# Introduction

## Problem

In this lab, you will expand your existing Windows Forms movie collection program to support multiple movies. Unless otherwise stated, all previous behavior and functionality remains the same.

## Solution

You will not be adding any new forms for this lab but you will be changing the behavior of the existing code.

Note: All screenshots are for demonstration purposes. You can make adjustments to meet your needs.

# Setting Up the Solution

As with previous labs, copy the entire solution (folder and all) into a new folder (Lab3). This will allow you to reuse all your existing code but still keep it separate from previous lab assignments.

# Movie Database

Now that there are multiple movies to support, a single field in the form is no longer sufficient. The lab will eventually use a database so now is a good time to start setting up the application to use one. For now the “database” will be stored in memory.

## IMovieDatabase Interface

Add a new interface called IMovieDatabase to the business project. This interface will act as the abstraction to the underlying database that is used.

The implementation will need to keep track of the movies that are added to it. Each movie should be given a unique ID to represent it. The ID can be used to retrieve a movie later. No two movies should have the same ID. The ID must be greater than 0.

Ensure the interface is properly documented using doctags.

The interface will have the following members.

### Add

Adds a movie to the database. The implementation will need to keep track of which movies have been added to it. Each added movie should be given a unique ID.

#### Parameters

* The movie to add to the database.

#### Returns

The newly added movie if it is added successfully. The movie’s ID should be set to the correct value.

If the method fails then return null.

#### Validation

The following validation rules should be enforced. If any validation fails then return null.

* The movie parameter is not null.
* The argument provided for the movie is valid.
* A movie with the same name does not already exist. Case does not matter.

### Get

Gets a specific movie from the database.

#### Parameters

* The unique ID of the movie.

#### Returns

The movie with the given ID. If no movie exists then return null.

To better emulate the behavior of a database and to ensure “database” objects are not accidentally modified, return a copy of the movie.

#### Validation

The following validation rules should be enforced. If any validation fails then return null.

* The ID must be greater than 0.

### GetAll

Gets all the movies from the database.

#### Parameters

* None.

#### Returns

An enumerable list of movies. To better emulate the behavior of a database and to ensure “database” objects are not accidentally modified, return a copy of the movie.

#### Validation

None.

### Remove

Removes the specified movie from the database. If the movie does not exist then nothing happens.

#### Parameters

* The unique ID of the movie.

#### Returns

True if the movie is removed or false if the movie was not removed.

#### Validation

The following validation rules should be enforced. If any validation fails then return null.

* The ID must be greater than 0.

### Update

Updates a movie in the database.

#### Parameters

* The updated movie information.

#### Returns

The updated movie.

#### Validation

The following validation rules should be enforced. If any validation fails then return null.

* The movie parameter is not null.
* The argument provided for the movie is valid.
* A movie with the same name does not already exist, except the movie itself. Case does not matter.
* The movie ID must be valid and exist.

# Movie Class

The following changes should be made to the Movie class.

The type should now implement the IValidatableObject interface. The validation should enforce the following rules.

* Name cannot be null or empty.
* Length must be greater than or equal to 0.

A new Id property should be added to uniquely identify the movie in the database.

# In Memory Database

In order to use the interface in code it is necessary to create a class that implements it. The purpose of an interface is to separate the contract from the implementation so it is generally a good idea to keep them separated. We will create a separate project to store the first implementation, an in-memory database.

Create a new Class Library project in the solution called MovieLib.Data.Memory. The project will need a reference to the MovieLib project.

Create a new class (ex. MemoryMovieDatabase) to implement IMovieDatabase using an in-memory collection. Implement each of the methods following the rules of the interface. Ensure the class is properly documented using doctags.

Some guidelines about the implementation.

* In a real database, changes made to the movie instance after add/update calls does not impact the version stored in the database. Nor does making changes to an instance returned from the database. The memory implementation should “clone” the data that is stored in the database and never return object directly.
* The implementation will need to assign a unique identifier to each movie added to the list. Do not reuse IDs. The IDs do not have to be sequential but a simple counter would be sufficient.
* Private helper methods will be useful in some of the core functionality to allow reuse in the implementation of the code.
* Do not overthink this implementation. It should be no longer than a couple hundred lines, including comments.

# Main Form

The main form will be updated to use the IMovieDatabase to manage the movies. The main form will display the list of movies in the main window area.

## Using the Database

Replace the existing field that was used to store a single movie with a field to hold the movie interface. In a commercial application the interface implementation would be dynamically injected into the code but for our purposes the form will set the field to an instance of the memory database class. Do not use the memory database type anywhere else in the code, only the interface. You will need to add a reference to the implementation project.

# Displaying the Movies

The main area of the form will show a grid of movies along with basic movie information. The user will be able to select a movie in the grid to edit or delete. Add, edit and delete of movies will update the grid accordingly. The movies will be shown sorted by title. The grid will not support inline editing.

## Creating a Binding Source

A binding source is used to bind the business objects to the UI.

* Add a BindingSource to the form.
* Set the DataSource to the Movie type. (The code must already compile for this to work.)

## Adding the Grid

A grid will be used to render the binding source data in the UI.

* Add a DataGridView to the form.
* Configure the grid.
  + Use the Dock property to fill the grid to the parent.
  + Using the AlternatingRowsDefaultCellStyle property, give alternating rows a gray background.
  + Use the RowHeadersVisible property to hide row headers.
  + Use The AllowUserTo properties to disable add and delete of rows and to resize rows.
  + Use the EditMode property to disable editing from the grid.
  + Use the SelectionMode property to allow the user to select a row by clicking anywhere in the row.
* Set the DataSource of the grid to the BindingSource created earlier.
* Adjust the column properties to display in the following order.
  + Title
    - Freeze the column so it is always visible.
    - Set the FillWeight so the column takes up about a third of the initial space.
    - Set the MinimumWidth to 100.
  + Description
    - Set the AutoSizeMode so it fills the remaining space.
    - Set the FillWeight so the column takes up about a third of the initial space.
    - Set the MinimumWidth to 100.
  + Length
    - Set the FillWeight so the column takes up about a third of the initial space minus the space for the IsOwned column.
    - Set the MinimumWidth to 50.
  + IsOwned
    - Do not allow resizing.
    - Should take up enough space to show the header and checkbox.
    - Set the header to Is Owned?
  + Id
    - Hide the column (do not remove it)

## Loading the Grid

When the form is displayed, or whenever a change is made to a movie, the grid needs to be bound to the movies from the database.

* Get the movies from the database.
* Suspend the binding on the BindingSource.
* Set the DataSource property of the BindingSource to the movies.
* Resume the binding on the BindingSource.

## Implementing Selection

Create a method to get the selected movie from the grid. It is needed for several of the commands. The BindingSource tracks the currently selected item in the UI using the Current property.

## Implementing Edit

The grid will not support inline editing. Instead if the user double clicks a row then the Movie \ Edit command will be used.

* Handle the CellDoubleClick event.
* Ensure a movie row was clicked and not the column header.
* If a movie was clicked then call the Movie \ Edit handler.

For keyboard navigation, if the user presses ENTER while a movie is selected then the Movie \ Edit command will be used.

* Handle the KeyDown event.
* If a movie is selected and the key pressed was ENTER then edit the selected movie. Be sure to suppress the key press if a movie is edited.

## Implementing Delete

If the user presses DEL while a movie is selected then the Movie \ Remove command will be used.

* Update the KeyDown event to also detect the DEL key.
* If a movie is selected and the DEL key is pressed then delete the selected movie (using the normal handler logic from earlier). Be sure to suppress the key press if a movie is edited.

# Update Commands

In the previous labs we have worked with a single movie and therefore added code to detect whether a movie already existed and/or overwrote the existing movie. This is no longer true so the code must be updated.

## Movie \ Add

When adding a new movie, the new movie should be added to the database. It will no longer overwrite any movie that is already in the database. If the call is successful then update the UI with the new movie.

If the call to the database fails then display an error and return the user to the detail form to try again.

Note: For this command to work it is necessary to pass the database instance to the child form.

## Movie \ Edit

In order to edit a movie, it must be selected in the main form. Use the method created earlier. If there is no selected movie then display an error and do nothing.

When editing an existing movie, the movie should be updated in the database. If the call is successful then update the UI with the movie changes.

If the call to the database fails then display an error and return the user to the detail form to try again.

Note: For this command to work it is necessary to pass the database instance to the child form.

## Movie \ Remove

In order to remove a movie, it must be selected in the main form. Use the method created earlier. If there is no selected movie then display an error and do nothing.

When removing a movie, remove the movie from the database, after confirmation.

When removing the movie, if the call to the database fails then display an error. It is not an error if the remove returns false.

# Movie Detail Form

A few changes need to occur to the movie detail form to handle working with the database. The form will need access to the database, provided by the main form.

## Adding UI Validation

Update the UI to show validation errors before the user attempts to save any changes.

* Add an ErrorProvider to the form.
* Handle the Validating event for the Title field. The title is required.
* Handle the Validating event for the Length field. The length must be >= 0.
* Modify the form load logic to validate the form when it is first shown.
* Allow the user to tab out of a field even if it is invalid using the AutoValidate property.
* Do not validate the fields if the Cancel button is clicked using the CausesValidation property.

## Adding Save Validation

The Movie class now implements IValidatableObject and therefore can self-validate. The database will verify the data is invalid so it no longer needs to be handled by the save method.

* Remove any logic around save validation in the save method.
* Ensure the UI validation is successful before attempting to save.
* Add or update the movie in the database.
* If the database call fails then display an error and return the detail form so the user can fix the error(s).

# Requirements

Your program must meet all the following requirements.

* Compile cleanly with no warnings or errors.
* Meet all the points mentioned in the solution.
* Ensure each file has a file header indicating the course, your name and date.
* Solution, project and code should be uploaded to Github
* Submit lab in MyTCC by providing link to Github URL