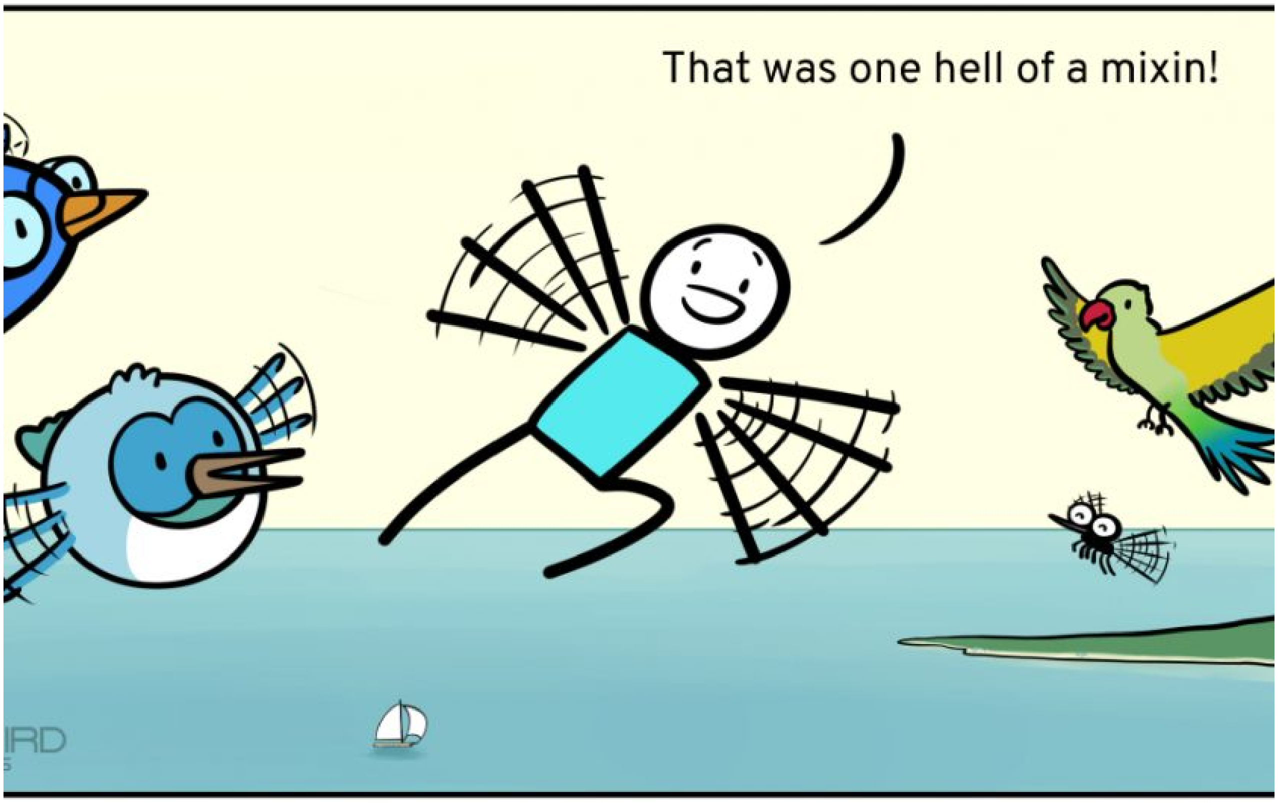
Use Dart Mixins More Often! Here is Why…

quickbirdstudios.com/blog/flutter-dart-mixins/

Marvin März

October 4, 2021



Mixins are a super-powerful feature in Dart and if you're developing apps with Flutter, you should use them more often! 💙 In this article, we'll show you why – and how.

What's a Mixin?

Mixins are awesome because they solve a basic problem that comes with inheritance. They are an elegant way to reuse code from different classes that don't quite fit into the same class hierarchy.

> A mixin is a class whose methods and properties can be used by other classes – **without subclassing**.

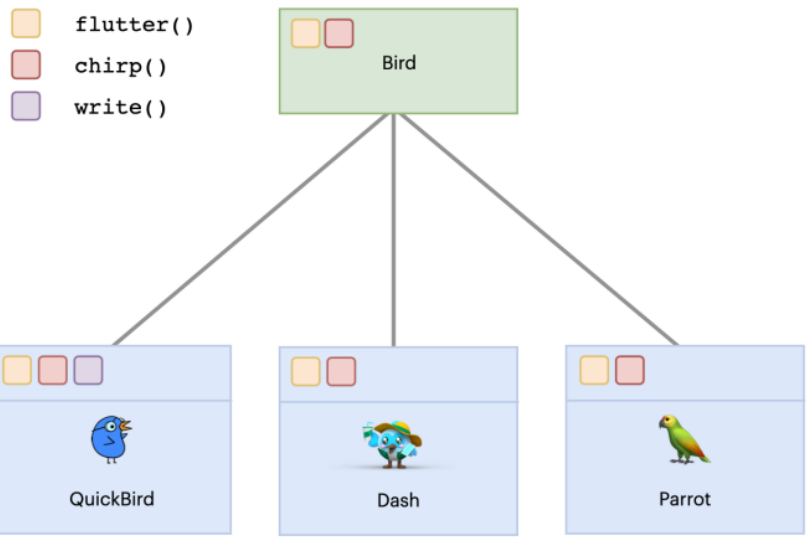
It's a reusable chunk of code that can be "plugged in" to any class that needs this functionality. Mixins are supported by Dart, and Flutter uses them in many places. As a Flutter developer, you are probably familiar with the `SingleTickerProviderStateMixin` that you need to use when creating animations in Flutter. We'll get to that later in this article. But first, let's take a closer look at the problem that mixins solve!

Why Mixins?

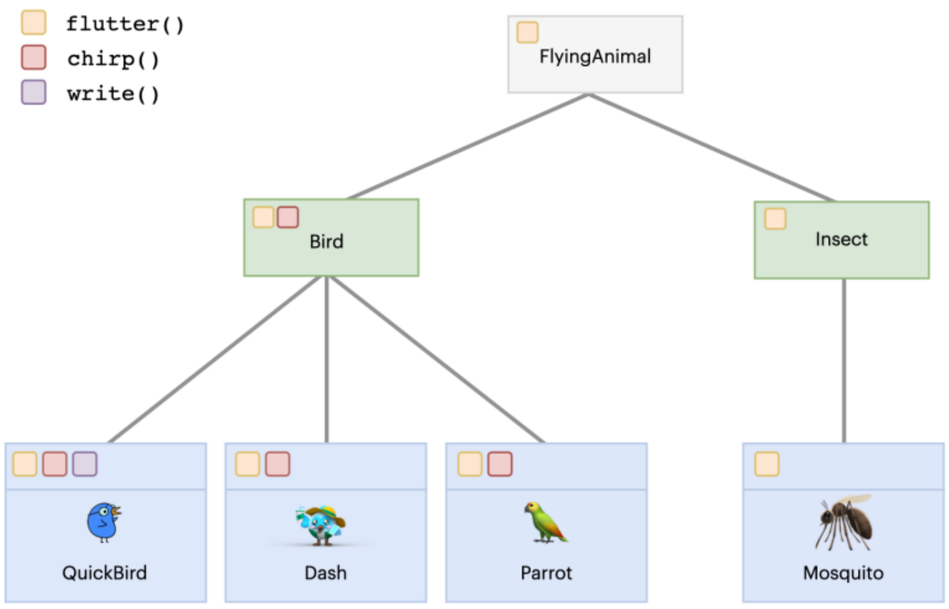
Do you know what Flutter's mascot *Dash* and our team's mascot *QuickBird* have in common? They both love to flutter around the world! And sometimes, their common friend *Parrot* joins them and they all flutter together. How would we model this in an object-oriented programming language?

Pure Subclassing

Obviously, all of them are birds. They all have the same abilities: They can flutter and chirp. So it's reasonable to have all three birds inherit from a common superclass `Bird` . Only our QuickBird is a little special: It can also write awesome blog posts. 😉 In a diagram, our class hierarchy would look like this:

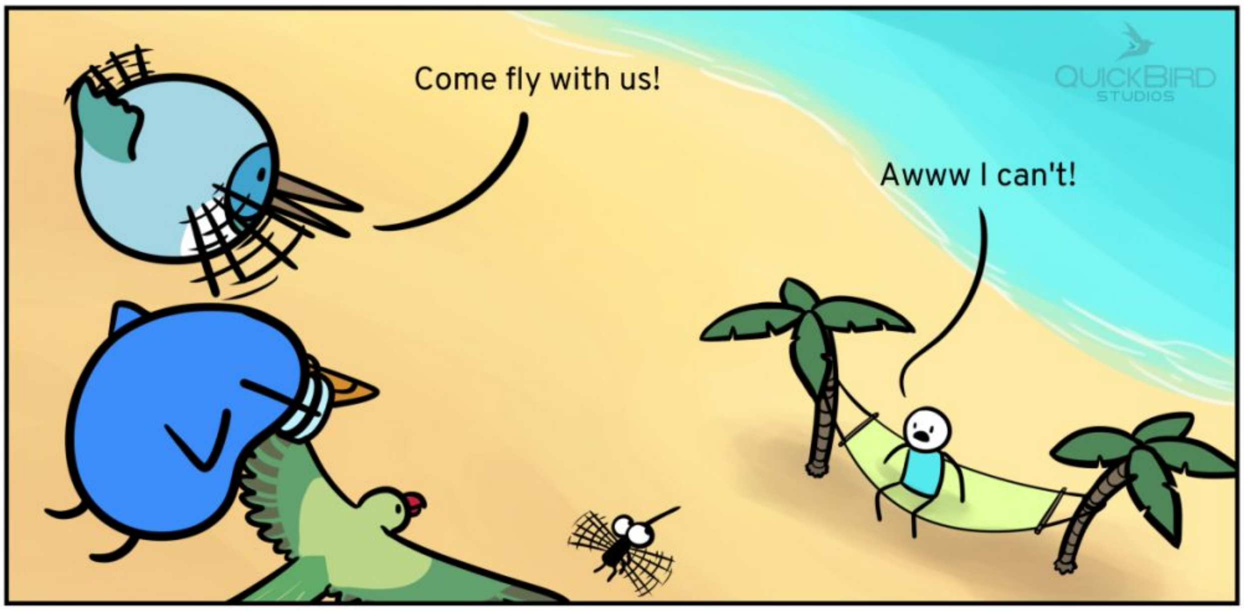


After some time flying around, our three birds make a short stop in Mexico where they make a new friend: Mosquito. Mosquito is also a master in fluttering his wings. Dash, QuickBird, and Parrot want to welcome him into their `Bird` group. But it turns out he's not a bird after all! 😯 He can't even chirp! Thus, he can't really join the bird's class, so they decide to create an additional superclass `FlyingAnimal` . Let's add this new superclass and `Mosquito` to our class diagram:

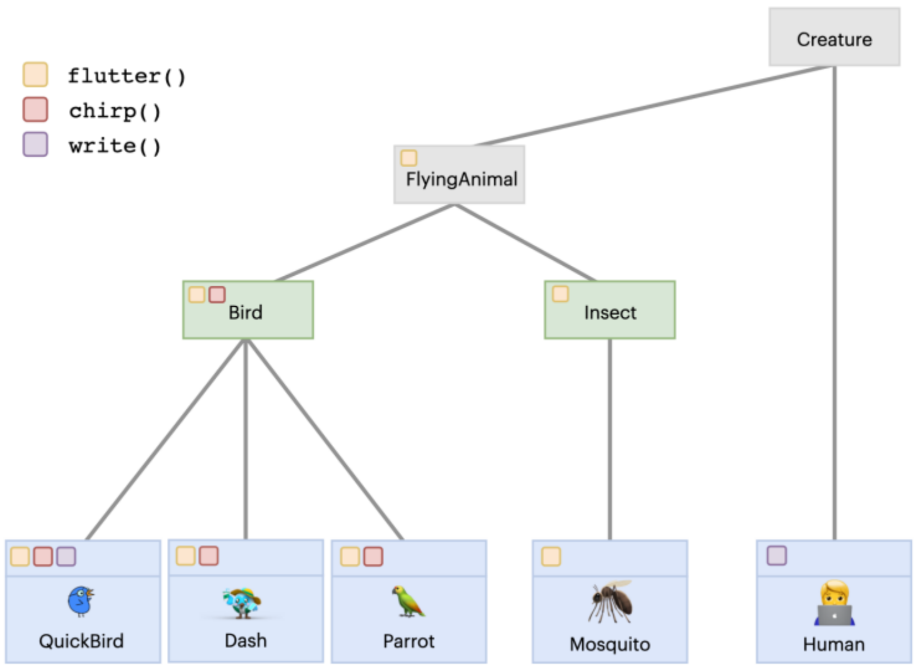


We also create a subclass `Insect` as our group of flying animals might encounter others who are like Mosquito. The `FlyingAnimal` class provides the common ability to flutter. The `Bird` subclass adds the ability to chirp. With that new class hierarchy, the four animals continue their journey.

The Problem



They touch down for another stop in the Bahamas. At the beach, they meet a guy who catches QuickBird's attention: He's definitely not a flying animal as he doesn't even have wings. But he's enthusiastically hitting his laptop's keyboard writing a blog post, so he seems to be able to `write` , just like our QuickBird. Let's include the human in our class diagram:



Now we have a problem: When implementing this in code, we would want to reuse the functions `flutter()` , `chirp()` and `write()` in all classes that offer the respective functionality. In object-oriented languages, code-reuse is done through subclassing. But we cannot create a common superclass for `Bird` , `Insect` , and `Human` that provides all the methods required by its subclasses.

- If we add the `write()` function to a common superclass `Creature` , all birds and insects would inherit this ability. But only `QuickBird` and `Human` should have this ability!
- If we create a new superclass `BlogWriter` and only have `QuickBird` and `Human` inherit from it, `QuickBird` would lose its `Bird` abilities to flutter and chirp.
- If we don't create a common superclass at all, we would have to implement the `write()` function in *both* classes, `QuickBird` and `Human` , leading to code duplication and thus, a violation of the *DRY* principle (Don't repeat yourself).

As you might have noticed, the structure is also getting more and more complicated, making it harder to understand and maintain in the future. To solve this problem, we need a way to reuse the code in multiple class hierarchies. The first thing that comes to mind is to allow classes to inherit from multiple superclasses, but multiple inheritance comes with the diamond problem and it's not supported in Dart. Instead, we use mixins!

Mixins in Practice

With Dart mixins, we can reuse our code without any restrictions in multiple class hierarchies. Let's try to remodel our class diagram with mixins! First, we create a `FlutterMixin` that provides the ability to flutter. We inject it into our `Bird` and `Mosquito` class. Next, we define a `ChirpMixin` that we only inject to the `Bird` class and a `WriteMixin` that we inject to the `QuickBird` and `Human` classes. (We only show the code for the first mixin to illustrate the syntax.)

Syntax

Mixins can be defined with the `mixin` keyword or via a normal class declaration.

```
mixin FlutterMixin on Creature {
  int height = 0;

  void flutter() {
    height++;
  }
}
```

As you can see in this example, both methods and properties can be defined in a mixin. The `on` keyword describes which inheritance structure the mixin can be used on. In this case, the `FlutterMixin` can only be used on `Creatures` . You don't have to restrict a mixin to a certain class, but it's often useful to keep your future self or another developer from mixing in functionality to classes that absolutely have no business dealing with this functionality. (For example, a mixin to edit and publish blog posts should be restricted to a `BlogAuthor` class, so that no one can accidentally add the `EditMixin` to a `BlogReader` .)

To add the mixins' functionality to the `Bird` and `QuickBird` classes, we use the `with` keyword:

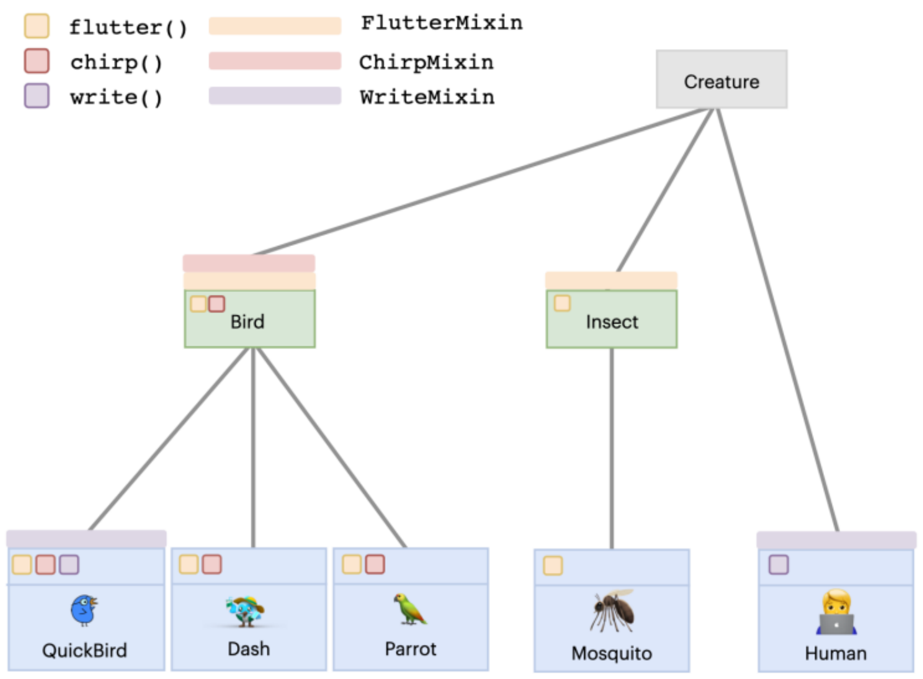
```
class Bird extends Creature with FlutterMixin, ChirpMixin {}

class QuickBird extends Bird with WriteMixin {}
```

Now, we can use the functions that are defined inside the mixins. Example:

```
void main(String[] args) {
  final quickBird = QuickBird();

  quickBird.flutter(); // inherited from Bird which includes the FlutterMixin
  quickBird.chirp();   // inherited from Bird which includes the ChirpMixin
  quickBird.write();   // directly mixed-in via WriteMixin
}
```

How Mixins Fit Into The Inheritance Hierarchy

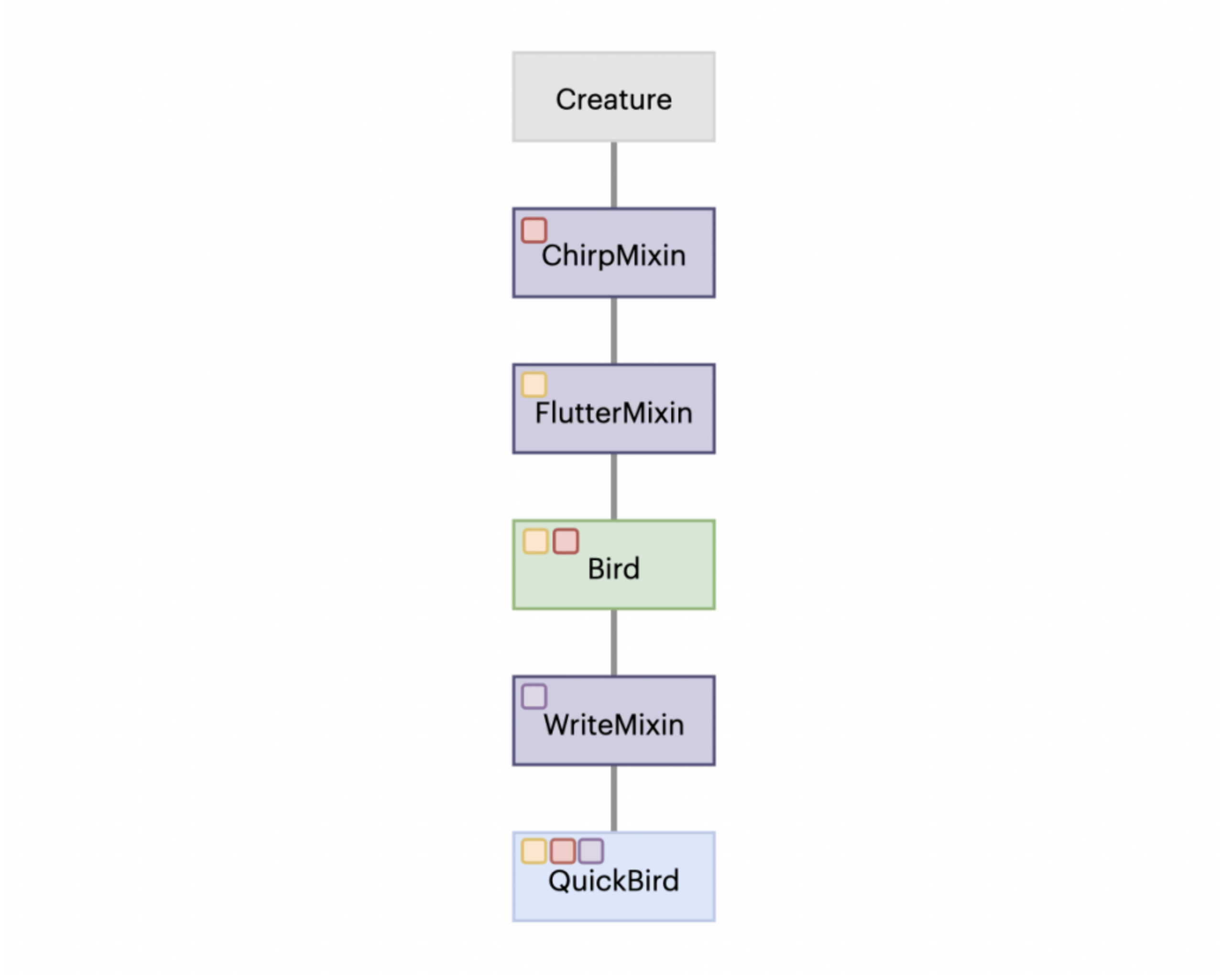
Let's take a moment to reflect how we modified our inheritance structure with the mixins:



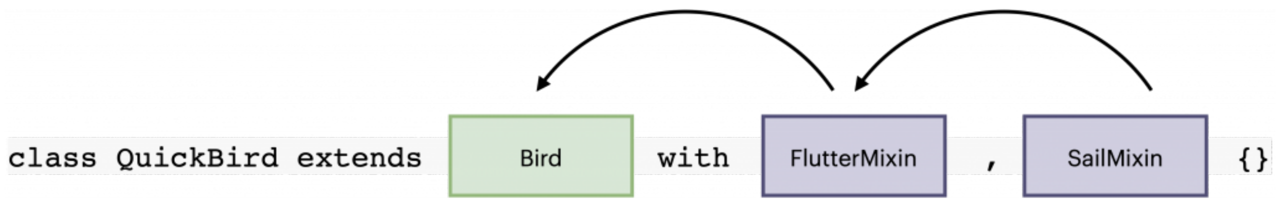
Now that we've mixed our `QuickBird` class with so many mixins, what's its actual type? How are the mixins inserted in the inheritance hierarchy? Let's check that in code!

```
void main(String[] args) {
  final quickBird = QuickBird();

  if(quickBird is Creature)
    print('I am a Creature');      // prints: "I am a Creature"

  if(quickBird is Bird)
    print('I am a Bird');          // prints: "I am a Bird"

  if(quickBird is FlutterMixin)
    print('I am a FlutterMixin');  // prints: "I am a FlutterMixin"
}
```

As we can see from the output, mixins are polymorphic, just like classes. A class assumes the type of its ancestor classes and also the type of its injected mixins. If we take a look at the inheritance structure at runtime, we can see why it works this way:



Every mixin creates a new class/interface at runtime that is inherited by the next class in the hierarchy. The order of the mixins matters as it specifies the order in which the mixins and classes inherit from each other. As a core rule, the inheritance hierarchy always reads from right to left in the definition of a class:



The Order Matters

As mixins are integrated into a class hierarchy at runtime, they can also override methods. Here's an example:

```
abstract class Bird {
  void move() {
    print('Bird ▶ The bird is moving');
  }
}

mixin FlutterMixin on Bird {
  @override
  void move() {
    super.move();
    print('FlutterMixin ▶ The bird is fluttering through the air');
  }
}

mixin SailMixin on Bird {
  @override
  void move() {
    super.move();
    print('SailMixin ▶ The bird is sailing in the wind');
  }
}
```

If we now define `QuickBird` as a subclass of `Bird` and add the two mixins in the following order

```
class QuickBird extends Bird with FlutterMixin, SailMixin {}
```

the code

```
QuickBird().move();
```

will call the `move()` functions of all the superclasses in the hierarchy and print the following output:

```
Bird ▶ The bird is moving
FlutterMixin ▶ The bird is fluttering through the air
SailMixin ▶ The bird is sailing in the wind
```

When we call the `move` function, the `SailMixin` 's implementation is called first as it's the first mixin from the right. As it calls `super()` first, the `move()` implementation of the next mixin from the right (the `FlutterMixin`) is called before printing the output. Again, it first calls `super()` super which executes the `move()` method on `Bird` . So the `Bird` 's message is printed first, then the `FlutterMixin` 's message and finally the `SailMixin` 's.

If we change the order of the `FlutterMixin` and the `SailMixin` ,

```
class QuickBird extends Bird with SailMixin, FlutterMixin {}
```

the output changes as follows:

```
Bird ▶ The bird is moving
SailMixin ▶ The bird is sailing in the wind
FlutterMixin ▶ The bird is fluttering through the sky
```

As the `FlutterMixin` is now the first mixin from the right, it’s the lowest in the class hierarchy, and its `move()` method is executed first. The `super()` call executes the `SailMixin` ‘s `move()` method and ultimately the `Bird` ‘s `move()` method. So the `Bird` ‘s message is still printed first as it’s the top-most class in the hierarchy, but the order of the `SailMixin` message and `FlutterMixin` message changes.

If we wanna go nuts, we can even include the same mixin multiple times in a class:

```
class QuickBird extends Bird with FlutterMixin, FlutterMixin {}
```

The output will be:

```
Bird ▶ The bird is moving
FlutterMixin ▶ The bird is fluttering through the sky
FlutterMixin ▶ The bird is fluttering through the sky
```

In this case, the order of the mixin specification doesn’t matter anymore as both mixins are the same. But is there any real-world use-case for this? No idea. 🤷‍♂️ If you find one, please let us know on Twitter! 😉

When to Use Mixins

Now we saw that mixins can be a really useful tool to share properties and behaviors across multiple classes, let’s have a look at the mixins `TickerProviderStateMixin` and `SingleTickerProviderStateMixin` . Both are used for explicit animations in Flutter. By specifying one of these mixins in our widget definition, we tell Flutter that the widget is animated and that we want to get notified for frame updates so that our `AnimationController` can generate a new value and we can redraw the widget for the next animation step.

As the name suggests, the `SingleTickerProviderStateMixin` is used when you only have a single animation controller and the `TickerProviderStateMixin` is used when you have multiple animation controllers.

```
mixin SingleTickerProviderStateMixin<T extends StatefulWidget> on State<T>
    implements TickerProvider {

    Ticker? _ticker;

    @override
    Ticker createTicker(TickerCallback onTick) {
        // ...
        _ticker = Ticker(
            onTick,
            debugLabel:
                kDebugMode ? 'created by ${describeIdentity(this)}' : null
        );
        return _ticker!;
    }

    @override
    void didChangeDependencies() {
        if (_ticker != null)
            _ticker!.muted = !TickerMode.of(context);
        super.didChangeDependencies();
    }

    // ...
}
```

The code example is shortened to just get a quick overview. As you can see, the `SingleTickerProviderStateMixin` makes use of the Flutter `State` -lifecycle methods. If we would now take a deeper look, we would also see that the Ticker that is used in the `SingleTickerProviderStateMixin` inherits the `StatelessWidget` to trigger the redrawing.

One other area where mixins are heavily used in Dart and Flutter is JSON (de)serialization. Writing methods by hand to serialize and deserialize a data class is often error-prone and a repeating task. That's one of the reasons why many developers resort to code generation tools to create that functionality. With mixins, the generated code can be easily included in the original data class.

There are many more examples where mixins are super useful, for example:

- Service locators
- i18n (Internationalization/Localization)
- Logging
- etc.

Where would mixins benefit your project?

Downsides & Dangers

This all sounds very promising, like a secret tool to rule them all. But what are the downsides? One main issue (as with any programming technique) is heavy over-use: If you start to rely on mixins only without the “good old” inheritance, your classes will soon look this:

```
class Creature with FlutterMixin, SailMixin, ReadMixin, EatMixin,....
```

In other words: Mixins make it very tempting to include all sorts of functionality in a single class. But when a class can do too many things, it violates the single-responsibility principle and it’s hard to keep an overview and an intuitive understanding of the class’ purpose. Furthermore, it can become pretty confusing which method was inherited from which mixin and it might not be clear to other developers in which order the code is executed.

Mixins don’t replace inheritance. They *extend* it. They patch some of its flaws. Use them wisely and always consider subclassing as an alternative. But in our experience, mixins are still heavily *under*used and underappreciated. So if you find a good application, don’t be afraid to use them!

Conclusion

Do we really want to end like this? Leaving our human friend behind with his laptop in the Bahamas? Of course not! Now that we have our `FlutterMixin` ready, we can simply add that to the `Human` class as well and give him the ability to fly. And the sky is **not** the limit when working with mixins. 🚀

In this article, you’ve learned what mixins are and what problems they solve. We’ve shown you how mixins integrate into the class hierarchy and a real-life example that you typically use with animations. At QuickBird Studios, we believe that mixins are a powerful feature in Dart and should be used more often in production apps. If you have any questions, comments, or feedback, feel free to contact us on Twitter.

By the way: Another great feature for adding functionality to classes are Dart Extension Methods and we’ve covered this topic in another article. Thanks for reading! 😀

Do you search for a job as a Flutter Developer?
Do you want to work with people that care about good software engineering?
Join our team in Munich

🐣

Get notified when our next article is born!
(no spam, just one app-development-related article per month)