So we're on to our final pillar and that's polymorphism or changing shapes.

How does this work?

Let's say that we're abstracting our program and we're creating a separate module for separate roles.

And we're inheriting a lot of the behavior from parent classes.

But when we're doing a lot of inheritance, we know that we get basically a copy of what the parent class is able to do.

So our electric car can drive and our electric car has the same properties. But we don't really want to end up with just a whole bunch of clones in our program. That isn't great right?

We want to be able to define customer behavior as well.

So in the case of our chef and our pastry chef,well, there's probably similarities between those two roles.

They can probably both bake something in the oven and they can probably both make some sauce in a saucepan.

But they might go about those behaviors a little bit differently.

So maybe the pastry chef has to be a little bit more delicate with the things that he's working with,and the chef maybe,well she has to do things a little bit quicker because it's complete mayhem in the kitchen.

But the point is that when you are inheriting from a parent class, there are probably things that will save yourself time that you don't have to recreate and you can simply inherit.

But some of those behaviors that you inheriting, some of the methods, you might want to customize it for your own needs.

There's two ways that you can do this.

Continuing our car analogy, we might have simply a LevitatingCar, and this LevitatingCar is going to inherit from our Car class.

So we're going to write extends the Car class, and this is because a levitating car probably also has 5 seats, then it probably also looks pretty much the same as a car. It might share the same properties but a levitating car doesn't have wheels, at least in my mind.

And instead, if we were to implement the drive functionality of a levitating car, it would probably be a little bit different. Because we're already inheriting from the Car class that has a drive method, if we want to use the same method, we have to override it and to do that we use the @ sign to specify that there's a very important word coming up, and the word is override. And then on the next line, we can create our own version of drive so we can write void drive and it looks exactly the same as before. But now, it's our own version of the drive method and our levitating car instead of moving wheels,it's going to simply glide forwards, very classy.

So now when we create our levitating car,and we call upon our levitating car, our MagLev to drive, then you can see that it should now glide forwards instead of using its parents implementation of drive. So even though I inherit from my parents a number of traits and a number of behaviors, there's also things that I do differently from them or I'd like to think I do differently.

For example the way I use a computer, I might get to use a computer to program it whereas my parents might just use it to go on Facebook.

But the point is we are now able to override a parent behavior and provide our own custom version of it but we can actually go even further than that.

We can actually take some of the good parts from our parent class, but simply just add to it.

Let me show you what I mean.

Let's build a new class and this is going to be a self driving car.

And of course it's a car.

So it's going to extend the car class.

It's going to have a drive method and it's going to have 5 seats. So, easy. That was done.

But in addition to being able to drive and make the wheels turn,

I also want my self-driving car to be able to steer the car as well. Instead of simply just getting the wheels to turn,I'm going to get my self-driving car to drive me towards a destination.

So I'm going to create a new property called destination. And this destination is going to be set when I create a new object from the self-driving car.

So I'm going to define a custom constructor, so self-driving car, when we create it,we have to provide a destination. And we'll call that userSetDestination.

And then once we create our self-driving car, we'll set the destination property to equal the user set destination that was provided when we created our new object.

Now inside this self-driving car, when we decide to drive it, we kind of want to inherit the functionality of driving which is wheels moving,and as a consequence the car moving.

But we want to add to it as well. We can again override our parent method code drive.

So it has to look exactly the same.

It's also void drive empty parentheses.

And then we're going to trigger the parent behavior by calling super, which stands for the superclass or the parent.drive.

This is going to carry out the behavior of the superclass' drive method.

But then afterwards, we're going to do our own thing. We're going to say that the steering is going to steer us towards the destination.

Here's how it would look when we actually call this method in our main. So we could create our new self driving car, we'll call it myWaymo. And it's gonna be equal to a new self driving car. And when I create myWaymo as a new self-driving car, I have to provide a destination.

So let's drive it towards 1 Hacker way.

And now when I say myWaymo.drive calling that drive method,then when I hit run, you'll see two things happen.

One is that it'll carry out the parents version of drive, which is simply to print 'wheels turn', but it'll also do its own thing which is steering towards the destination.

And this all comes from just calling that one drive method.

This is polymorphism in action.

We're able to inherit from our parents, but we're able to also improve on it.

We're able to change it be a little bit different by overriding their methods. And if we come back to our Flutter code, you can see that we're doing that every time we're calling the build method.

So a stateless widget already has an implementation for the build method, but in our code, we override it and we provide what it is that we want to do instead.

And this way, even though we're inheriting, we're able to be a little bit different and customize what our stateless or our stateful widgets do.

So getting the best parts, but then also doing custom things using these inherited classes.