Introduction to declarative UI

Contents

This introduction describes the conceptual difference between the declarative style used by Flutter, and the imperative style used by many other UI frameworks.

Why a declarative UI?

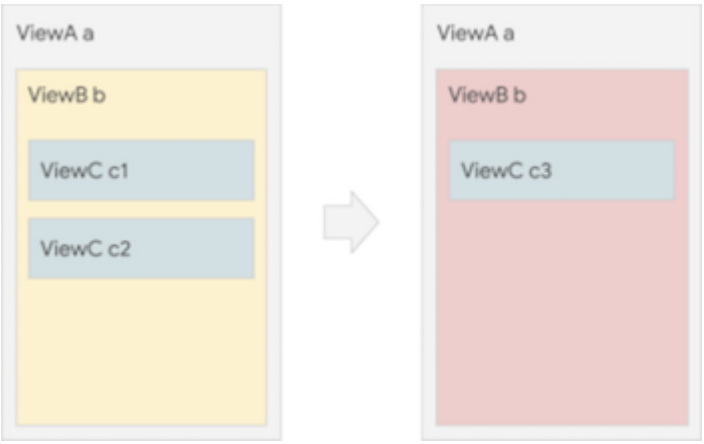
Frameworks from Win32 to web to Android and iOS typically use an imperative style of UI programming. This might be the style you're most familiar with—where you manually construct a full-functioned UI entity, such as a UIView or equivalent, and later mutate it using methods and setters when the UI changes.

In order to lighten the burden on developers from having to program how to transition between various UI states, Flutter, by contrast, lets the developer describe the current UI state and leaves the transitioning to the framework.

This, however, requires a slight shift in thinking for how to manipulate UI.

How to change UI in a declarative framework

Consider a simplified example below:



In the imperative style, you would typically go to ViewB's owner and retrieve the instance `b` using selectors or with `findViewById` or similar, and invoke mutations on it (and implicitly invalidate it). For example:

```
// Imperative style
b.setColor(red)
b.clearChildren()
ViewC c3 = new ViewC(...)
b.add(c3)
```

content_copy

You might also need to replicate this configuration in the constructor of ViewB since the source of truth for the UI might outlive instance `b` itself.

In the declarative style, view configurations (such as Flutter's Widgets) are immutable and are only lightweight "blueprints". To change the UI, a widget triggers a rebuild on itself (most commonly by calling `setState()` on StatefulWidgets in Flutter) and constructs a new Widget subtree.

```
// Declarative style
return ViewB(
  color: red,
  child: ViewC(...),
)
```

content_copy

Here, rather than mutating an old instance `b` when the UI changes, Flutter constructs new Widget instances. The framework manages many of the responsibilities of a traditional UI object (such as maintaining the state of the

layout) behind the scenes with RenderObjects. RenderObjects persist between frames and Flutter's lightweight Widgets tell the framework to mutate the RenderObjects between states. The Flutter framework handles the rest.