EF Intro Lab

Using TDD, let's get Entity Framework set up in our project so we can read from the database. This takes a lot of work at first, but it will save us lots in creation and maintenance and will eliminate scads of duplicated code.

And bonus! In an attempt to reinforce good design practices, we're going to implement the repository pattern. Like using an ORM, it costs us time in setup, but gains us all kinds of flexibility and reduced coupling so it is totally worth it!

In order to use TDD techniques, we'll write automated tests but since they are testing more than one thing at a time (code and database), they're strictly speaking integration tests.

Writing the tests first

1. Add two new Class Library projects, one called IntegrationTests and another called Infrastructure.
2. Add some references. Both of these new projects should know about Core. IntegrationTests should know about Infrastructure and Nunit.Framework.
3. Add a class called ProductTests. Don't forget the TestFixture decorator.
4. Now take a deep breath and build because this is the last time your project will compile for a long time. :-O
5. Declare a private class-level variable like this:

```
private ProductRepository _repository;
```

6. Add a method called SetUp and decorate it with [SetUp] attribute.
7. In it, do this:

```
_repository = new ProductRepository();
```

8. Create a test in there called CanReadAProductGivenAProductId().
9. In it, do this:

```
var product = _repository.GetProductById(14);
Assert.AreEqual("Tofu", product.ProductName);
```

10. Create a test in there called CanReadAllProducts().
11. In it, do this:

```
var products = _repository.ReadAll();
Assert.AreEqual(77, products.Count);
```

Creating the ProductRepository

Now, of course it won't build, so let's fix that.

12. Add a new class to the Infrastructure project called ProductRepository.
13. Give it a private class-level variable:

```
private NorthwindContext _db;
```

14. In a parameterless constructor, do this:

```
_db = new NorthwindContext();
```

15. Create two method stubs, one called GetProductById() that receives an integer and returns a Product and one called GetAllProducts() that returns a List of Product.

Creating the EF Context

ProductRepository now exists, but we've added a new requirement for compling: we have to have a NorthwindContext.

16. Add a new class to the Infrastructure project called – you guessed it – NorthwindContext.

17. Add this property:

```
public DbSet<Product> Products { get; set;}
```

18. Finally! You should be able to build. Go for it!
19. You can build, but your test still don't pass. Your mission, should you choose to accept it, is to get the tests to pass.

Special keys

20. In the IntegrationTests project, create these classes: CustomerTests, EmployeeTests, and OrderTests.
21. Create these tests: CanGetCustomerById, CanGetEmployeeById, CanGetOrderById.
22. In the NorthwindContext class, add DbSets for these entities/tables: Customer, Employee, Order, OrderDetail.
23. You'll see that you have problems with OrderDetail and with Customer because their keys are nonstandard. To fix it, add a method in NorthwindContext that looks like this:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
  base.OnModelCreating(modelBuilder);
  modelBuilder.Entity<OrderDetail>().HasKey(a =>
    new { a.ProductId, a.OrderId });
}
```

Bonus! Refactor CanReadAProductGivenAProductId() into a TestCase, passing it several product ids and several product descriptions.

Bonus: Set up the other tables one-by-one.