In the last lesson, we've seen how we can create our classes and our objects using Dart.

But why do we actually need them anyways?

What purpose do they serve?

Well in order to understand this, we first have to understand how computers actually work. Behind the metal shell and the pretty keyboard and the nice screen,a computer is just composed of a large number of switches. And these switches can be in two states, on or off. And it's through switching on and off many different switches we end up telling the computer what it should do.

Now if you had to sit there and write all of these ones and zeros in order to program your computer,you'd be there for years and in fact this is how people used to code way way back.

But then we thought that this is a bit painful because in order to just represent the letter A,

I have to write 01000001. And that is very painful and very time consuming.

This is what we would say is machine code.

So we came up with all sorts of ways of making it easier to write programs that sound and look more like English so that we can actually interact with the computer without having to get down to the machine level and start writing ones and zeros.

In the early days, a lot of the programming languages were what we would call procedural.

They would go from top to bottom/ So do this, then this then this then this then this, until forever basically.

And this is kind of easy to understand. But as your code gets more and more complex, then it gets a bit hairy cause it's hard to know when something goes wrong,what exactly went wrong?

Because you have to follow the logic from top to bottom and you have to really untangle it and you can really shoot yourself in the foot quite easily.

So in a lot of places, complexity is always the enemy.

As humans we have pretty capable brains but we can only hold so much information in it at one time. The more complex a program gets, the more likely it is that it will crash and it will have problems and it won't perform very well.

So along came this concept of Object Oriented Programming, and a lot of people really loved the concept.

So languages that became really big such as Oracles, Java or Apple Swift or in fact Dart,which we're working with, are all Object Oriented Programming languages.

So what does that mean?

Well it means that we're working with objects to do everything that our app or our program needs to do.

And for those guys who are looking for a job as a software engineer, one of the most common questions you get asked is 'What are the four pillars of Object Oriented Programming?'

What is Object Oriented Programming all about? And the four pillars you'll very often hear people reel off are abstraction, encapsulation, inheritance and polymorphism.

Now it's a mouthful but very few people actually fully understand what these big words actually mean.

So I want to break it down and show you in the context of our app, why OOP is so great, and how does it apply to Dart.

One of the first pillars is abstraction.

How can we make something that's complex into smaller pieces to make it more abstract?

Now if you've taken my courses before, you know how much I love food.

So in order to understand abstraction, we're going to talk about, you guessed it, food.

One of my favorite restaurants in Japan are these really tiny kind of hole-in-the-wall places where there's only maybe four or five seats. But there's also just one guy who is the waiter, the chef, the cleaning staff and he just does everything from behind the bar counter. When you make an order, you do it on a vending machine and you give him the order. And then he'll cook you order he'll put it on the table for you because it's so easy to reach.

And then once you're done he'll take your bowl and he'll put it in the dishwasher.

Now this is really cool but there are some limitations. He can't have more than say 10 customers right?

Because it will be complete nightmare to try and do all of those jobs himself.

And that's kind of the downside of that.

If we have a large piece of code that is trying to do many many things, then it's kind of like just having one employee in your restaurant who's cooking, who's waiting on staff, who's taking payment at the cash register,who's also taking reservations and bookings. And firstly, good luck finding somebody with the CV that can do all of those things.

And if you hire them, you're going to completely destroy them because one person shouldn't have to do so many things.

And if they did, then you end up having a limitation in how many customers you can serve or how big your restaurant can be or how complex your app or program can be.

So as your program gets more complex and more interesting, then what you want to do is you want a modular rise.

You want to make it more abstract and separate out different jobs. So you'll have the waiter who actually brings the food to the customers.

You'll have the pastry chef who makes the cakes. You'll have the sous-chef and the chef who does the rest of the cooking.

And now, what it means is that everybody has their own jobs.

They all know what they need to do and they don't get into each other's way and they don't have to all do too much at once.

And now, you can expand your restaurant. You can serve more people and you can do more interesting things with more abstraction.

So if you were creating a program, instead of creating a single large component which is capable of everything it's all knowing and all doing, that kind of tends to end up in what we call spaghetti code where everything is intermingled with each other.

And it's a mess to try and debug when something goes wrong or when something doesn't do what you expect it to.

Instead it's much better to split out functionality into different components: the chef component, the waiter component, the receptionist and the pastry chef.

And this is what abstraction is all about, being able to build more complex systems by creating smaller pieces that have a defined job or defined role so that you can actually manage the complexity.

What does that actually mean when we try to apply it to our code?

Well at the moment, if you look inside our main.dart, you can see that it's getting pretty long right?

And it's trying to do a lot of things.

For example it has the questions of our question bank,it tries to track which question we're on at the moment, when we need to get a new question we have to look into our question bank.

And it also does things such as putting on the layout for our app, and just you know being able to implement the buttons and all of that functionality. And it's getting a bit messy in here.

If our app got any more complex, then you might start getting a bit confused about exactly what does what in this main.dart file.

So let's abstract it. Let's make it more modular. In order to do that,we're going to go into our lib folder. We're going to right click and we're going to create a new Dart file.

And we gonna call this Dart file quiz\_brain because this is going to be the brains of our quiz.

Now when I asked you if you want to add it to Git, go ahead and click

Yes.

And now we can create on new quiz\_brain.And of course it's going to be a class.

So we're going to create a class called Quiz brain. So remember classes have to start out with capitals.

And in this class, we're going to define all of the things that a quiz should be and should be able to do.

First of all, our quiz needs to have some questions.

Let's head into our main.dart,and instead of having our question bank mixed in with all of the rest of the functionality, let's go ahead and cut it out of here and put it into where it should be inside our quiz\_brain.

The first thing you'll see is we have some errors here.

It doesn't know what this question thing is.

Well it's because we haven't actually yet imported our question.dart file which is where that class

Question is defined now.

While we're on the topic of our question file, the first thing to note is that at the moment, this class is doing a very very simple job.

It defines the structure of every question namely that it should have a question and it should have an answer. When we're creating the constructor for our question class,we have named parameters here.

So in order to create a new question object, we have to write question newQuestion let's call it equals a new question.

And we have to specify that q and also the a.

Now this is a bit wordy. Given that every question just has a text which is a piece of string and an answer which is a boolean,

it's pretty straightforward to be able to see that if we create a question that has a string.

This is the text and then it has a boolean,this is the answer.

It's not completely necessary to have the parameter names q and a in there, but this is your judgment as well.

So if you prefer it the other way, keep it the other way.

But I find that this is much easier to read once we start building up a large bank of questions.

Speaking of a large bank of questions, it's not much of a quiz app with just three questions right?

So if you open up your readme file, you'll notice that we've already included some pre typed questions.

There's 13 of them.

And if you copy everything between the back ticks, all of those questions, you can paste it and replace what you have in your question bank at the moment. And they're already formatted so that we create a new question object with a question text and an answer. Now all we have to do is to go into our main

.dart file and fix the errors that we have here. Because we deleted our question bank from our main

.dart file,this no longer exists and it tells you as much. But how can we refer to our quiz\_brain?

Well we have to create a new quiz brain object. So first we have to go ahead and import our quiz\_brain

.dart,so let's put that in there.

And now we're able to create a new quiz brain object which is going to be called quizBrain.

And notice that when we name objects, they start out with a lowercase and we're going to set it to equal a new quiz brain object.

So that constructs a new object which we can now tap into by referring to quizBrain. Down here where we have our question bank, instead of using just the question bank which should exist locally,we're going to write quizBrain.questionBank, because our question bank is now a property of our quiz brain object. And it's right here. It's a list of questions. Now we no longer need to know about questions in our main.dart file and we can delete this unused import. Notice that every time you stop requiring some code in an import, it'll tell you by making it gray and giving it a squiggly line.

So let's delete that and we now need to only know about quiz brain inside our main.dart.

Now we've still got a few more errors down here.

So my challenge to you is to try and fix everything and run it to make sure that it still works in the same way.

Pause the video and give that a go. All right.

So down her, we're going to do exactly the same as what we did before. Because our question bank is now a property of our quiz brain,we have to use the dot notation instead.

We have the write quizBrain.questionBank. And similarly down here, we have to write quizBrain.

questionBank, so that we're actually tapping into the actual question bank. Now even though we had to add a little bit more code in our main.dart, at least our question bank which is now pretty massive, is separated into a separate class. And this now abstracts some of the functionality namely the brain for our quiz, into a separate object.

And now we've abstracted our code just a little bit more. And the other benefit is that say if we were to create a different quiz, if we were to create a sports quiz for example, we don't have to change anything about our main.dart.

We just have to provide a new file, say a sport quiz brain and everything in here will still work the same.

And we just have to change the class that we're using to construct the quiz brain.

So you could have a quiz that has lots of different types of quiz brain, sports and news and general knowledge and every time you want to switch between one to the other, you just change the object that you're actually using for the current quiz.

So we've made our code base more reusable more modular and we've separated different jobs into different classes.

Now if we run our app you can see that we now have more questions, 13 to be precise.

But here's a problem. In our quiz app,let's say that we really hate losing.

Well we can actually change the answer to our questions.

So for example here we're trying to get the correct answer by tapping into our quiz brain, looking at the question bank property and picking out the current question out of the question bank, then tapping into the answer.

And this is what we would call getting. We're getting the value of that correct answer.

But equally I can also change it.

I can simply write quiz brain question bank current answer and I can set it to a new value. I can set it to whatever it is that I picked, which in this case would be true right?

Well if this happens before I check my answer, then if we hit save and we take a look at the actual correct answers and what I get as a result here, you can see that we're on this question about the loudest sound produced by an animal and the correct answer should be false.

But I'm going to press true anyways and it tells me that I got it right.

So how is it possible? I shouldn't be able to just change that, even if I want to be right all the time.

This is not very safe for our app.

Let's comment that out, and in the next lesson we're going to learn how we can make our code a bit better and conform with the second pillar of OOP, which is encapsulation.

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