```
            UserName = sourceDTO.UserName;
            Password = sourceDTO.Password;
            Email = sourceDTO.Email;
        }
        #endregion

        public  LoadStatus         loadStatus;

        public System.Int32 ID;
        public System.String  Name = null;
        public System.String  Address = null;
        public System.String  PhoneNo = null;
        public System.String  UserName = null;
        public System.String  Password = null;
        public System.String  Email = null;

    }//end class
}//end namespace
```

Let us understand this code, step-by-step. First note that each DTO class would be marked with an attribute (`Serializable`). This is because the DTOs would be transferred across the tiers, and as explained earlier in this chapter, we need to serialize data so that it can be transferred and deserialized by other applications. In our sample, if we have the `5Tier.Common` assembly running in the same process as the `5Tier.Business` and `5.GUI`, then this serialization process will not be necessary as there would be no cross-application domain call. But there can be cases when either the BL, the DAL or the GUI assemblies can be separated physically and located on a different tier. In these cases, there would be cross-boundary and cross-application domain communication. We will need to serialize all of the DTOs so that they can pass through the network across application domains. The `Serializable` attribute marks the class as serializable so that .NET runtime can handle this serialization work for us. You can get better control over the entire serialization/deserialization process by using custom serialization, where you need to write the code manually.

Next, in the `CustomerDTO` class, we have defined all of the attributes as public variables and set them to their default values. We should try to use only primitive data types in DTOs as they should be kept as simple as possible. If they are not generic enough, we can have problems in using them in remote calls by different language libraries (as they might not have those specific data types defined).

We have also defined a `copy constructor` in our DTO class, which just copies the data from another DTO to populate the attributes in the current DTO. We will see the importance of this when creating business layer classes.

Lazy Loading

An important thing to note here is a variable named `LoadStatus`. This is used to implement lazy loading, a design pattern that helps us make our application more efficient by loading only the required data. Using lazy loading pattern, we can defer the loading of all of the properties of an object until they are really needed. Let me explain with an example. In our OMS application, consider a form that shows the list of all of the customers in a grid. Now, in this form, only the `Customer ID`, the `first name` and the `last name` are shown, along with an `edit` and `delete` button. The `Customer address`, `email address`, `password` and other fields are shown only when someone edits an existing customer or adds a new one, a process which is done through another form. So if we want to load a list of `Customer` objects on this `Customer List` form, we don't need to fetch all the fields at once from the database. We only need to fetch the `Customer ID`, `first name` and `last name` fields to make our application more performance-efficient (by getting only the data required). And when we are on the `Edit Customer` form, we need to fetch all of the details. This can be done by having two methods in the DAL: one for a partial fetch and another for a complete fetch. But this approach is cumbersome, and we cannot always write two methods for each entity like this. So we follow the lazy loading design pattern, and use an enum named `LoadStatus` in our code, which can have one of three status values:

- `Initialized`
- `Ghost load`: object is partially loaded
- `Loaded`: object is completely loaded

We will see in the business and data access classes how we are using the lazy loading design to make our application more efficient.

Coming back to our DTO class, note that we have used the status value of `Initialized` in our default constructor when creating a new DTO. This indicates that the state of the DTO is initialized and there is no actual data in it, only default values. Also, as you might have noticed, we have not added any method here because the DTO is simply a data carrier, and not a business object in any sense.