Now in the last lesson, I showed you that in addition to being able to get the value of any of the properties in our quiz brain, so this other class. We're currently also able to change any of the properties in our other class. And in certain cases, that can make our app behave in a way that we don't expect.

And so we get onto the second pillar of OOP which is encapsulation.

And just as you might want to encapsulate your workers in different office cubes so that they work on their own thing,you might also want to encapsulate your classes and your objects.

So recently I came across an interesting quiz on Inc. Magazine and it was about

Are you a micromanager? And it's a quiz that checks to see whether if you're basically trying to micromanage everybody else's time and you're actually trying to do everybody else's job. When we're abstracting our classes and we're trying to separate out different jobs into different roles and into different classes,what we want is for each class to have a separate role and a separate job and to keep them simple.

We want them to be just concerned with their own jobs and do their own jobs well.

So if we have a waiter class and we have a chef class, we don't want the waiter to be meddling with the chef's job.

We don't want the waiters to just come and, 'You're not cooking it right.

You should be doing it like this.'

That's not the waiter's job.

And this will lead to a really sad chef who will be really annoyed that he's getting micromanaged by the waiter.

But it also means that the waiters job gets more complex. And if the waiter was a class, then he would have more code and it would be more intermingled with the code of the chef.

And this leads to actually more complexity and not less complexity, which defeats the purpose of abstracting all classes into separate modules.

What do we do?

Well we could build a wall between the waiter and the chef.

And this is why the kitchen is usually separate from the restaurant right?

So that the chefs can do their own job and the waiters can do their own job.

So the chef is only concerned with cooking food and the waiters are only concerned with getting orders and delivering the food to people.

And you only have small windows between the chef and the waiter where there are things that they strictly definitely need to interact with.

So when the chef passes the food to the waiter for example.

So how can we do this for our code?

Now we saw in the last lesson that we probably shouldn't be able to change the value of the correct answers in our main.dart right?

It's none of the main.dart's business to change the question answers. It's very much to do with the quiz brain. The quiz brain should control what is true and what is false.

And ideally, the answers to the questions should never really change.

So what can we do in this case to encapsulate this question bank?

Well, we can make this property private. And we do that in Dart by simply adding a underscore in front of the name of the property.

So now, if we went back to our main.dart and we tried to meddle with the value of question bank, it doesn't work anymore.

In fact, we can't even tap in to that question bank anymore.

We can't say 'quiz brain give me your question bank' because question bank is now private. So it means only the quiz brain has access to the contents of this property and nobody else can.

So how do we make our app still work?

Well let's delegate the task of serving up these questions and answers to the person who's responsible for all things quiz related, the quiz brain.

So let's create a new function. Let's create something called a getQuestionText function. And remember inside a class it's now called a method but it's basically the same thing. And our get question text needs a integer passed in, which is the current question number.

So what's the question number that you actually want to get hold of?

And then it's going to return or output the actual question text for that question number.

So the way we would express that is we would happen to the question bank, which is private, but because we're inside the same class, we're able to access it. And then we're going to tap into the question bank at the index of question number.

So which question do we want? If that was equal to zero, then we're going to get hold of this question object and then we're going to tap into it's Question text, so .questionText. And now we can go back into our main.dart, and instead of trying to get hold of this private property which is none of our business to meddle with, we can simply ask our quiz brain to get me the question text, and get me the question text at this particular question number.

So to start with that will be zero.

And we're going to go into our quiz brain and call get question text on quiz brain, and we're going to get the first question, and get it's questionText. So now, you can see that we've actually not only just encapsulated our question bank and it's now safe from meddling from other parties, but we've also actually abstracted our quiz app a little bit more. So now as a challenge, I want you to be able to address the other two issues that we have.

Don't worry about this one.

But we need to get a hold of the correct answer when the user presses true, and when the user presses false. Pause the video and complete that challenge.

All right.

So first things first, let's go into quiz brain and we're going to get our quiz brain to serve up the correct answer.

So I'm gonna create a method that returns a boolean, and it's going to be called getCorrectAnswer or whatever it may be that you called it, doesn't really matter. And we gonna pass in again, the question number that we are trying to get the correct answer for.

So now we are going to return our question bank at the question number that they provided .questionAnswer. And now we can go back and do our main.dart and we can fix our code.

So instead of trying to get hold of the entire question bank into our main.dart, we're instead gonna say correct answers equal to quiz brain get me the correct answer for the current question number. And then we can compare to see if the correct answer is equal to true or down here if it's equal to False.

So let's fix this in the same way down here.

Correct answer should now be equal to get correct answer from the quiz brain. So now our app works and we're able to get hold of the correct answer and the correct question, but we're no longer able to just meddle with that question bank anymore.

We can't say quiz brain question bank,let's just set one of your answers to true. Because our question bank is now encapsulated inside its own class and only its own class.

The quiz brain can decide what it should do with this question bank.

So you can't accidentally in your main.dart, for example, completely mess up and change the answers or change the questions or even delete the question bank.

So we're a little bit safer because of encapsulation. If we run our app at the moment,you can see that it still works exactly the same way. But it will still crash once you get to the end of the 13 questions.

We've only got 13 questions and if we try to pull out the 13th item from our question bank, that's equivalent to the 14th question because lists of course start counting from zero.

So what can we do about this?

Well inside our quiz brain, we should be able to check to see if our question bank still has more questions to serve up.

And if so, then we'll serve it up but otherwise we should protect the app by not trying to get hold of a question that doesn't actually exist.

So how can we do this?

Well instead of keeping track of which question we're on in our main.dart which is not really its job, let's remove this and let's put it into our quiz brain. So I'm going to put it inside my quiz brain class as a property and in fact I'm going to make it a private property.

So only the quiz brain has access to this. So I can't accidentally say in main.dart or in question

.dart, suddenly sets my question number to 12 or 13 which will crash my app. And instead of calling getQuestionText and providing the question number from our main.dart, I'm going to use our questionNumber inside our quiz brain. So we need to change it here to that private question number. And in order to go to the next question, I'm going to create a method that's just called nextQuestion and all it does is it checks to see if it's safe to actually go to the next question. And that means we have to check to see if the questionNumber is less than our questionBank.length.

So we know that .length gives us the size of the questionBank.

So this is going to be 13 because we have 13 items in our question bank.

And when our question number is less than that, then theoretically, it means that it should be safe to go to the next question.

So then we could say questionNumber++. And this should increment our questionNumber every time we need to go to the next question.

So in our main.dart instead of updating our questionNumber, we're going to instead call upon our quiz brain to go to the next question instead. And we don't need to deal with our question numbers or keeping track of it inside our main.dart anymore.

Let's go ahead and change that in both places.

So it's gonna be quizBrain.nextQuestion. And we're gonna delete this print statement and also remember that get question answer no longer need an input and also get question text no longer needs an input.

So now let's run up from the start by clicking either the run there or clicking our hot restart. And we should start off from the beginning of our quiz.

So now if we go through our quiz and we get to the end you can see that our app still crashes. It's still saying that we're trying to get the 13th element of our list which only goes up to 12.

So what's the reason here?

Well let's add a print statement right here, so that we get to see what question number is.

And let's also print our questionBank.length. And now let's run our app again from the start, so that the first time when we click true and we go to the next question, the question number goes to one.

So we go to the next question and we have a total of 13 questions. So let's keep hitting true until we get close to crashing the app.

So now currently, our question number is 12. And it means that we're trying to pull out the 12th item out of our questionBank which still exists.

It's this last question right here,the question about West Virginia.

The next time that I click the true button, it will trigger the next question method. And now it's going to check to see if the question number at this point, which is 12, is less than questionBank.length which is 13.

This is true.

So it's going to go to the next question and then it's going to update the question text. And it's going to try and pull out the 13th item out of our list, which doesn't exist.

So right now, this is the point when our app crashes.

So how can we solve this problem?

Well, we can simply make our questionBank.length check it for one less than the size of the questionBank.

Now we never actually get to 13.

We get to 12 and the next time I click it, nothing else happens.

We are at the end of the quiz and our app no longer crashes.

Perfect. So now our app is being safety-checked inside our quiz brain, which is the logical place that should deal with all things quiz related.

And our main.dart simply just is concerned with displaying things on screen or clicking a button to go to the next question, and it doesn't have to deal with the quiz functionality anymore.