Now in this lesson, we're going to look at mostly how to extract and refactor widgets as separate classes.

And the reason why we're doing this is because as our apps get more and more complex, it makes sense to take parts of it that are repeated in the widget tree and remove them so that we can consolidate them into a single widget.

And then we can simply use that widget wherever we need the same appearance or the same functionality.

But before we can do this, we first have to actually have something on screen.

We have to design the user interface. And the first step of creating this is actually to create the skeleton.

So the individual cards that you see on screen and they sort of have this slightly lighter color just so that we'll be able to group them into individual components or areas. And the user will be able to see the different parts that come together to calculate their BMI.

So before we actually add any of the content or the text, we're first going to create the cards that will be the background for each of those areas.

So we're aiming for something that looks a bit like this.

Let's start off by building a single one of these cards first. I'm going to go ahead and delete everything that's in the body of my scaffold because I don't actually need any of the text.

And instead, what I want to have here is a container.

Now my container is going to have a background color and it's going to be the color that we see in this design right here.

And I've already extracted the hex code so if you want to simply add it, it's just a color that is 0xFF, and then it's 1D1E33.

And now we should have a container on our screen which has that slightly lighter sort of purply color.

Now I don't want it to cover the entire screen, so I want to be able to give it maybe a height.

Let's give it a height of say 200 and a width of 170.

Now we should have a small square that's stuck to the top left. But I actually want it to have a little bit of margin away from everything so that they're separated from each other like this.

So I'm going to go ahead and add a margin to my container as well.

And I've worked out that having a margin of about 15 from all sides looks pretty good.

So that's what we've got now.

Now the next step, you'll notice is that here the edges of my container is pretty sharp.

I would love it if it looked a little bit more rounded like this.

It looks a little bit more friendly, and it looks quite nice. To my eye at least. In order to achieve this,we need to be able to create a border radius on our container.

And this is very similar to the border radius that you would see in sebsites if you've done any Web

Development.

And in order to implement it, we add something called a decoration property.

And the decoration property expects something called a boxDecoration. And the box decoration has properties such as the color, the image border and also border radius which is what we're interested in.

And this allows us to change our boxes to have a rounded corner.

So I'm going to inset that border radius and I'm going to set it as a circular border radius, with a radius of 10 pixels.

Let's hit save, and you'll see that our app will actually crash.

And the reason is because when we use the color property for our container, it's actually just a shorthand for the color property of our box decoration.

Notice that our box decoration also has a color property and that is actually the proper way of setting the color of a container's background.

But because it's so frequently used, you don't want to have to write decoration, box decoration,color etc. all the time. The Flutter team made it simple for us to add a shorthand color on our container by using this property.

But if we have a box decoration, then we have to move our color into the box decoration.

And this way our code will no longer crash and we will have that light purple color background with rounded corners.

Now if we were to go around our app and add a hardcoded size for each of these cards, then it won't really adapt to the size of the screen.

So if the screen was changed to, for example, landscape or if it was on a smaller screen size, it will overflow and it will have those yellow bars where widgets are off the screen.

So that's not what we want. The solution to that is to use an expanded widget, which we've seen previously.

Let's go ahead and delete our height and width. And the challenge is for you to create this layout using the container that we've created just now, but laid out like this on the screen to fit the size that is available.

Pause the video and see if you can achieve this look.

All right.

So first thing to notice is that everything is pretty much stacked in a column. And inside the first column,we've got a row of two cards and the same applies to the last column.

So let's go ahead and cut out our container from the body and put it down here. And then go ahead and add a column which has some children. And our column is going to contain three expanded widgets, each with their own children.

So I'm simply going to copy this and paste it three times. So each of these expanded widgets will be a single row in our column here, one, two and three.

Now the first expanded widget is going to have a child that is a row widget. And it's also going to have some children and it's going to have two expanded widgets.

So now we have our column with three rows which are each expanded widgets.

We've added a row widget in the first row so they can have two expanded widgets and we're going to replicate that in the last row.

So I'm simply going to copy this entire expanded widget and I'm going to replace the very last row with that.

Now at the moment if you had save and you look at the screen, nothing really happens. And also we've got our code just lying around here,so it's not going to do much. But if we go ahead and we copy our container and add it as the child to every one of our expanded widgets, then you will see that all of our hard work laying out the screen using the expanded widgets, the columns and the rows now actually gets implemented on the screen and we actually see that design that we saw earlier on that we're trying to mimic. If this is at all confusing,be sure to head back to the video where we talked about columns, rows expanded widgets, flex properties and all of that so that this is really clear in your mind.

Now be sure to add commas at the end of all of the widgets so that when you hit save, Dart will reformat your code properly for you so that everything looks neat and tidy. So this code works and it looks pretty much like what we want, but it's very very repetitive,right?

We actually copied and pasted our container into six places, even though the container that we're putting in is completely identical.

This should really set off your programmer spidery senses and it's a good time to actually refactor our code to be able to reduce the amount of repetition, so that we keep our code dry and you do not repeat yourself. In order to extract these containers,there's actually a really neat trick that we have access to from Flutter.

If you select the widget that is repeated, which in our case is the container, and you go into the Flutter outline tab, you'll see it selected in the widget tree here.

Now if you right click here, you can select extract widget.

And what this will do is it'll ask you for a name for the new widget, which you're going to call reusable Card and we're going to refactor this into a new class.

So this is now a new reusable card, and Flutter has actually created all of this code for us which simply extracts the widget into a separate class which extends a stateless widget and we can now use our reusable card anywhere where we have that container with the margin and the decoration and the color etc. Then that's why we kind of called it a reusable card right?

So when Flutter generates this code automatically, you'll see it creates a 'new' keyword.

And we haven't really seen that so far.

And the reason is because as of Dart 2.0 the latest version of Dart, this was actually deprecated.

So when we create new objects from a class using the latest version of Dart, we actually don't need this new keyword at all. We can simply just write the of name the class and then add the parentheses.

We can now go and delete all of the places where we have that same container and simply create a new reusable card instead. Now you can see that our code is vastly simplified and we've reduced the code by something like 30 lines.

And now it looks a lot clearer as well.

So if we had a problem with our reusable card, all we need to do is to find the place where was created and actually tackle it here rather than having to fix it in six different places. Now in our stateless widget, you can see that we have this constructor up here.

And if I change the way that it is formatted, it kind of starts looking familiar to what we learnt in the Quizzler module when we first learned about Dart classes and objects.

In this case it keeps on talking about this thing called key.

Now what exactly is a key? The key class is something that you use when you need to keep track of the state of widgets.

And this is especially useful when your widgets are moving around in the widget tree, when they're changing their position in the widget tree.

This usually happens when your widgets are physically moving on the screen such as in an animation or when you see things scrolling on screen.

But in most other cases, we don't actually need it.

So we're going to go ahead and delete this constructor that got generated automatically and we're going to create our very own constructor.

Now if you want to learn more about when to actually use these keys and what they're used for, then head over to this link where Emily from the Flutter team actually explains to you in great detail how to use it and what it's used for and in which cases it will help your widgets be able to keep track of its state.

A link to this in the course resources as well.

But in our case, we don't need to do anything with the key. But if we want to be able to specify a custom value for one of the properties that we use in our reusable card, then we will need to have a constructor.

The property that we might want to change about our reusable card is actually the color of the card because we might not always want to have the same color.

And if you take a look at the final result, you can see that when we click on the male or female selector,then it actually changes the background color of the card to show that it's actually selected versus when it's inactive.

This is something that we will need to change about our reusable cards. So it makes sense to have it as a property on the reusable card which we can pass over when we create a new reusable card widget.

So we can go ahead and create a property and that property is going to be of the type color.

And I'm simply going to call it colour as well.

And because I'm from London, I will spell it this way but you can spell it whichever way you like. But it actually helps me to differentiate this color property that I've created from the color property that is from the container widget.

So I prefer to spell it like this.

And we're going to start it off with no value, so I'm simply going to end it with a semicolon. And then we're going to create the constructor for our reusable card. As a quick reminder of how you create constructors,we simply add the name of the class,we open up a set of parentheses and then inside here we add the property that we want to initialize, which in our case is this.colour,and then we finish off that statement.

Now if we want to be able to refer to the name of this property, then we can also add a set of curly braces around all the properties that we want to be named.

So now, that means when I go over here and I create my reusable card, I can actually add in the colour property and I can specify a custom color.

So for example, maybe this one.

And inside my reusable card instead of always using the same color whenever I create a new reusable card widget, I can use whichever colour was passed over when this widget was created.

So now, we're getting closer to creating custom widgets whenever we need custom functionality or custom look and feel as in this case. Now at the moment, this property could colour is actually an optional property.

So you can see up here, we created a reusable card here where we specified a color.

But over here, we actually just initialized a reusable card widget without specifying anything in the constructor.

And this is allowed because we're not getting any warnings over here or any errors.

But if you think about it, every card probably is going to need a background color right?

We can't have a null color for it.

So it makes sense to always require a color whenever we're creating a reusable card widget. In order to make certain properties required,what we can do is in the constructor we can add an annotation that is the required keyword. And then after a space, we have the property that is required.

So now if we scroll up, you'll be seeing these yellow highlights on all of these other reusable cards.

And it's telling us that the parameter colour for this reusable code is required.

So we have to put it in.

So let's go ahead and add the same colour to all of our reusable cards.

And now our warnings on the right will go away.

So now if we had save and we take a look at our app, then absolutely nothing has changed. But we've refacted our code to make it more synced, more readable and much easier to fix when something goes wrong.

And we've essentially created of very own custom widget which is a reusable card that we can use whenever we need that particular widget which is a rounded corner container with some margin and a background color which we specify when we create a new reusable card. And you can imagine that as we add more properties to our widget, we start building out a widget that's really similar to the default Flutter widgets. For example, our container has a margin property, a decoration property, a color property.

And these are all properties that are in that particular class right?

You can see that the container is a stateless widget and it has properties such as alignment, padding,decoration.

And we can specify all of these when we create a new container widget.

But in this case when we create a new reusable card widget, we're actually looking to our very own code to see how it should look and how it should behave.

For example we could have a card that has a simply, a blue color.

And now our widget updates to have that appearance, just as a floating action button would or a flat button would change when we change its property.

So it's pretty much the same, but it's defined by us.

Now the last thing I want to show you is here, we actually have a warning.

And when we hover over it, it tells us that this class is marked as immutable. But one or more of its instance fields are not final, namely the Reusable Card.color field that refers to this right here.

And what it's complaining about is that I haven't got the key word final in front of this field or property.

The reason why this is important and the reason why we're getting these warnings is because final makes this property immutable.

It means that when we create a new reusable card, we can set the color and we can't change it again.

Now why this is important and how this actually works,

I'm going to leave until the next lesson where we talk about final versus const, in the Dart programming language and how this affects the way that our widgets work and behave onscreen.

So for all of that and more, I'll see you on the next lesson.