Lab 1: Domain Models

Programming in C# for Visual Studio .NET Platform II

Objective

In this lab, we will begin work on the demonstration project for the course, starting with a Domain Model. This exercise covers a lot of ground, touching on requirements definition, architecture, organizing solutions in Visual Studio, simple class design and construction, object modeling, C# coding style, and many other related subjects in varying levels of detail. You should have enough background with C# and Visual Studio to follow the mechanics as we work through the design and construction of the domain model, building the classes, getting the solution to compile and then building a console application to test the model. However, it may not be very apparent why some parts of the model are designed as they are until we build out the user interface and database parts of the application in subsequent modules.

The sample application we are building will be called the "Talent Application". This application allows us to keep track of movies or other entertainment programming, which we will call Shows, and people who are credited with being on the cast or crew for these shows. Shows are also associated with one or more Genres, such as Comedy, Action, Drama, etc.

We will use a **Domain-Driven Design** approach to the application design and architecture, which starts by creating a **domain model** of classes that represent objects from the business domain of the intended users. We call such classes **Domain** classes and will begin by constructing these classes without too much regard for how the User Interface might look or how the database needs to be organized.

A big advantage of the domain-centric technique is that it facilitates communication with users in a language familiar to the users (called a **ubiquitous language**), and keeps the project on-track as the requirements evolve.

NOTE: This exercise is fairly long and gets somewhat tedious after Step 17, so feel free to just skim the instructions for the remainder of the Lab to get an idea of what I needed to do to complete the Domain Model for the example project. Then for the next Lab, you can continue building on my solution to this exercise.

The Talent Domain

Here are the key domain objects in the application:

A **Show** represents some sort of audio-visual entertainment – nominally a movie, but we chose the name "Show" as a somewhat more generic term that could apply to a television series episode or any other kind of entertainment product. Each show has a Title, a Theatrical Release

Date, a DVD release date and an MPAA Rating, though only the Title is a required attribute for a show.

A **Genre** is a general category of show, such as Drama, Comedy, Action, Documentary, etc.

A Show is associated with zero or more Genres, so you can say that an additional attribute of a show is a (possibly empty) collection of associated Genres.

A **Person** represents a real person, and has the following important attributes in show business:

- Salutation: The prefix for their name (Ms., Mr., Dr., Rev., etc.)
- First Name
- Middle Name
- Last Name
- Suffix: (PhD, M.S., CPA, Esq., etc)
- Date of Birth
- Height (in inches)
- Hair Color
- Eye Color

If a Person is involved in the production of a Show, they are given a **Credit** which represents their participation in some capacity. There is one Credit for each role the Person plays in the Show, and the Credit indicates what job the Person is credit with as a **CreditType**, such as Actor, Director, Writer, etc. So, for example, if Clint Eastwood is both an actor and a director in a movie, there will be separate credits for his two roles as actor and director. If the CreditType is actor, we also need to be able to keep track of the character name(s) as just a free-form text entry field.

We would also like for the application to be able to allow us to store and retrieve files associated with each Person, which we will call **Attachments**. These can be images or files of any sort that we'd like to upload, so the application will need to keep track of a description of the attachment, as well as the original file's contents, name and file extension.

That's pretty much all there is to our Domain Model. As you can see, the terms used to describe the domain are taken from the ubiquitous language, so the domain model is constructed using software constructs that directly represent the users' reality. This is a simple enough application that it should be easy to envision the requirements, but complex enough that we will need to deal with most of the major architectural considerations that you will encounter in building real-world line-of-business applications.

When designing any application, we need to begin with a pretty clear vision of what tasks the application user needs to perform, and write down those tasks in some form. For an application like the Talent application this will be pretty straightforward, but still helpful as guide during the design and development process. A rough outline of the task(s) is called a **User Story**. If we

explicitly list each step in the interaction between the user and the application, it is called a **Use Case**. As the development proceeds, it is very easy to lose sight of what the user is attempting to do with the application, so frequently referring back to the User Stories or Use Cases helps us to make sure that we don't accidentally introduce extra fluff in our application that does not help the user accomplish their tasks.

It is pretty easy to envision a few use cases that the application would need to support:

The Show Use Case: The user accesses a list of Shows. He can then either choose to edit an existing show or elect to create a new one. In either case, the user can enter or edit the relevant characteristics of the show by entering or modifying the title, run time, theatrical release date, and MPAA Rating, and associate the show with one or more pre-defined genres. He can also add credits to the show by associating the show with a person, a credit type and possibly entering the name of the character portrayed if it is an acting credit. As mentioned earlier, a person can have multiple credits for a show if they perform multiple jobs, such as writer and director. When the user saves his changes, all the new information about the show and its associated credits should be saved to a data store for future reference within the same application session, or potentially by other users from different instances of the application user interface.

The Person Use Case: The user accesses a list of People. She can then choose to edit an existing person or elect to create a new one. In either case, the user can enter or edit the relevant characteristics of the person by entering or modifying various parts of their name, date of birth weight, height, hair color, and eye color. She can also add any credits to the person by associating the show with a show, a credit type and possibly entering the name of the character portrayed if it is an acting credit. When the user saves her changes, all the new information about the person and their credits should be saved to a data store for future reference within the same application session, or potentially by other users from different instances of the application user interface.

A number of minor use cases could also be defined for deleting shows or people, maintaining the lists of Genres, Credit Types or MPAARatings.

Notice how the Use Case descriptions do not explicitly refer to user interface components such as screens, buttons, menus, etc., and do not use technical jargon. This is a good practice to adopt, since the use cases should only describe what the user needs to do, and not dictate how the user interacts with the system to accomplish each step. Though we will be building a graphical user interface client application, the use cases you write should be equally applicable to building an application that uses speech recognition and is implemented as a "conversation" between the user and the application, as an example. In short the use cases should describe ***what*** the user needs to do and not ***how*** she interacts with the application.

For large applications, identifying domain objects and developing use cases is even more critical, and should be developed jointly with representative users or subject matter experts. Since the

domain objects use terms from the ubiquitous language, both users and developers should be able to fully understand and contribute to the refinement of each use case. Good use cases will be the primary agreement between users and the development team about what should be built. In my experience, developing use cases frequently causes users to re-examine the way they have been doing their work for many years and triggers significant changes in their entrenched business processes as a by-product of the application development.

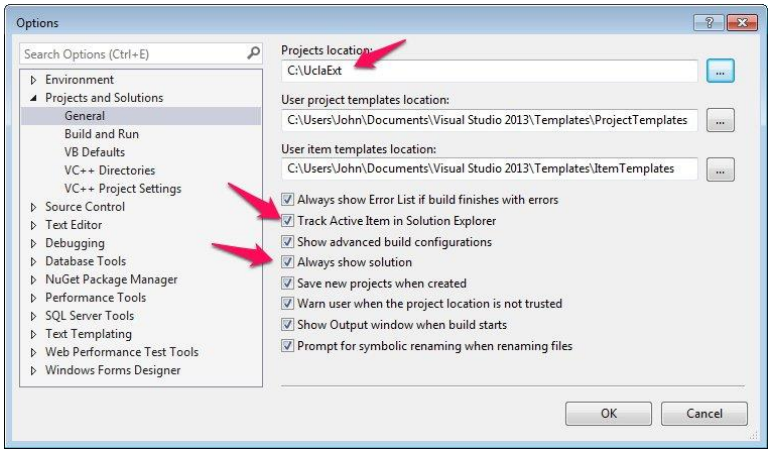
These use cases provide guidance as we develop the project by determining the operations we need to support. For example, notice that we only ever need to save changes to a Credit within the context of editing the related Show or Person for the corresponding use case. This is a principle of Domain-Driven Design called an **Aggregate Root** pattern. For the Person use case, the Person is the "Aggregate Root" object, meaning that when we save changes to a Person, we are saving changes to all the properties of a single Person, plus the various domain objects associated with the person (i.e., Credits). Likewise, the save operations in the Show use case revolve around the Show aggregate root. Domain objects that are not an aggregate root are only edited as dependent children of some other aggregate root object.

Creating the Solution and Domain Project

To get started, we will dive right into Visual Studio and build a project that represents the various domain objects outlined above. Depending on your prior experience, the Visual Studio solution organization may seem a bit more complex than you are used to. We will talk more about the overall architecture as the course progresses, but one course objective is for you to become proficient in managing solutions that contain many inter-dependent projects.

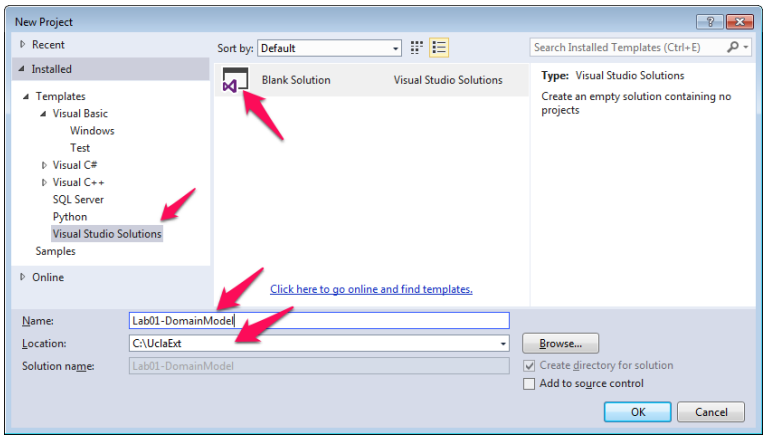
Along the way, we will review a few basic concepts such as project creation, namespaces, class definitions, properties and enumerations.

Step 1: Create a folder C: \UclaExt to hold your class labs. You can use a different folder location if you wish, but all of the lab materials will assume that you have a C:\UclaExt, so you will need to translate the instructions accordingly if you use a different folder structure. Then select the Tools |Options menu and highlight the Projects and Solutions item in the tree view:



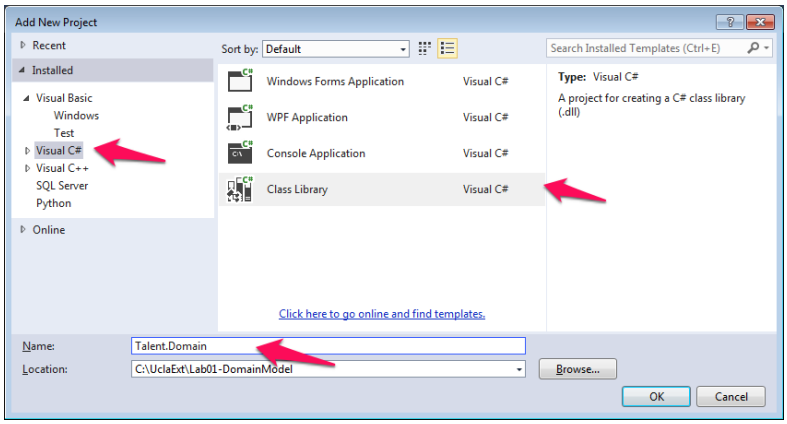
I would recommend setting your Projects location to the directory you just created. This way, each time you create a new solution, it will be created in this directory, so all your class work is kept together. I'd also recommend checking the "Track Active Item in Solution Explorer" and "Always show solution options", so your IDE behavior more closely follows the lab instructions.

Step 2: Create a Blank Solution. Open VS and select the File | New Project menu item. Create a new blank solution called "Lab01-DomainModel".

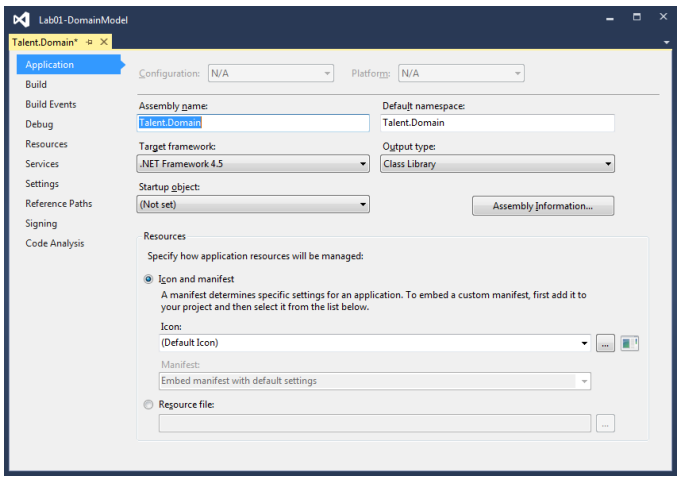


Use Windows Explorer to verify that this created the folder c:\UclaExt\Lab01-DomainModel, and created the Lab01-DomainModel.sln and Lab01-DomainModel.v12.suo files within this folder. The *.sln file contains the actual solution information – the *.suo file just contains a bunch of information about the arrangement of Visual Studio the last time you worked with it, such as what source files were open (confusingly enough, "v12" refers to Visual Studio 2013). I will always start building an application with a Blank Solution and then add projects to it, as needed. This allows me to easily specify the solution name and to be consistent in my procedure for solution organization. This may be new to you if you are accustomed to building single-project solutions.

Step 3: Add a Class Library project called Talent.Domain to your solution. Right-click on the Solution node in Solution Explorer and select Add | new Project from the context menu. Be sure to select Visual C# in the left-hand list box and to choose Class Library from the list of templates in the central pane. Name your new project "Talent.Domain". This will set up the project with a default namespace of Talent.Domain, so any types you create will have a fully-qualified type name that starts with Talent.Domain. Click OK to create the new project.



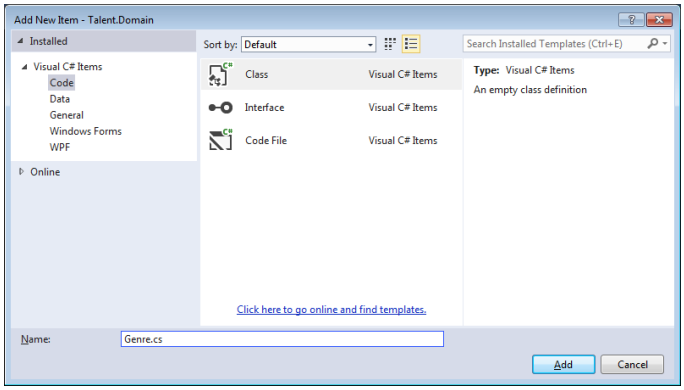
If you do this correctly, a new node representing this project will appear in the Solution Explorer under the Lab01-DomainModel node, called Talent.Domain. If you check the properties of project (right-click on the Talent.Domain node in Solution Explorer and select Properties from the context menu), they should look like this:



The Assembly name indicates that this project will compile to an assembly with the name "Talent.Domain.dll", and the Default namespace means that any class you create will, by default have the namespace Talent.Domain – the assembly name and namespace do not have to be the same, but it is usually a good practice for the assembly name and default namespace to be the same. Make sure that the Output Type is Class Library.

If you right-click on the project in Solution Explorer and choose the Open Folder in File Explorer context menu item, it should open Windows Explorer and show you a directory containing a Talent.Domain.csproj file. In Solution Explorer, you can select the Class1.cs node and press delete to delete the class file that Visual Studio created in a misguided attempt to be helpful.

Step 4: Create a Genre Domain Class. We will start by creating the class that represents a single **Genre**. Right-click on the Talent.Domain project in SE and select Add | New Item



Make sure you select the Class template and the language shows as Visual C# (not Visual Basic) and give the class a name of Genre.cs, then press Add to add the beginnings of a source code file to the solution.

If you double-click on the new Genre.cs file, VS should open the source code editor and show you the beginnings of a class definition like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.Domain
{
    class Genre
    {
    }
}
```

Note that the namespace declaration identifies the namespace that will be used for anything created inside the body of the namespace declaration (i.e., outer braces), so the fully-qualified name of the Genre class will be Talent.Domain.Genre. Edit the file to declare the Genre class like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.Domain
{
    public class Genre
    {
        #region Constructor

        #endregion

        #region Fields

        private int _id;
        private string _name;
        private string _code
        private bool _isInactive;
        private int _displayOrder = 10;
```

```
        #endregion

        #region Properties

        public int Id
        {
            get { return _id; }
            set { _id = value; }
        }

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }

        public string Code
        {
            get { return _code; }
            set { _code = value; }
        }

        public bool IsInactive
        {
            get { return _isInactive; }
            set { _isInactive = value; }
        }

        public int DisplayOrder
        {
            get { return _displayOrder; }
            set { _displayOrder = value; }
        }


        #endregion

        #region Overrides

        public override string ToString()
        {
            return Name;
        }

        #endregion
    }
}
```

where the bold blue code is new.

Important: *Note the public access modifer before the word "class". If you omit the access modifier for a class, the compiler will compile the class as if you had set the access modifier to "internal", which would prevent accessing the class from outside the Talent.Domain project. In most of the classes we create, the class should be declared public like this so it can be referenced by code from other assemblies (projects). This is one of the most common sources of problems when you first start working with multi-project solutions.*

Id: When working with Domain Model classes, we need a way to uniquely identify a specific instance of an object. A common convention that we will adopt is to just name the property "Id", which will serve as the "key" to identify a specific object (Genre, in this case). When we create a new instance of a domain object, its Id will be assigned a value of 0 by default, but when it is stored permanently or needs to be referenced by other objects, it will need to be

assigned a unique non-zero value. Think of this as a sort of a serial number to unambiguously identify specific instances of the domain objects.

Name: This is the full name of the Genre. We'll eventually specify that the name can be up to 50 characters long. Even though the string datatype can hold very long names, as a practical matter, we want to place a reasonable length limit on string properties so we can layout the User Interface screens with some idea how much space to allot for the data, and to define the reasonable limits on the size of the corresponding columns in our database or other data store, where we may have technical limits on the length of a piece of string data.

The **Code** property is an abbreviated name that we will be able to use in our application when there is not much room for display. This allows the Name to somewhat longer and more descriptive and still gives us an alternative short name to show when space is limited.

The **IsInactive** boolean property can be set ot true to "retire" a particular Genre object without physically removing it from the system. This is often helpful when we cannot remove the object because existing objects reference it, but we still want to flag the entry as deprecated so we have a way of preventing it from being used in the future if we need to.

The **DisplayOrder** property is an integer that can be set up to control the order in which a list of Genre objects is presented to the user. This is helpful when the best order to display items is not necessarily in alphabetical order by name. List Boxes and other controls can then sort the Genres by DisplayOrder and then by Name, so configuring the Display Order for different Genres allows us to easily control the Genre ordering in list controls. I like to set up a default value of 10. This way, if users never bother to change the display order of any Genres, the end result is that the Genres will be listed alphabetically, but if we want to move a particular Genre to the top of the list for some reason, we can just change that Genre's DisplayOrder to 5 or something. If we used a default value of 0 (as C# would for an integer), then when we wanted to promote a Genre to an earlier position in the list, we would have to use negative numbers, and it's somewhat easier if users can just work with non-negative numbers when arranging the order.

There are some important details about this particular implementation that I want to point out:

Full vs. Automatic Property Syntax: In the code, the IsInactive property is implemented like this:

```
private bool _isInactive;

public bool IsInactive
{
    get { return  isInactive; }
    set { _isInactive = value; }
}
```

This is "full property syntax", where the _isInactive private field is a **backing field** that is part of the state of the object and is not directly accessible by code outside the class itself – it is safely encapsulated within the class and only code defined in the class can manipulate it. The **property declaration** for the IsInactive property is a "wrapper" around the backing field that

allows users of the class to get or possibly set the value of the _isInactive field. If we omit the get{} clause from the property definition, code outside the class will not be able to retrieve the value, making the IsInactive property "write-only" for consumers of the class. If we were to omit the set{} clause from the property declaration, the property would be "read-only" for consumers of the class. I will use the widely adopted convention that member field variables like _isInactive are named with camel casing and prefixed with an underscore, while property names will use Pascal casing.

You may be familiar with automatic property syntax – an alternate way to write properties in C# that does not explicitly use a backing field. Instead of the "long form" property syntax shown above, C# allows a more compact syntax like this:

```
public bool IsInactive { get; set; }
```

At compilation, the compiler will still create the backing field in Intermediate Language, but it will not have a name and therefore cannot be referenced in source code – other than that, the property will function exactly as if it were written in the more verbose long form that we are using at this time.

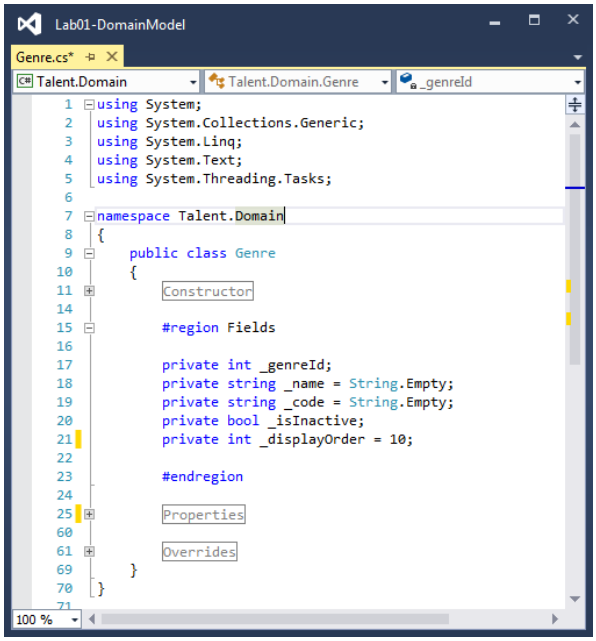
So why am I choosing to use the full property syntax? For now, the primary reason is that I want to control how properties are initialized with default values as described shortly. For example, the DisplayOrder property is defined like this:

```
private int _displayOrder = 10;

public int DisplayOrder
{
    get { return _displayOrder; }
    set { _displayOrder = value; }
}
```

so I can initialize the _displayOrder field with a default value of my choice – something that automatic property syntax does not really allow (though I could introduce a default constructor and set the property to my chosen default value in the body of the constructor and still use the automatic property syntax).

As the development progresses, we will also find that we need to add a bit of additional logic to the setter methods and would need to change from automatic property syntax to the "long form" anyway. To avoid that re-work, we'll just start with the long form.

Code Organization: To keep the code tidy, I separated the various parts of the class definition into regions. Regions begin with a #region <Region Name> and end with an #endregion construct. This is merely a developer convenience – within VS you can collapse or expand regions as an aid to viewing the parts of the class by clickin on the + or – icon in the left margin of the code editor. For example, if I just want to examine the field declarations, I can collapse all the regions except for the Fields region in the code editor:



For small classes like this, such organization isn't really necessary, but you can imagine that it makes it much easier to find your way around a class if you adopt a consistent organization of code within your classes and regions help. I like to show the Constructors, if any, first, followed by the field declarations, properties, overrides, and then any methods. If you are working with a team of developers, you will probably have coding standards that recommend or require your code to adhere to a specific organization.

Default Values: When you declare a member variable as we are doing for the backing fields and do not initialize the variable with a value (as for the _isInactive field in this example), C# will automatically assign the value the default value for the corresponding data type. Since the C# default for bool is "false" and that is what I want, I did not need to explicitly initialize _isInactive to false. However, in many cases, there is a good default value. As mentioned earlier, though C# would initialize _displayOrder to 0, I prefer to use a default value of 10.

Override the ToString() method: Every class in C# inherits from the System.Object class, which has just a few methods. It is a really good idea to override the ToString() method in each of the classes you create to return some sort of reasonable description of the object. Visual Studio and many features of the .Net framework will display the results of a call to the ToString() method by default in Visual Studio and various .Net framework classes (such as WPF List Boxes, which we will see later). If you do not override the ToString() method in your class definition, then the base Object class will just display the fully-qualified class name of the object's type (e.g., "Talent.Domain.Genre" for this class), which just isn't terribly informative. You will find it very helpful to always override the ToString() method in your classes, and have it return some sort of string describing the instance – in this example, it just returns the genre's Name property.

You should be able to build the solution. Nothing should happen, except that the Output Window will indicate that the build succeeded.

Step 5: The CreditType class. To represent the various types of Credits (Actor, Director, etc.), we can define a CreditType class. This is virtually identical to the Genre class, except for the name of the class itself, and looks like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.Domain
{
    public class CreditType
    {
        #region Constructor

        #endregion

        #region Fields

        private int _id;
        private string _name;
        private string _code;
        private bool  isInactive;
        private int _displayOrder = 10;

        #endregion

        #region Properties

        public int Id
        {
            get { return _id; }
            set { _id = value; }
        }

        public string Name
        {
            get { return _name; }
            set { _name = value ?? String.Empty; }
        }

        public string Code
        {
            get { return _code; }
            set { _code = value ?? String.Empty; }
        }

        public bool IsInactive
        {
            get { return _isInactive; }
            set { _isInactive = value; }
        }

        public int DisplayOrder
        {
            get { return _displayOrder; }
            set { _displayOrder = value; }
        }


        #endregion

        #region Overrides
```

```
        public override string ToString()
        {
            return Name;
        }

        #endregion
    }
}
```

It should bother you that, except for the class name, the code is identical. We can easily fix this by creating a base class called LookupBase like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.Domain
{
    /// <summary>
    /// Base class for various "lookup table" type classes.
    /// </summary>
    public abstract class LookupBase
    {

        #region Fields

        private int _id;
        private string _name;
        private string _code;
        private bool  isInactive;
        private int _displayOrder = 10;

        #endregion

        #region Properties

        public int Id
        {
            get { return _id; }
            set { _id = value; }
        }


        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }

        public string Code
        {
            get { return _code; }
            set { _code = value; }
        }

        public bool IsInactive
        {
            get { return _isInactive; }
            set { _isInactive = value; }
        }

        public int DisplayOrder
        {
            get { return  displayOrder; }
            set { _displayOrder = value; }
        }
```

```
        #endregion

        #region Overrides

        public override string ToString()
        {
            return Name;
        }

        #endregion

    }
}
```

We can then re-write the Genre class like this:

```
namespace Talent.Domain
{
    /// <summary>
    /// Represents a Genre that can be associated with a Show.
    /// </summary>
    public class Genre : LookupBase
    {

    }
}
```

And the CreditType class like this:

```
namespace Talent.Domain
{
    /// <summary>
    /// Describes the type of job a person did for a show (e.g.,
    /// Actor, Director, etc.).
    /// </summary>
    public class CreditType : LookupBase
    {

    }
}
```

This may seem a bit strange to have a concrete classes that add no functionality to the base class, but doing so allows us to segregate the different types of lookup objects into separate types, allowing us to strongly type the different lists we need.

Step 6: Create "lookup classes" for HairColor, EyeColor and MpaaRating. With the Genre and CreditType classes as an example, define domain classes for HairColor, EyeColor and MpaaRating. The HairColor and EyeColor classes will have no additional methods or properties not found in the LookupBase class, but the MpaaRating class should add one more string property called Description, and change its ToString() method to return the Code instead of the name (since users are accustomed to seeing ratings expressed by their code (e.g. "PG" instead of "Parental Guidance Suggested"). Compile your application to find any synatx errors.

At this point, we have domain classes for all of the simple domain objects, so we are ready to look at implementing the two non-trivial classes, Show and Person.

Step 7: Create the Show domain class definition. Add another class to the Talent.Domain project to represent a Show. The class should look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.Domain
{
    public class Show
    {
        #region Constructor

        #endregion

        #region Fields

        private int _id;
        private string _title;
        private int? _lengthInMinutes;
        private DateTime? _theatricalReleaseDate;
        private DateTime? _dvdReleaseDate;
        private int? _mpaaRatingId;

        #endregion

        #region Properties

        public int Id
        {
            get { return _id; }
            set { _id = value; }
        }

        public string Title
        {
            get { return _title; }
            set { _title = value; }
        }

        public int? LengthInMinutes
        {
            get { return _lengthInMinutes; }
            set { _lengthInMinutes = value; }
        }

        public DateTime? ReleaseDate
        {
            get { return _releaseDate; }
            set { _releaseDate = value; }

        public int? MpaaRatingId
        {
            get { return _mpaaRatingId; }
            set { _mpaaRatingId = value; }
        }

        #endregion

        #region Overrides

        public override string ToString()
        {
            return Title;
        }

        #endregion
    }
}
```

Several parts of this class definition that might be new to you are:

The `int?` keyword is a C# compiler alias for the generic struct type `Nullable<Int32>`, a structure that allows us to represent a data type that is normally an integer, but can have the special value `null`, which means that no value is assigned. This is important for many domain object properties where we want to allow for the possibility that a value is not known. The normal `int` type, in contrast, has a default value of 0 and does not have a way of being set to "nothing". When working with an `int?`, you can check to see if the variable has a value through the `HasValue` property. That is, `LengthInMinutes.HasValue` will evaluate to true if the property is assigned a value, and the value itself can be obtained as an int by the expression `LengthInMinutes.Value`. The primitive data types can all have a nullable version, except for `System.String`, which is a reference type and can be a null reference if it does not point to an allocated string object.

The **MpaaRatingId** property holds a nullable integer value that refers to another domain object (an MpaaRating instance) by it's Id value. This property can be null, indicating that the Show does not have an assigned rating, or it can have a value which links it to an object of type MpaaRating. Keep in mind that the same MpaaRating object could be referenced by several different Show objects. This property then represents a "many-to-zero or one" relationship from Shows to MpaaRatings – i.e., several different Show objects can reference each MpaaRating, but a Show object can reference a single MpaaRating (or none), since the MpaaRating property is scalar – it cannot hold multiple references to different MpaaRating objects. In Object-Oriented parlance, we say that the Show **contains** an MpaaRating. When we create our database, the MpaaRatingId will be called a **foreign key** to the MpaaRating.

Step 8: Create the Person class. Now we can create a Person class. This uses the same basic principles as the other classes, but we'll give it a few additional features:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.Domain
{
    public class Person
    {
        #region Constructor

        #endregion

        #region Fields

        private int _id;
        private string  salutation;
        private string _firstName;
        private string _middleName;
        private string _lastName;
        private string _suffix;
        private DateTime?  dateOfBirth;
        private double? _height;
        private int _hairColorId;
        private int _eyeColorId;

        #endregion
```

```
#region Properties

public int Id
{
    get { return _id; }
    set { _id = value; }
}

public string Salutation
{
    get { return _salutation; }
    set { _salutation = value; }
}

public string FirstName
{
    get { return _firstName; }
    set { _firstName = value; }
}

public string MiddleName
{
    get { return _middleName; }
    set { _middleName = value; }
}

public string LastName
{
    get { return _lastName; }
    set { _lastName = value; }
}

public string Suffix
{
    get { return _suffix; }
    set { _suffix = value; }
}

public DateTime? DateOfBirth
{
    get { return _dateOfBirth; }
    set { _dateOfBirth = value; }
}

public double? Height
{
    get { return _height; }
    set { _height = value; }
}

public int HairColorId
{
    get { return _hairColorId; }
    set { _hairColorId = value; }
}

public int EyeColorId
{
    get { return _eyeColorId; }
    set { _eyeColorId = value; }
}

#endregion

#region Computed Properties

public string FirstLastName
{
    get
    {
```

```
                /* Write code here to return the name formatted as
                        John Doe
                */
            }
        }

        public string LastFirstName
        {
            get
            {
                /*
                   Write code here to return the name formatted as
                   John Doe
                */
            }
        }

        public string FullName
        {
            get
            {
                /*
                   Write code here to return the name formatted as
                   Mr. John Q. Doe, Esq.
                */
            }
        }

        public int? Age
        {
            get
            {
                /*
                   Write code here to return the person's age as
                   An integer number of years based on the current date
                   and the DateOfBirth field. It should be null only
                   if the DateOfBirth is null.

                */
            }
        }
        #endregion

        #region Overrides

        public override string ToString()
        {
            return FirstLastName;
        }

        #endregion
    }
}
```

This should be about what you might expect at this point, though there are a few items worth mentioning:

The `HairColorId` is a link to the corresponding HairColor object, and the `EyeColorId` property is similarly a link an EyeColor object. There are a few different ways we could deal with the possibility that the Hair or Eye Color is not specified. For example, we could make the HairColorId property nullable. Instead, I will just create a HairColor choice that has HairColorId = 0 with the name "Unspecified" and Code value "Un" to represent an unspecified HairColor.

The above code shows partially completed "stubs" for some handy read-only properties. See if you can provide implementations for the FirstLastName, LastFirstName, FullName and Age properties based on the previously defined properties. You will need to be careful to handle the possibility that parts of the name may be null or empty, and perhaps do a bit of research to figure out how to compute the Age from the DateOfBirth and the current date. In an upcoming exercise we will create some unit tests to see if your computed property definitions handle several important cases correctly.

Note: We could have used methods instead of read-only properties to get the age or computed name properties, but we will see that properties have better data-binding support when we eventually start building user interfaces. As a rule of thumb, you should use properties when the intent is to just set or retrieve the property value, there are no potentially long-running calculations involved, and the act of setting or getting the property doesn't alter the state of the object. Basically, users of a class expect property reads and writes to be very fast and have no side-effects.

The `ToString()` override just returns the `FirstLastName`, since that is a pretty unique and helpful way to represent the Person in a string.

Step 9: The Many-To-Many Relationship between Shows and Genres. We have already seen how an object can have a link to another object when we created an MpaaRatingId property that links to an MpaaRating object in creating the Show class. This represents a many (Show) to one (MpaaRating) relationship, since zero, one or more than one Show can link to the same MpaaRating. However, a Show can have zero, one or more than one Genre, and conversely, a Genre can be associated with zero, one or more than one Show, so there is a ***many-to-many*** relationship between Shows and Genres. There are a few ways to represent many-to-many relationships in an object model, but we are going to choose a technique that will prove quite compatible with relational databases.

The idea is to introduce another object that represents an association between the two objects in the many-to-many relationship. We'll call this intermediate object a `ShowGenre`, since each instance will represent an association between one `Show` and one `Genre`.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.Domain
{
    public class ShowGenre
    {
        #region Fields

        private int  id;
        private int _showId;
        private int _genreId;

        #endregion
```

```
        #region Properties

        public int Id
        {
            get { return _id; }
            set {_id = value; }
        }

        public int ShowId
        {
            get { return _showId; }
            set { _showId = value; }
        }

        public int GenreId
        {
            get { return _genreId; }
            set { _genreId = value; }
        }

        #endregion
    }
}
```

As you can see, each `ShowGenre` has an Id identifying the `ShowGenre` association itself, plus a link to one `Show` and a link to a `Genre`. Several `ShowGenres` can be associated with the same `Show`, establishing a many (`ShowGenre`) to one (`Show`) relationship, and Several `ShowGenres` can be associated with the same `Genre`, establishing a many (`ShowGenre`) to one (`Genre`) relationship – we have effectively converted what was a single many-to-many relationship into two many-to-one relationships by introducing the ShowGenre association object.

Step 10: The Credit Class. Building on the idea of converting a many-to-many relationship into multiple many-to-one relationships one step further, consider the credits. In this case we have a many-to-many relationship between people and shows, but it is also possible for one person to play multiple roles on a show. For example a single person might be both an actor and a director, or portray multiple characters on a show. This is effectively a many-to-many-to-many relationship between Shows, People and CreditTypes. Again we introduce an association object, but rather than calling it ShowPersonCreditType, we'll just call it a Credit, since that is what the users would call it as part of the ubiquitous language anyway. Here is what the Credit class looks like:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.Domain
{
    public class Credit
    {
        #region Fields

        private int _id;
        private int  showId;
        private int _personId;
        private int _creditTypeId;
```

```
        private string _character = String.Empty;

        #endregion

        #region Properties

        public int Id
        {
            get { return _id; }
            set { _id = value; }
        }

        public int ShowId
        {
            get { return _showId; }
            set { _showId = value; }
        }

        public int PersonId
        {
            get { return _personId; }
            set { _personId = value; }
        }

        public int CreditTypeId
        {
            get { return  creditTypeId; }
            set { _creditTypeId = value; }
        }

        public string Character
        {
            get { return  character; }
            set { _character = value ?? String.Empty; }
        }

        #endregion
    }
}
```

The other thing I did here is add a Character property, which is used to enter the character played by an actor, if applicable, as a free-form string property.

Step 11: PersonAttachment object. To store the attachments, we'll have a PersonAttachment domain object. Since the original attachment is a file, any easy way to keep track of what kind of file we're dealing with is to record the Windows file extension. We'll also keep the original file name, so we can use that as a default file name when the user wants to export an attachment from the application. The contents of the file can be stored as array of bytes, which is easy to get from the file system or write to the file system as a binary stream without regard for details like encoding. We'll also allow the user to add an optional Caption describing the attachment.

```
namespace Talent.Domain
{
    /// <summary>
    /// A file that can be attached to a Person.
    /// </summary>
    public class PersonAttachment
    {
        #region Fields

        private int _id;
        private int _personId;
        private string _caption;
        private string _fileName;
```

```
        private string _fileExtension;
        private byte[] _fileBytes;

        #endregion

        #region Properties

        public int Id
        {
            get { return  id; }
            set { _id = value; }
        }

        public int PersonId
        {
            get { return  personId; }
            set { _personId = value; }
        }

        /// <summary>
        /// Optional Description of File
        /// </summary>
        public string Caption
        {
            get { return  caption; }
            set { _caption = value; }
        }

        /// <summary>
        /// Default root file name for downloading purposes
        /// (without path or file extension)
        /// </summary>
        public string FileName
        {
            get { return _fileName; }
            set { _fileName = value; }
        }

        /// <summary>
        /// Standard Windows file extension that determines the file
        /// format (e.g., jpg, png, docx, xlsx)
        /// </summary>
        public string FileExtension
        {
            get { return  fileExtension; }
            set { _fileExtension = value; }
        }

        public byte[] FileBytes
        {
            get { return _fileBytes; }
            set { _fileBytes = value; }
        }

        #endregion

        #region overrides

        public override string ToString()
        {
            return (FileName ?? "") + "." + (FileExtension ?? "");
        }

        #endregion
    }
}
```

Step 12: Navigating the Object Model. With the set of domain classes created so far, we can create instances of each type of object and populate collections with these instances to represent any "real-life" objects we need. We have also designed our model such that all relationships among objects in the model are many-to-one, introducing "association" objects to decompose any many-to-many relationships into two or more many-to-one relationships, and we set up the object model such that many-to-one relationships are represented by a single property of the Object on the "many" side of the relationship (e.g. the MpaaRatingId property of a Show object) to the identifying field of the object on the "one" side of the relationship (e.g., the Id of the related MpaaRating object). In this object model, it is possible in code to navigate from an instance of any object to any related object through these references.

If we have a Show object and need to obtain the related MpaaRating, we can locate the MpaaRating with a simple method like this:

```
public MpaaRating FindMpaaRatingById(List<MpaaRating> ratings, int id)
{
    foreach (var r in ratings)
    {
        if (r.Id == id) return r;
    }
    return null;
}
```

If, on the other hand, we have an MpaaRating object and want to find the shows that use it, we can get a list of such shows with a method like this:

```
public List<Show> FindShowsByMpaaRatingId(List<Show> shows, int mpaaRatingId)
{
    var list = new List<Show>();
    foreach (var s in shows)
    {
        if (s.MpaaRatingId == mpaaRatingId)
            list.Add(s);
    }
    return list;
}
```

So at this point we have a pretty good object model that allows us to create collections of domain objects and navigate around among the objects in the collection. This would be sufficient if all we needed to do was load objects into our model and then access them in a read-only fashion.

Step 13: Editing and Aggregate Root Objects. Let's now consider what happens when we want to edit some objects in our object model. A good editing operation goes like this:

1. Obtain "working copies" of the set of related object(s) to edit.
2. Make changes to the working copies of the objects.
3. If we are partway through the editing session and decide to cancel the editing, we can just throw away the "working copies" and the original objects are unmodified.
4. Otherwise, when we decide to save our changes, a sequence of operations is performed to update the permanent objects.

An important question to ask for each type of editing session is “What set of objects do I need in my “working copy” of the object model?”

To help sort this out, let’s re-visit some of our use cases:

In the Edit Genres use case, the user would first retrieve a read-only list of all Genres in the system then select a single Genre to Insert, Edit or Delete – but let’s focus on the Edit operation. When we are editing the list of Genres, the only domain object we will be editing is an instance of a Genre in the editing session, so the “working set” of objects in this case is just one Genre object. So the editing session consists of getting a single Genre object, using the user interface to manipulate one or more properties, and then replacing the old genre in our “master” object model with the updated genre object.

However, in the Edit Show use case, the user would first view a read-only list of Shows from the system, and then select a Show to edit. When editing a Show, we again need to make working copies of the objects we want to edit, but to edit a show we need not only the show object, but also the related objects that “belong to” the show. In particular, the objects that can be modified from a show editing session are the Show itself, the show’s ShowGenres and the Credits. So the working copy consists of an object graph containing a parent or “root” Show object, along with a collection of Credits and a collection of ShowGenres.

In either case, we can identify an object graph that has a root object. In the Edit Genre case, it is a degenerate object graph consisting of a single Genre object, while in the Edit Show case, it is a parent “Aggregate Root“ Show object, along with two child collections.

As you can see, having a clear set of use cases is invaluable in identifying the Aggregate Roots and corresponding object graphs. For each editing operation in our use cases, we should be able to identify the appropriate aggregate root and object graph.

Since our domain model needs to support each of our use cases, we want to make sure that each aggregate root object in our model actually contains all the child objects in its object graph for editing. For example, if we get a Show object to use as a working copy, we want to make sure that the show object actually contains the related ShowGenres and Credits that could be edited in the same session. For this reason, we want to add two collection properties to the Show class, a ShowGenres collection and a Credits collection. Here is how we can do that:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.Domain
{
    public class Show
    {
        #region Constructor
```

```
        #endregion

        #region Fields

        private int _id;
        private string _title;
        private int? _lengthInMinutes;
        private int?  mpaaRatingId;
        private List<ShowGenre> _showGenres = new List<ShowGenre>();
        private List<Credit> _credits = new List<Credit>();

        #endregion

        #region Properties

        public int Id
        {
            get { return _id; }
            set { _id = value; }
        }

        public string Title
        {
            get { return _title; }
            set { _title = value; }
        }

        public int? LengthInMinutes
        {
            get { return _lengthInMinutes; }
            set { _lengthInMinutes = value; }
        }

        public int? MpaaRatingId
        {
            get { return _mpaaRatingId; }
            set { _mpaaRatingId = value; }
        }

        public List<ShowGenre> ShowGenres
        {
            get { return _showGenres; }
        }

        public List<Credit> Credits
        {
            get { return _credits; }
        }

        #endregion

        #region Overrides

        public override string ToString()
        {
            return Title;
        }

        #endregion
    }
}
```

Note that the collection objects are instantiated when a Show instance is constructed and each of these collection objects is “read-only” to consumers of the class, so consumers cannot add or delete collection objects, but they can still get a read-only reference to either collection and add or delete objects from the collections.

With this in mind, let's survey the object model and see which objects are our root objects for editing:

- HairColor and EyeColor will not have screens for editing – they will just be implemented as a permanent set of choices, so they are not aggregate roots. If we later decided to add an administrative screen for editing hair and eye color options, then they would become roots.
- MpaaRating, Genre, and CreditType are simple roots where the editable object graph just consists of the object itself, along with its scalar properties.
- Show and Person are aggregate roots. The object graph for a Show is the Show plus a child collection of ShowGenres and a child collection of Credits. The object graph for a Person is the Person plus a child collection of Credits and a collection of PersonAttachments.
- ShowGenre, Credit and PersonAttachment are not aggregate roots, since they are not edited as the root of a transaction, but only as child object in the object graphs for Show or Person.

So the only thing we need to take care of is making sure that the Person object has child Collection properties for the Credits and PersonAttachments like this:

```
using System;

.
.
.

namespace Talent.Domain
{
    /// <summary>
    /// A Person object representing an Actor, Director, etc.
    /// </summary>
    public class Person
    {

        #region Fields

        private int _id;
         .
         .
         .
        private List<Credit> _credits = new List<Credit>();
        private List<PersonAttachment> _personAttachments = new List<PersonAttachment>();

        #endregion

        #region Properties

        public int Id
        {
            get { return _id; }
            set { _id = value; }
        }

         .
         .
         .

        public List<Credit> Credits
```

```
        {
            get { return _credits; }
        }

        public List<PersonAttachment> PersonAttachments
        {
            get { return _personAttachments;  }
        }

        #endregion

        #region Computed Properties

         .
         .
         .

        #endregion

        #region Overrides

        public override string ToString()
        {
            return FirstLastName;
        }

        #endregion
    }
}
```

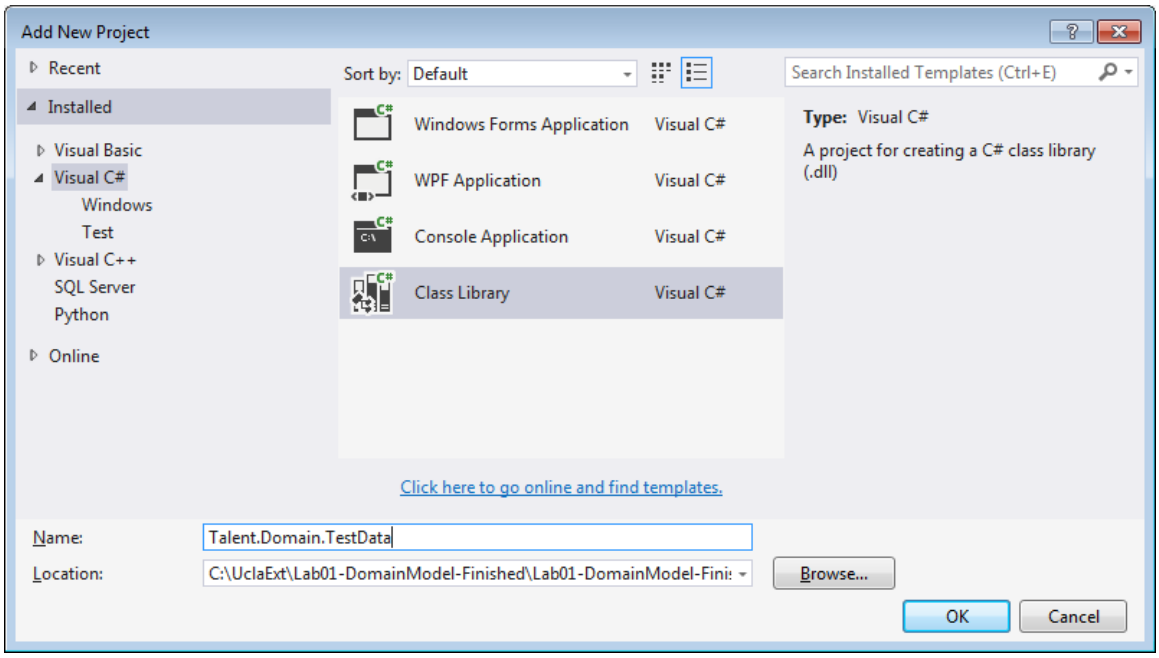
There area few significant implications of adding the child collection properties to the Show and Person classed:

- The Show and Person objects now contain all the objects in their editable object graph.
- These properties allow easier navigation to the child objects, and
- We must remember when building a Show or Person object to populate these collections.

You should make sure you can compile the solution again.

Step 14: Create a TestData assembly. We now have a class library that allows us to represent objects within the Talent application's subject area. In this lab, we will put together a simple console application to exercise domain model objects, so we will want to have a place to put collections of objects to work with which I'll call a DomainObjectStore. We could create this object store in the console application, but we want to be able to use it later with different executable assemblies, so we'll create a new project called Talent.Domain.TestData for the object store.

First add a new project to the solution using the Class Library template, and call it Talent.Domain.TestData.



Delete Class1 from this new project, and add a project reference to the Talent.Domain project.

Then add a new class called DomainObjectStore. The DomainObjectStore will serve as a place to keep collections of inter-related domain objects for testing purposes, so we will start by adding a collection property for each type of domain object, named with the plural variant, using the automatic property syntax:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.Domain.TestData
{
    public class DomainObjectStore
    {
        #region Constructor

        public DomainObjectStore()
        {

        }

        #endregion

        #region Properties

        public List<MpaaRating> MpaaRatings { get; set; }
        public List<Genre> Genres { get; set; }
        public List<CreditType> CreditTypes { get; set; }
        public List<HairColor> HairColors { get; set; }
        public List<EyeColor> EyeColors { get; set; }
        public List<Show> Shows { get; set; }
        public List<Person> People { get; set; }
        public List<ShowGenre> ShowGenres { get; set; }
        public List<Credit> Credits { get; set; }

        #endregion
```

```
    }
}
```

Step 15: Generate some Test MpaaRatings data. To get started, let's add some code to the Genres collection of the DomainObjectStore, and then create a Console Application to work with the store.

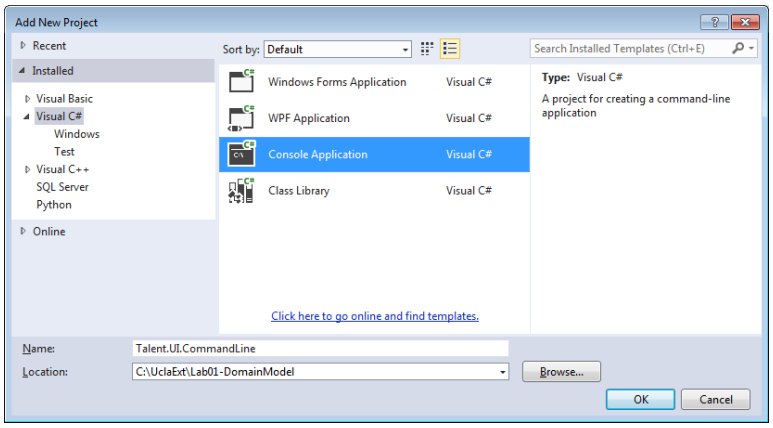
To keep things organized, create a parameterless constructor to the DomainObjectStore, and have it populate the MpaaRatings collection by calling a private static property LoadMpaaRatings, which returns a Generic List of MpaaRatings:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.Domain.TestData
{
    public class DomainObjectStore
    {
        #region Constructor

        public DomainObjectStore()
        {
            MpaaRatings = LoadMpaaRatings();
        }

        #endregion

        #region Properties

        public List<MpaaRating> MpaaRatings { get; set; }
        public List<Genre> Genres { get; set; }
        public List<CreditType> CreditTypes { get; set; }
        public List<HairColor> HairColors { get; set; }
        public List<EyeColor> EyeColors { get; set; }
        public List<Show> Shows { get; set; }
        public List<Person> People { get; set; }
        public List<ShowGenre> ShowGenres { get; set; }
        public List<Credit> Credits { get; set; }

        #endregion

        #region Load Methods

        private static List<MpaaRating> LoadMpaaRatings()
        {
            var newId = 0;

            List<MpaaRating> list = new List<MpaaRating>();
            list.Add(
                new MpaaRating()
                {
                    Id = ++ newId,
                    Code = "G",
                    Name = "General Audiences",
                    Description = "All ages admitted.",
                    DisplayOrder = 10
                }
                );
            list.Add(
                new MpaaRating()
                {
                    Id = ++ newId,
                    Code = "PG",
                    Name = "Parental Guidance Suggested",
```

```
                Description = "Some material may not be suitable for children.",
                DisplayOrder = 20
            }
            );
        list.Add(
            new MpaaRating()
            {
                Id = ++ newId,
                Code = "PG-13",
                Name = "Parents Are Strongly Cautioned",
                Description = "Parents are urged to be cautious.",
                DisplayOrder = 30
            }
            );
        list.Add(
            new MpaaRating()
            {
                Id = ++ newId,
                Code = "R",
                Name = "Restricted",
                Description = "People under 17 years must be accompanied.",
                DisplayOrder = 40
            }
            );
        list.Add(
            new MpaaRating()
            {
                Id = ++ newId,
                Code = "NC-17",
                Name = "Adults Only",
                Description = "Exclusively for adults.",
                DisplayOrder = 50
            }
            );
        list.Add(
            new MpaaRating()
            {
                Id = ++ newId,
                Code = "UR",
                Name = "Unrated",
                Description = "Not submitted for rating.",
                DisplayOrder = 60
            }
            );
        return list;
    }

    #endregion
  }
}
```

At this point, we should be able to create an instance of the DomainObjectStore, and it will automatically have the MpaaRatings collection property populated, so let's give it a try by adding a Console application to the solution.

Add a new Console application to the solution called Talent.UI.CommandLine:



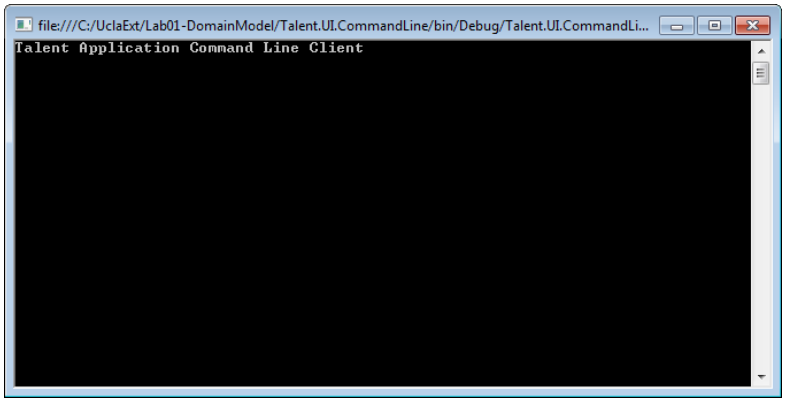
You will need to add project references in the Talent.UI.CommandLine project that reference the Talent.Domain and Talent.Domain.TestData projects. This is what allows the console application access to the public classes within the Talent.Domain project and the DomainObjectStore test data.

As you should know, a console application template creates an executable program with a command-line style interface. The template sets up a Program.cs class definition that contains a static Main method where the CLR begins execution. We can communicate with the application via the command-line console using various methods defined on the static Console class.

Start by adding the Console.WriteLine and Console.ReadLine method calls like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Talent.UI.CommandLine
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Talent Application Command Line Client");

            Console.ReadLine();
        }
    }
}
```

Be sure to set the Talent.UI.CommandLine project as the start-up project for the solution, then compile and run the solution. You should see the awe-inspiring command-line screen:



And pressing the <Enter> will close the application. The Console.ReadLine() method causes the application to wait for a key press before exiting the main method – otherwise the application would display the screen for an instant, but then immediately exit before you had a chance to look at the console window.

To exercise the DomainObjectStore, just add code to create an instance of the DomainObjectStore, and then loop over the store's MpaaRatings collection to dump information about each MpaaRating to the console:

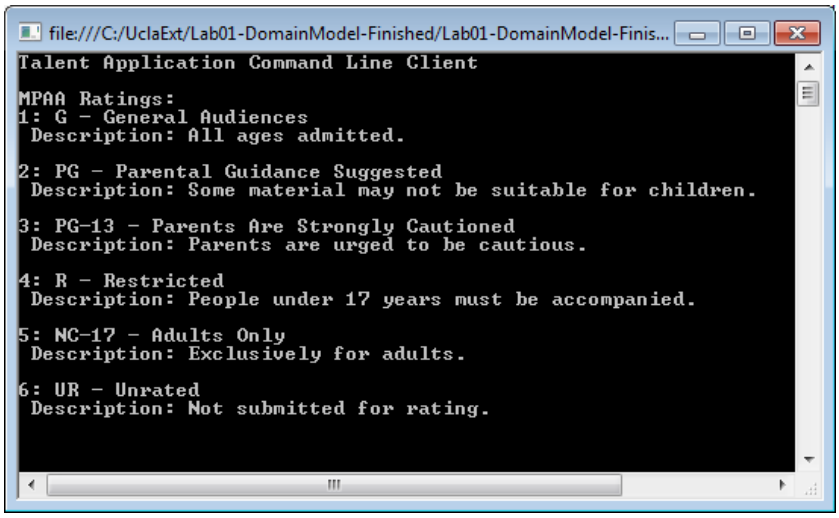
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Talent.Domain;
using Talent.Domain.TestData;

namespace Talent.UI.CommandLine
{
    class Program
    {

        #region Main

        static void Main(string[] args)
        {
            Console.WriteLine("Talent Application Command Line Client");

            DomainObjectStore store = new DomainObjectStore();

            Console.WriteLine("\r\nMPAA Ratings:");
            foreach (var m in store.MpaaRatings)
            {
                Console.WriteLine(String.Format(
                    "{0}: {1} - {2}\r\n Description: {3}\r\n",
                    m.Id, m.Code, m.Name, m.Description));
            }

            Console.ReadLine();
        }

        #endregion

    }
}
```

Tip: If you omit the using Talent.Domain.TestData using statement, you will notice that the DomainObjectStore class references have a squiggly line under them. If you hover the mouse over any of these references, VS will display a message saying “The type or namespace name “DomainObjectStore” could not be found (are you missing a using directive or an assembly reference?)”. Since we already added the assembly reference to the CommandLine project, what we are missing is the using directive. You can add the missing using directive to the list of using directives at the beginning of the Program.cs file like this:

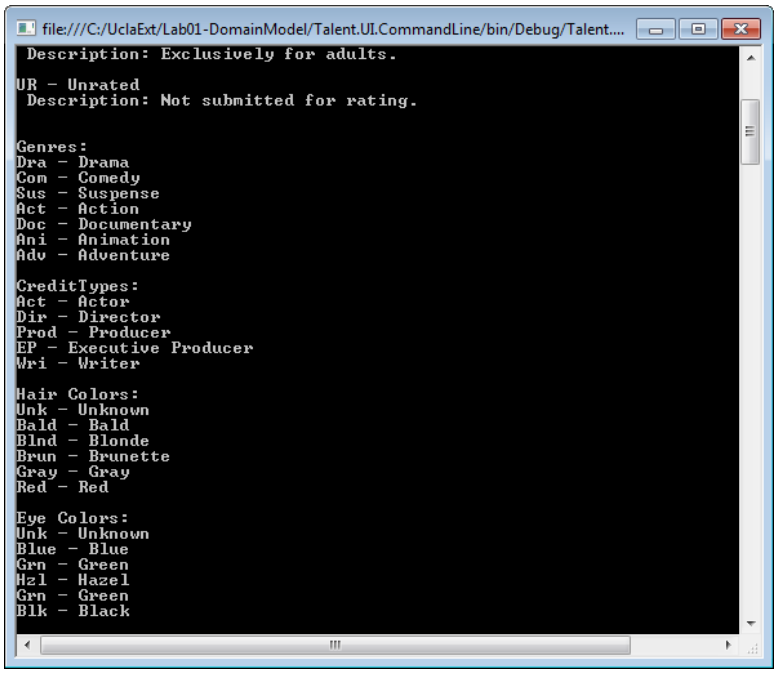
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Talent.Domain.TestData;

namespace Talent.UI.CommandLine
…
```

Or, a take advantage of a nice VS feature by double-clicking on one occurrence of the DomainObjectStore class name and pressing Ctrl-. which will pop-up a context menu, and the first choice will say “using Talent.Domain.TestData” and selecting that item will add the using statement for you. This is a great feature that will save you a lot of time during development.

Now if you run the application, you should get a list of the MPAA ratings:



Step 16: Following the same pattern, see if you can create collections of Genres, CreditType, HairColor and EyeColor objects and list them such that the output is like this (just the output from “Genres:” onward is new):



To do this, you will add a new Load method for each domain class to the DomainObjectStore, and call that method from the constructor. Then you just add a loop to the console application to dump the results to the console.

Step 17: Creating Shows. We follow a similar pattern to create a collection of Shows. First create a List<Show> collection to hold the Show objects, and then instantiate and initialize a few shows:

```
private static List<Show> LoadShows()
{
    var maxId = 0;
    var list = new List<Show>();
    list.Add(
        new Show()
        {
            Id = ++maxId, // 1
            Title = "Philomena",
            LengthInMinutes = 98,
            MpaaRatingId = 3,
            TheatricalReleaseDate = new DateTime(2013, 11, 27)
        });

    list.Add(
        new Show()
        {
            Id = ++maxId, // 2
            Title = "Frozen",
            LengthInMinutes = 102,
            MpaaRatingId = 2,
            TheatricalReleaseDate = new DateTime(2013, 11, 27)
        });

    return list;
}
```

Notice that I populated the MpaaRatingId values based on the MpaaRatings that were loaded earlier.

Add a line to call this method from the DomainObjectStore constructor.

NOTE: Steps 18-21 get pretty tedious and are purely optional. I included them only for the sake of completeness and they really don't add much to the conceptual content of the lab.

Step 18: Creating People. Then create a LoadPeople method that creates a collection of people:

```
private static List<Person> LoadPeople()
{
    var maxId = 0;


    List<Person> list = new List<Person>();
    list.Add(
        new Person
        {
            Id = ++maxId, // 1
            Salutation = "Ms.",
            FirstName = "Judith",
            MiddleName = "Olivia",
            LastName = "Dench",
            Suffix = "",
            DateOfBirth = new DateTime(1934, 12, 9),
            HairColorId = 5,
            EyeColorId = 2,
            Height = 66
        });

    list.Add(
        new Person
        {
            Id = ++maxId, // 2
            Salutation = "Mr.",
            FirstName = "Stephen",
            MiddleName = "",
            LastName = "Frears",
            Suffix = "",
            DateOfBirth = new DateTime(1941, 6, 20)
        });


    list.Add(
        new Person
        {
            Id = ++maxId, // 3
            Salutation = "Mr.",
            FirstName = "Stephen",
            MiddleName = "John",
            LastName = "Coogan",
            Suffix = "",
            DateOfBirth = new DateTime(1965, 10, 14),
            HairColorId = 4,
            EyeColorId = 5,
            Weight = 185,
            Height = 72
        });

    list.Add(
        new Person
        {
```

```
            Id = ++maxId, // 4
            Salutation = "Ms.",
            FirstName = "Sophie",
            MiddleName = "Kennedy",
            LastName = "Clark",
            Suffix = "",
            DateOfBirth = null,
            HairColorId = 3,
            EyeColorId = 3,
            Weight = 115,
            Height = 68
        });

    list.Add(
        new Person
        {
            Id = ++maxId, // 5
            Salutation = "Mr.",
            FirstName = "Chris",
            MiddleName = "",
            LastName = "Buck",
            Suffix = "",
            DateOfBirth = null,
            HairColorId = 4,
            EyeColorId = 5,
            Weight = 175,
            Height = 68
        });

    list.Add(
        new Person
        {
            Id = ++maxId, // 6
            Salutation = "Ms.",
            FirstName = "Jennifer",
            MiddleName = "",
            LastName = "Lee",
            Suffix = "",
            DateOfBirth = null,
            HairColorId = 3,
            EyeColorId = 5
        });

    list.Add(
        new Person
        {
            Id = ++maxId, // 7
            Salutation = "Ms.",
            FirstName = "Kristen",
            MiddleName = "",
            LastName = "Bell",
            Suffix = "",
            DateOfBirth = new DateTime(1980, 7, 18),
            HairColorId = 4,
            EyeColorId = 6
        });

    list.Add(
        new Person
        {
            Id = ++maxId, // 8
            Salutation = "",
            FirstName = "Josh",
            MiddleName = "",
            LastName = "Gad",
            Suffix = "",
            DateOfBirth = null,
            HairColorId = 1,
            EyeColorId = 1
        });
```

```
    return list;
}
```

Add a line to call this method from the DomainObjectStore constructor.

For People, I can populate the link values to HairColor and EyeColor, but I am not yet able to populate the Credits collection, since the Credits are not yet created. We will need to create the Credits later. Likewise, I do not have ShowGenre objects yet to populate the ShowGenres collection.

Step 19: Creating ShowGenres. To populate the ShowGenres, I need to have created the Shows and People referenced first, so I can set the ShowId and GenreId values when instantiating the ShowGenre objects. After creating a ShowGenre object, I need to make sure to also add it to the corresponding Show object's ShowGenres collection:

```
private static List<ShowGenre> LoadShowGenres(List<Show> shows)
{
    var maxShowGenreId = 0;

    var list = new List<ShowGenre>();
    ShowGenre showGenre;

    showGenre = new ShowGenre
    {
        Id = ++maxShowGenreId,
        ShowId = 1,
        GenreId = 1
    };
    list.Add(showGenre);
    FindShowById(shows, showGenre.ShowId).ShowGenres.Add(showGenre);

    showGenre = new ShowGenre
    {
        Id = ++maxShowGenreId,
        ShowId = 2,
        GenreId = 2
    };
    list.Add(showGenre);
    FindShowById(shows, showGenre.ShowId).ShowGenres.Add(showGenre);

    showGenre = new ShowGenre
    {
        Id = ++maxShowGenreId,
        ShowId = 2,
        GenreId = 6
    };
    list.Add(showGenre);
    FindShowById(shows, showGenre.ShowId).ShowGenres.Add(showGenre);

    showGenre = new ShowGenre
    {
        Id = ++maxShowGenreId,
        ShowId = 2,
        GenreId = 7
    };
    FindShowById(shows, showGenre.ShowId).ShowGenres.Add(showGenre);

    list.Add(showGenre);
    return list;
}
```

Where the FindShowById helper method is defined like this:

```
public static Show FindShowById(List<Show> shows, int id)
{
    foreach (var c in shows)
    {
        if (c.Id == id) return c;
    }
    return null;
}
```

(I created a region in the DomainObjectStore class called “Helper Methods” for this method and others like it that we will create shortly.) Since Loading the ShowGenres depends on having a reference to the list of shows, this Load method is passed the list of shows as an argument.

Of course, you will need to call this method from the DomainObjectStore’s constructor.

Step 20: Populating Credits. To populate Credits, I need to already have created the referenced Shows, People and CreditTypes. For each CreditType, I also need to remember to add the new Credit to the People.Credits and Show.Credits collections for the corresponding people and shows to complete both of those objects.

```
private static List<Credit> LoadCredits(List<Show> shows, List<Person> people)
{
    var maxId = 0;
    var list = new List<Credit>();
    Credit crd;
    crd = new Credit
        {
            Id = ++maxId,
            ShowId = 1,
            PersonId = 1,
            CreditTypeId = 1,
            Character = "Philomena"
        };
    list.Add(crd);
    FindShowById(shows, crd.ShowId).Credits.Add(crd);
    FindPersonById(people, crd.PersonId).Credits.Add(crd);

    crd = new Credit
        {
            Id = ++maxId,
            ShowId = 1,
            PersonId = 2,
            CreditTypeId = 2
        };
    list.Add(crd);
    FindShowById(shows, crd.ShowId).Credits.Add(crd);
    FindPersonById(people, crd.PersonId).Credits.Add(crd);

    crd = new Credit
        {
            Id = ++maxId,
            ShowId = 1,
            PersonId = 3,
            CreditTypeId = 5
        };
    list.Add(crd);
    FindShowById(shows, crd.ShowId).Credits.Add(crd);
    FindPersonById(people, crd.PersonId).Credits.Add(crd);

    crd = new Credit
```

```
    {
        Id = ++maxId,
        ShowId = 1,
        PersonId = 3,
        CreditTypeId = 1,
        Character = "Martin Sixsmith"
    };
list.Add(crd);
FindShowById(shows, crd.ShowId).Credits.Add(crd);
FindPersonById(people, crd.PersonId).Credits.Add(crd);

crd = new Credit
    {
        Id = ++maxId,
        ShowId = 1,
        PersonId = 4,
        CreditTypeId = 1,
        Character = "Young Philomena"
    };
list.Add(crd);
FindShowById(shows, crd.ShowId).Credits.Add(crd);
FindPersonById(people, crd.PersonId).Credits.Add(crd);

crd = new Credit
    {
        Id = ++maxId,
        ShowId = 1,
        PersonId = 5,
        CreditTypeId = 2
    };
list.Add(crd);
FindShowById(shows, crd.ShowId).Credits.Add(crd);
FindPersonById(people, crd.PersonId).Credits.Add(crd);

crd = new Credit
    {
        Id = ++maxId,
        ShowId = 2,
        PersonId = 6,
        CreditTypeId = 2
    };
list.Add(crd);
FindShowById(shows, crd.ShowId).Credits.Add(crd);
FindPersonById(people, crd.PersonId).Credits.Add(crd);

crd = new Credit
    {
        Id = ++maxId,
        ShowId = 2,
        PersonId = 6,
        CreditTypeId = 5
    };
list.Add(crd);
FindShowById(shows, crd.ShowId).Credits.Add(crd);
FindPersonById(people, crd.PersonId).Credits.Add(crd);

crd = new Credit
    {
        Id = ++maxId,
        ShowId = 2,
        PersonId = 7,
        CreditTypeId = 1,
        Character = "Anna"
    };
list.Add(crd);
FindShowById(shows, crd.ShowId).Credits.Add(crd);
FindPersonById(people, crd.PersonId).Credits.Add(crd);

crd = new Credit
    {
        Id = ++maxId,
```

```
            ShowId = 2,
            PersonId = 8,
            CreditTypeId = 1,
            Character = "Olaf"
        };
    list.Add(crd);
    FindShowById(shows, crd.ShowId).Credits.Add(crd);
    FindPersonById(people, crd.PersonId).Credits.Add(crd);

    return list;
}
```

This required adding a FindPersonById method like this:

```
public static Person FindPersonById(List<Person> people, int id)
{
    foreach (var c in people)
    {
        if (c. Id == id) return c;
    }
    return null;
}
```

Since Loading the Credits depends on having access to the collection of Shows and People, this Load method is passed the list of shows and the list of People as an argument.

Of course, you will need to call this method from the DomainObjectStore's constructor.

Now we have an object model populated with sample objects, complete with links that define the relationships we will need to navigate around the model.

Step 21: Listing results to the Console. Now we can add code to the Main method to display information about all the domain object collections in the DomainObjectStore:

```
static void Main(string[] args)
{
    Console.WriteLine("Talent Application Command Line Client");

    DomainObjectStore store = new DomainObjectStore();

    Console.WriteLine("\r\nMPAA Ratings:");
    foreach (var m in store.MpaaRatings)
    {
        Console.WriteLine(String.Format(
            "{0}: {1} - {2}\r\n Description: {3}\r\n",
            m.Id, m.Code, m.Name, m.Description));
    }

    Console.WriteLine("\r\nGenres:");
    foreach (var g in store.Genres)
    {
        Console.WriteLine(String.Format("{0}: {1} - {2}",
            g.Id, g.Code, g.Name));
    }

    Console.WriteLine("\r\nCreditTypes:");
    foreach (var g in store.CreditTypes)
    {
        Console.WriteLine(String.Format("{0}: {1} - {2}",
            g.Id, g.Code, g.Name));
    }
```

```
    Console.WriteLine("\r\nHair Colors:");
    foreach (var g in store.HairColors)
    {
        Console.WriteLine(String.Format("{0}: {1} - {2}",
            g.Id, g.Code, g.Name));
    }

    Console.WriteLine("\r\nEye Colors:");
    foreach (var g in store.EyeColors)
    {
        Console.WriteLine(String.Format("{0}: {1} - {2}",
            g.Id, g.Code, g.Name));
    }

    Console.WriteLine("\r\nShows:");
    foreach (var show in store.Shows)
    {
        Console.WriteLine(String.Format("{0}: {1} - {2}",
            show.Id, show.Title,
            show.MpaaRatingId.HasValue ?
                DomainObjectStore.FindMpaaRatingById(
                    store.MpaaRatings,
                    show.MpaaRatingId.Value).Code : ""));
        Console.Write("\tGenres: ");
        foreach (var sg in show.ShowGenres)
        {
            Console.Write(DomainObjectStore.FindGenreById(
                store.Genres, sg.GenreId).Name + " ");
        }
        Console.Write("\r\n");
        foreach (var cr in show.Credits)
        {
            Console.WriteLine("\t"
                + DomainObjectStore.FindCreditTypeById(
                    store.CreditTypes, cr.CreditTypeId).Name + " "
                + DomainObjectStore.FindPersonById(
                    store.People, cr.PersonId).FullName + " "
                + cr.Character);
        }
    }

    Console.WriteLine("\r\nPeople:");
    foreach (var person in store.People)
    {
        Console.WriteLine(String.Format("{0} - {1}",
            person.FullName, person.Age));
        foreach (var cr in person.Credits)
        {
            Console.WriteLine("\t"
                + DomainObjectStore.FindCreditTypeById(
                    store.CreditTypes, cr.CreditTypeId).Name + " "
                + DomainObjectStore.FindShowById(
                    store.Shows, cr.ShowId).Title + " "
                + cr.Character);
        }
    }

    Console.ReadLine();
}
```

In order to get detailed information about the various domain objects referenced by each Show and Person, I needed a bunch of trivial helper methods to get the object by their Id, so I collected them all together with the previously created FindShowById and FindPersonId into the #Helper method region within the DomainObjectStore class, so the Helper class region of the DomainObjectStore class definition is now:

```
#region Helper Methods

public static MpaaRating FindMpaaRatingById(List<MpaaRating> ratings, int id)
{
    foreach (var r in ratings)
    {
        if (r.Id == id) return r;
    }
    return null;
}

public static Genre FindGenreById(List<Genre> genres, int id)
{
    foreach (var g in genres)
    {
        if (g.Id == id) return g;
    }
    return null;
}

public static CreditType FindCreditTypeById(List<CreditType> creditTypes, int id)
{
    foreach (var c in creditTypes)
    {
        if (c.Id == id) return c;
    }
    return null;
}

public static HairColor FindHairColorById(List<HairColor> hairColors, int id)
{
    foreach (var c in hairColors)
    {
        if (c.Id == id) return c;
    }
    return null;
}

public static EyeColor FindEyeColorById(List<EyeColor> EyeColors, int id)
{
    foreach (var c in EyeColors)
    {
        if (c.Id == id) return c;
    }
    return null;
}

public static Show FindShowById(List<Show> shows, int id)
{
    foreach (var c in shows)
    {
        if (c.Id == id) return c;
    }
    return null;
}

public static Person FindPersonById(List<Person> people, int id)
{
    foreach (var c in people)
    {
        if (c.Id == id) return c;
    }
    return null;
}

#endregion
```

These are all pretty trivial, and we'll see in upcoming lessons how we could replace them with very concise lambda expressions, but this exercise probably already has enough new concepts for you to digest.

You should now be able to run this and get an output that shows a reasonable listing of data from the populated data model.

What Have We Done?

At this point, we have constructed a domain model that allows us to represent objects in the business domain using purely object-oriented techniques. Here are the things we have accomplished:

- We outlined the problem domain of the application, identifying the important concepts and use cases from the problem space. An important by-product of this analysis is an understanding of the language that users are accustomed to using, which will facilitate communication about the system design as development proceeds.
- We created classes representing each of the domain concepts as relatively simple C# class definitions with properties representing the important attributes of each domain object, plus additional properties that establish the relationships among objects.
- We created a separate Talent.Domain.TestData that contains a DomainObjectStore class, representing an in-memory database for exercising the domain model with test data. When instantiated, the DomainObjectStore populates itself with sample data for development and testing.
- We created an experimental console application that instantiates a DomainObjectStore and dumps the sample data to the console as a simple test that the Domain model seems to be working properly. We will see how to create a better Unit Test environment in the Unit Testing Lab in a later course module.
- In the course of all this, we reviewed the applicable concepts from C# including class definition, some of the common data types, automatic property definition, read-only properties, and overriding the ToString() method. We also started building a multi-project solution that separated the domain model from the User Interface.
- We thought about how the domain object model would be used to implement the editing operations required for the use cases, and made some additions to the Show and Person objects by adding collection properties that will allow them to be used as self-contained object graphs to support the editing operations.

With a domain-centric architecture, this domain model forms the core of the application, and we will build upon this basic model throughout the course, making various enhancements to the domain model structure as we go, but striving to keep the Talent.Domain project focused on the essence of the problem space as much as possible by making an effort to keep code that is really related to specific data access or user interface technologies out of the domain model.

What's Next?

There are a few concepts in this exercise that we used without much explanation, so in Lab 2, we will look more closely at property definitions and generics.

In this lab, we just created what amounts to a "throw-away" console application to exercise just a few features of the domain model. This is a pretty lame approach to testing an application, so we want to look at how we can create a testable application and build up a set of unit tests that can be executed and analyzed easily at any time to verify that each part of the application works as intended. In Lab 4, we will take a closer look at creating Unit Tests that can exercise the domain model more thoroughly.

Once we are happy that the Domain Model is solid, we will develop:

- A User interface using Windows Presentation Foundation (WPF), and
- A database design and Data Access Layer components for our WPF application to interact with the database, so the domain objects are persisted and shared in a Microsoft SQL Server database.

References

[DDD] Domain-Driven Design. For a brief description of the idea, check the Wikipedia page on Domain-Driven Design (http://en.wikipedia.org/wiki/Domain-driven_design). The seminal work on the topic is the book **Domain Driven Design – Tackling Complexity in the Heart of Software** by Eric Evans, 2004, which is still the best treatment of this excellent approach to software development for non-trivial projects.

Finished Solution

My final solution to this exercise is available from the GitHub repository at: https://github.com/entrotech/Lab01-DomainModel.