Most legacy applications are built without unit testing in mind. So in order to unit test them, you need to refactor them towards a testable and loosely coupled design. In a loosely coupled design you can replace one object with another at run time. So when unit testing a class that uses an object that talks to an external resource you can replace that object with a fake object, which we call a test double. Now there are three steps that you need to follow to achieve a testable and loosely coupled design. First, you extract the code that uses an external resource into a

separate class. So you put the code that talks to an external resource into a separate class, and isolate it from the rest of your code.

Next, you extract an interface from that class. Why an interface? Because an interface is like a contract.

It simply declares a bunch of methods and properties but none of these have an implementation, none of these methods have code.

An interface simply tells C# compiler that somewhere in the code there are probably one or more classes that implement this contract. So these classes implement the members declared in the interface. You can have two different classes that implement that interface. One is the real implementation that uses an external resource, the other is the fake one which we call the test double. Finally, you modify the class under test, you talk to this interface instead of one of its concrete implementations. So instead of being dependent on a specific implementation, it will be dependent only on the interface or the contract. You can pass any object that implements that interface, and this way the class becomes loosely coupled and testable. In practical terms, this means you should delete the lines where you create an instance of bad implementation using the new operator.

Because when you use the new operator inside a class, you're making that class tightly coupled or tightly dependent on a given implementation

So we cannot replace that implementation in your test with a fake object, okay? So instead of newing up a specific implementation, your program begins an interface or contract, and then pass on the implementation of that interface from the outside. Here's one way to do this. So, you can pass that as a parameter to a method. You can also pass a dependency by a constructor, or a property. We'll look at examples of each of these approaches shortly. So, when you program against interfaces, you can provide different implementations at different times. In your production code, you'll provide the real implementation that talks to an external resource in your tests, you provide a fake implementation, and this is what we call dependency injection.

Instead of newing up dependencies, we inject them from the outside. I get a lot of emails from my students asking me, Mosh, what is dependency injection? It's so complicated. It's not. Dependency injection is a 25-dollar term for a 5-cent concept. It simply means to inject or supply the dependencies of a class from the outside, and this makes your classes loosely coupled and testable. So next we'll look at these three steps in action.