
  /// object notifies.
  void removeListener(VoidCallback listener);
}
```

Types of listenables

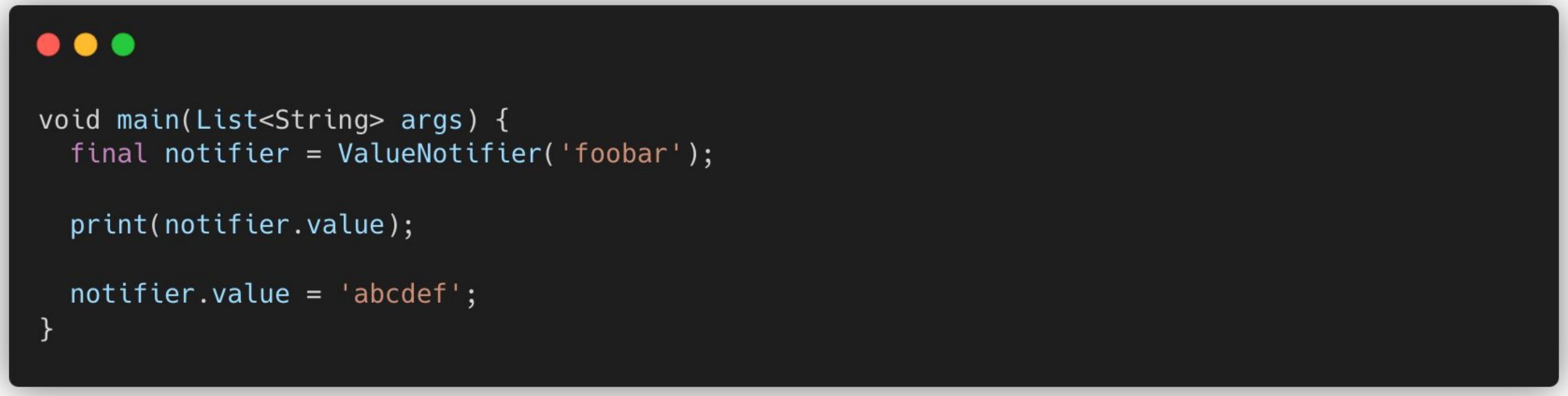


ChangeNotifier



```
class LightSwitch extends ChangeNotifier {
  bool _isOn = false;
  bool get isOn => _isOn;

  void switchOn() {
    _isOn = true;
    notifyListeners();
  }

  void switchOff() {
    _isOn = false;
    notifyListeners();
  }
}
```

ValueNotifier

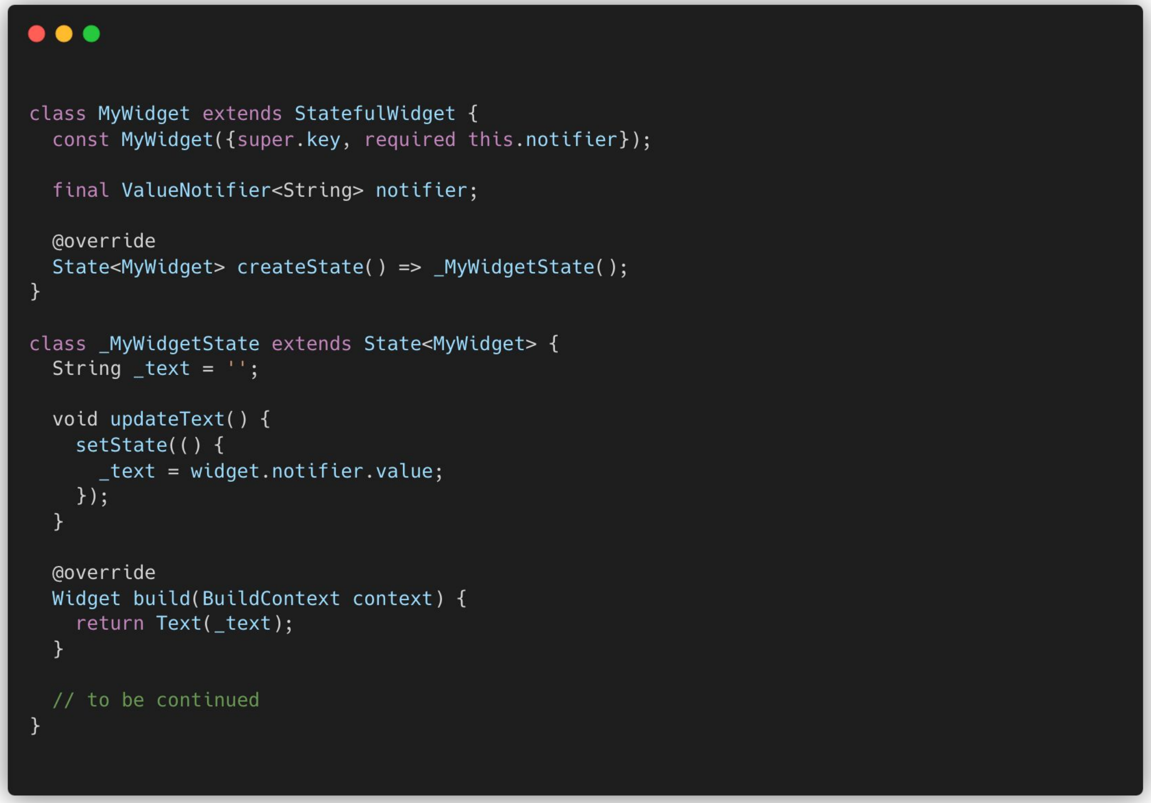


```
void main(List<String> args) {
  final notifier = ValueNotifier('foobar');

  print(notifier.value);

  notifier.value = 'abcdef';
}
```

We build digital products.

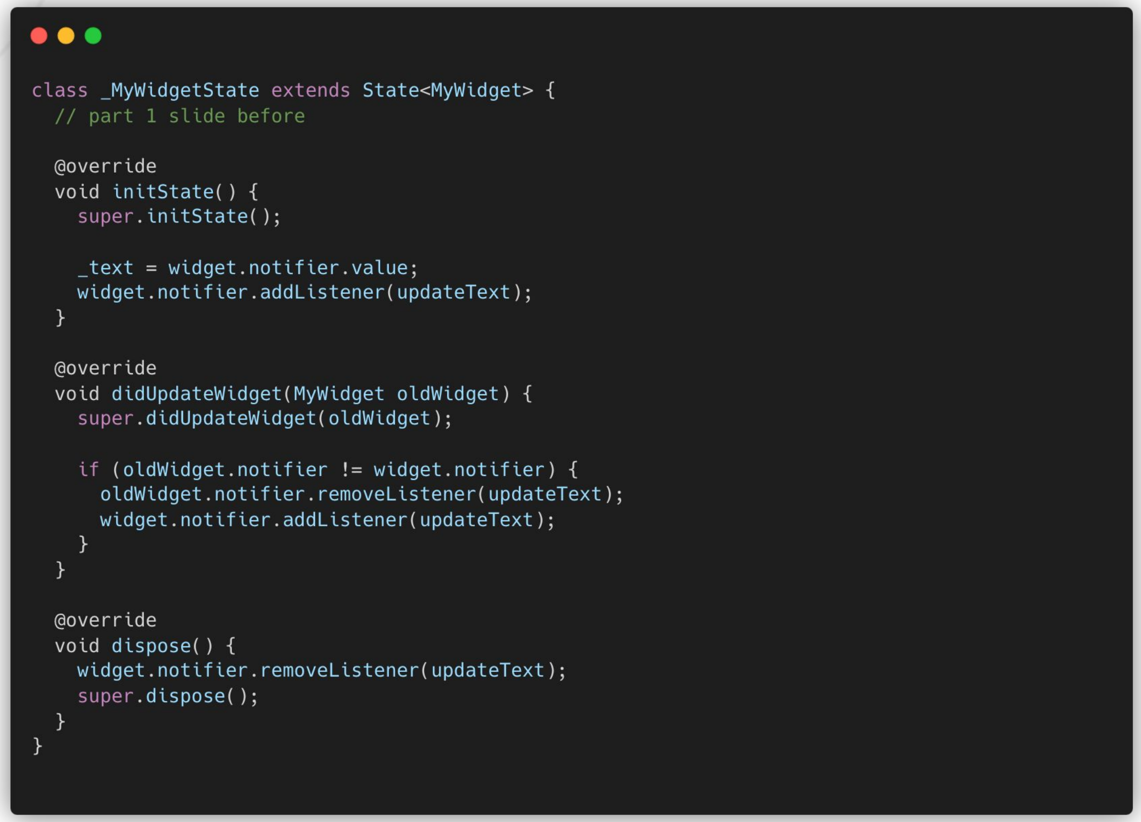
LeanCode

Subscribing to a Listenable



```
class MyWidget extends StatefulWidget {
  const MyWidget({super.key, required this.notifier});

  final ValueNotifier<String> notifier;

  @override
  State<MyWidget> createState() => _MyWidgetState();
}

class _MyWidgetState extends State<MyWidget> {
  String _text = '';

  void updateText() {
    setState(() {
      _text = widget.notifier.value;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Text(_text);
  }

  // to be continued
}
```

Subscribing to a Listenable



```dart
class _MyWidgetState extends State<MyWidget> {
  // part 1 slide before

  @override
  void initState() {
    super.initState();

    _text = widget.notifier.value;
    widget.notifier.addListener(updateText);
  }

  @override
  void didUpdateWidget(MyWidget oldWidget) {
    super.didUpdateWidget(oldWidget);

    if (oldWidget.notifier != widget.notifier) {
      oldWidget.notifier.removeListener(updateText);
      widget.notifier.addListener(updateText);
    }
  }

  @override
  void dispose() {
    widget.notifier.removeListener(updateText);
    super.dispose();
  }
}
```

We build digital products.

LeanCode

Listenables

Try putting a ValueNotifier or a custom ChangeNotifier in an InheritedWidget or a Provider!

Common listenables

A lot of builtin widgets use a listenable as an additional “controller” object when using the widget tree is uncomfortable or not performant enough. Examples:

- AnimationController – animations need that extra bit of performance
- ScrollController – all ScrollViews have one
- TextEditingController – handles more complex workflows involving text inputs

Extra sources

- Flutter architectural overview
    - On state and inherited widgets
    - On elements and render objects
- On widget reconciliation algorithm
- flutter_hooks | Flutter Package – alternative solution for managing state built on top of StatefulWidget