GRAND CANYON UNIVERSITY™

CST-150 Activity 6 Guide

Contents

Part 1

Writing Classes

Overview

Create a Windows Form Application that reads from a text file that is provided by the instructor. This text file must be used when turning in this assignment. The contents of the text file will be inserted into a List using a Class Model. The list will then be bound to the data grid view. The purpose of this exercise is to use N-Layer architecture, with all the logic in the business layer class and form in the root directory.

Execution

Execute this assignment according to the following guidelines:

1. Start a new Visual Studio Project.
    a. Start Visual Studio and select "Create a new project" as is shown in Figure 1.

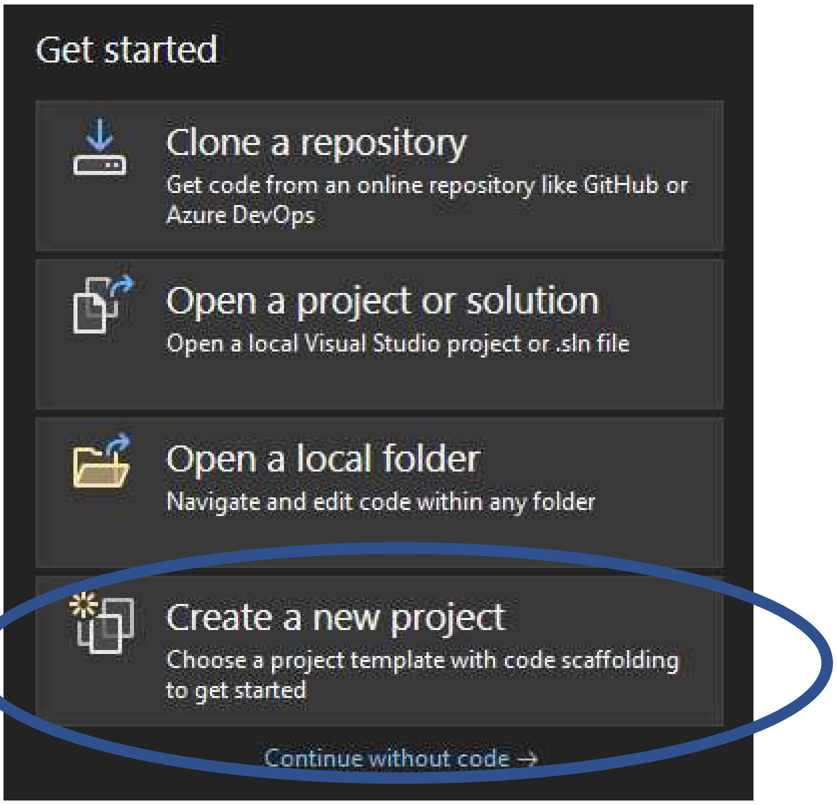
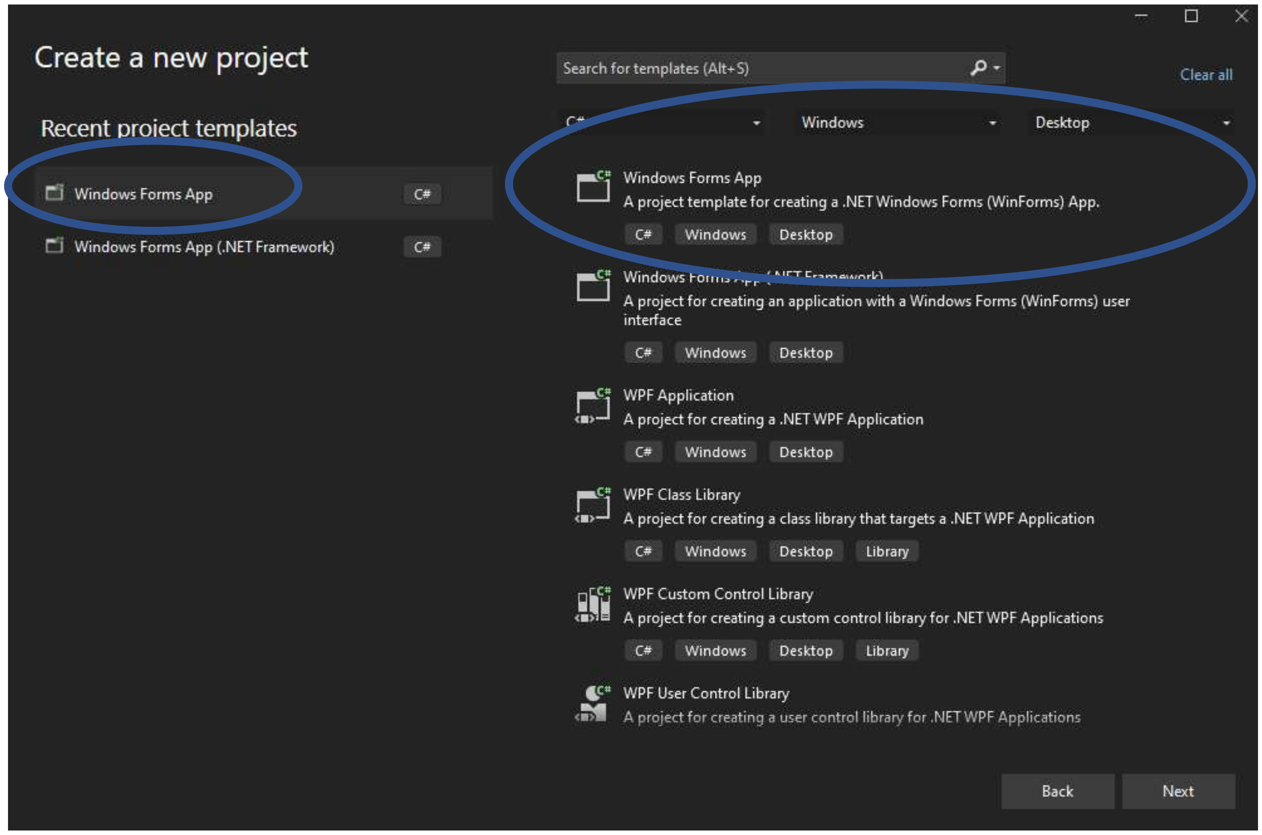

Get started
Clone a repository
Get code from an online repository like GitHub or Azure DevOps
Open a project or solution
Open a local Visual Studio project or .sln file
Open a local folder
Navigate and edit code within any folder
Create a new project
Choose a project template with code scaffolding to get started
Continue without code →

Figure 1: Select "Create a new project."

    b. Select the Project Template "WindowsFormsApp" as shown in Figure 2. Be sure to select "Windows Forms App." DO NOT select the one with (.NET Framework).



Create a new project

Recent project templates

Windows Forms App	C#
Windows Forms App (.NET Framework)	C#

Search for templates (Alt+S)

Clear all

C# | Windows | Desktop

Windows Forms App
A project template for creating a .NET Windows Forms (WinForms) App.
C# Windows Desktop

Windows Forms App (.NET Framework)
A project for creating an application with a Windows Forms (WinForms) user interface
C# Windows Desktop

WPF Application
A project for creating a .NET WPF Application
C# Windows Desktop

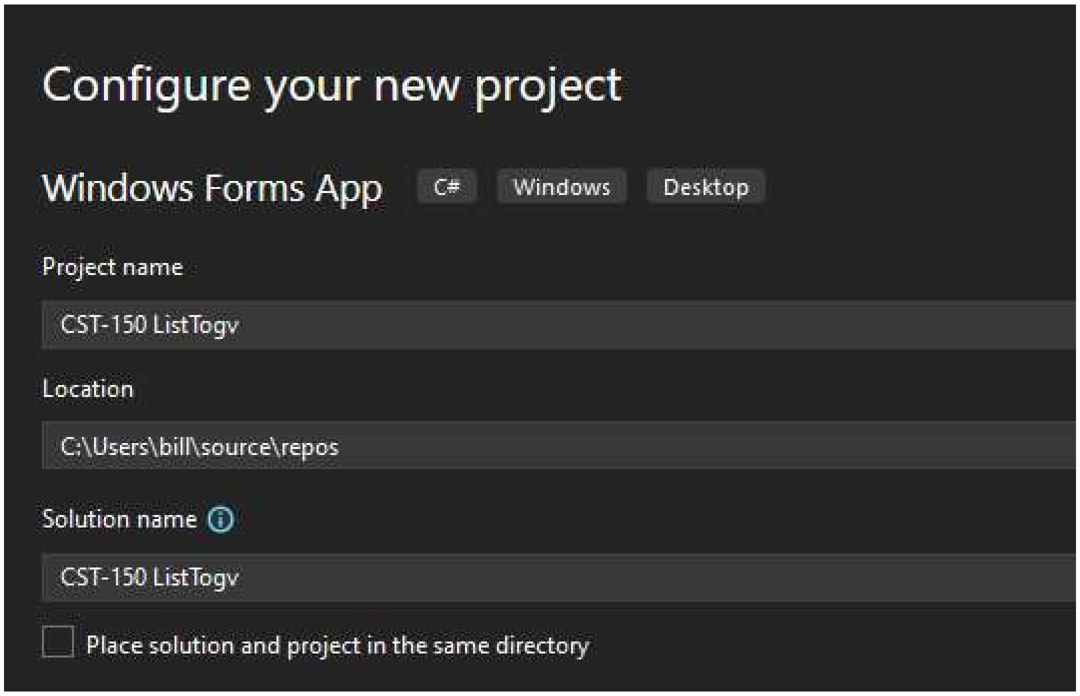
WPF Class Library
A project for creating a class library that targets a .NET WPF Application
C# Windows Desktop Library

WPF Custom Control Library
A project for creating a custom control library for .NET WPF Applications
C# Windows Desktop Library

WPF User Control Library
A project for creating a user control library for .NET WPF Applications

Figure 2: Select the Project Template.

c. Enter the Project name: "CST-150 ListTogv" as shown in Figure 3 followed by the "Next" button. Be sure the check box is not checked for "Place solution and project in the same directory."


Configure your new project

Windows Forms App

Project name: CST-150 ListTogv

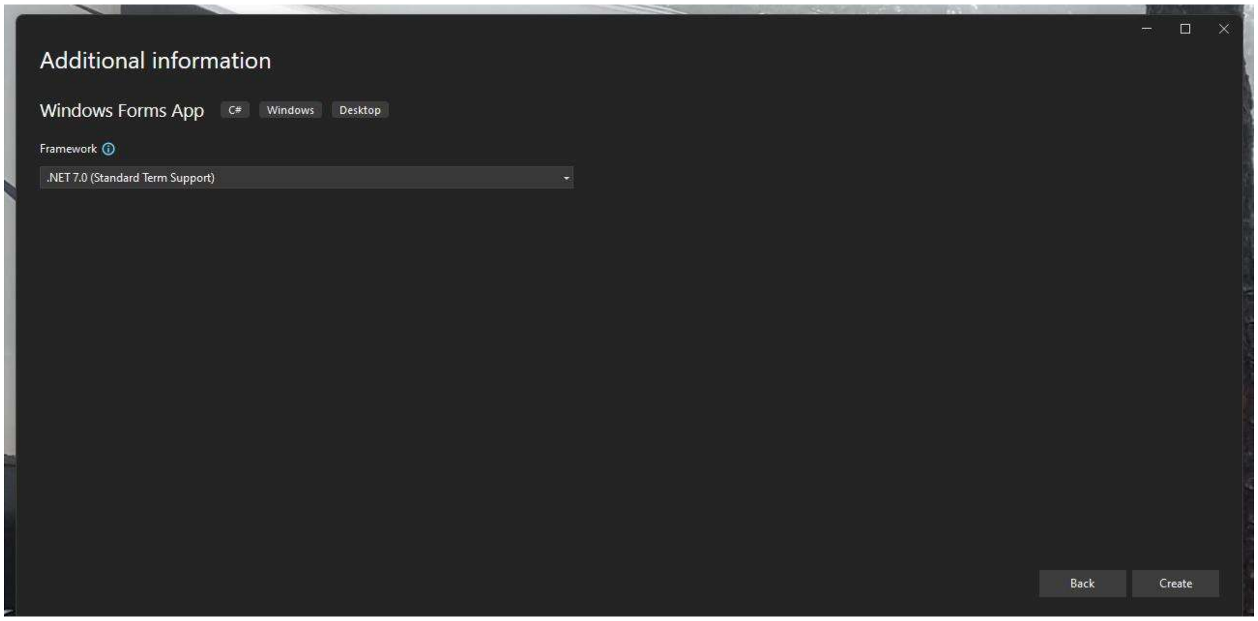
Location: C:\Users\bill\source\repos

Solution name: CST-150 ListTogv

Place solution and project in the same directory: [ ]

Figure 3: Enter Project Name.

d. Select the Framework as is shown in Figure 4 followed by the "Create" button. These in-class applications have all been tested using .NET 7.0. (The instructor may have the class us a different .NET version.)


Additional information

Windows Forms App

Framework: .NET 7.0 (Standard Term Support)

Figure 4: Select Framework.

2. Verify the application is working correctly.
    a. Follow the steps outlined in Activity 3.

3. Create a Model Class.
    a. This model plus the List will be used to hold the master inventory and pass the inventory up and down the Presentation and Business layers.
    b. In the Solution Explorer, create a "new folder" named "Models" as shown in Figure 5.

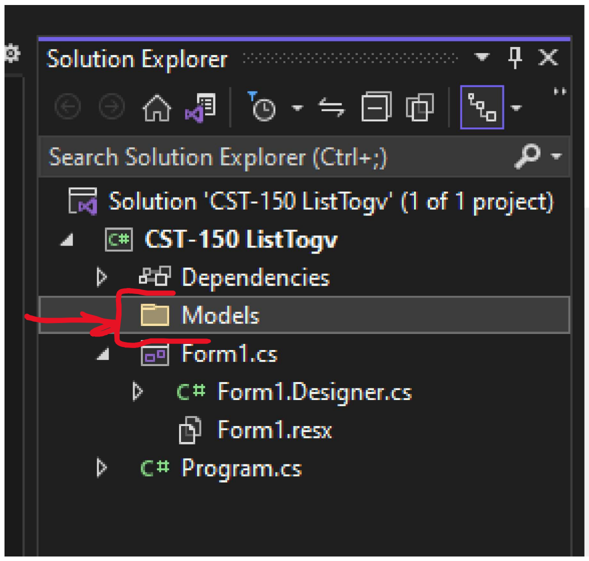


Figure 5: Models Folder

    c. In the "Models" folder, create a new class that will be used as a model class.
    d. A model will define the structure of our inventory attributes.
    e. Name the class "InvItem.cs" as shown in Figure 6. This name should be singular because it can only define one single inventory item at a time. We must instantiate it to have multiple items.

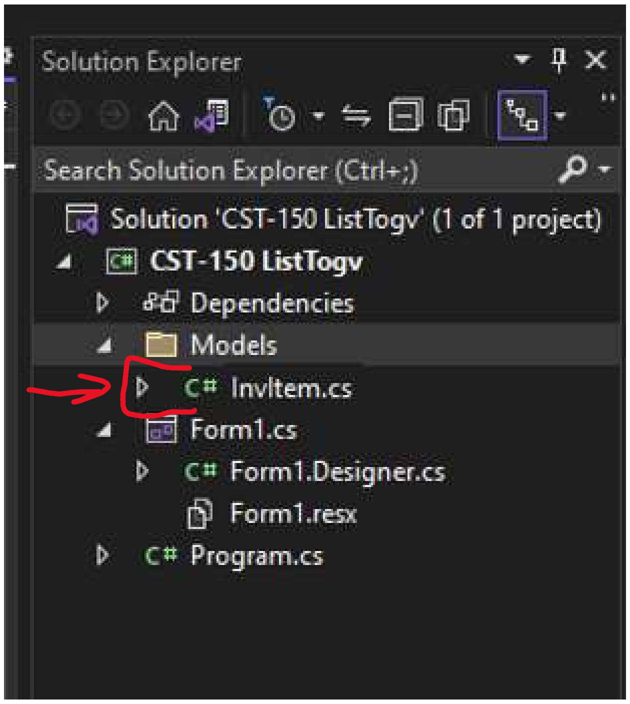


Figure 6: Add class named InvItem.cs.

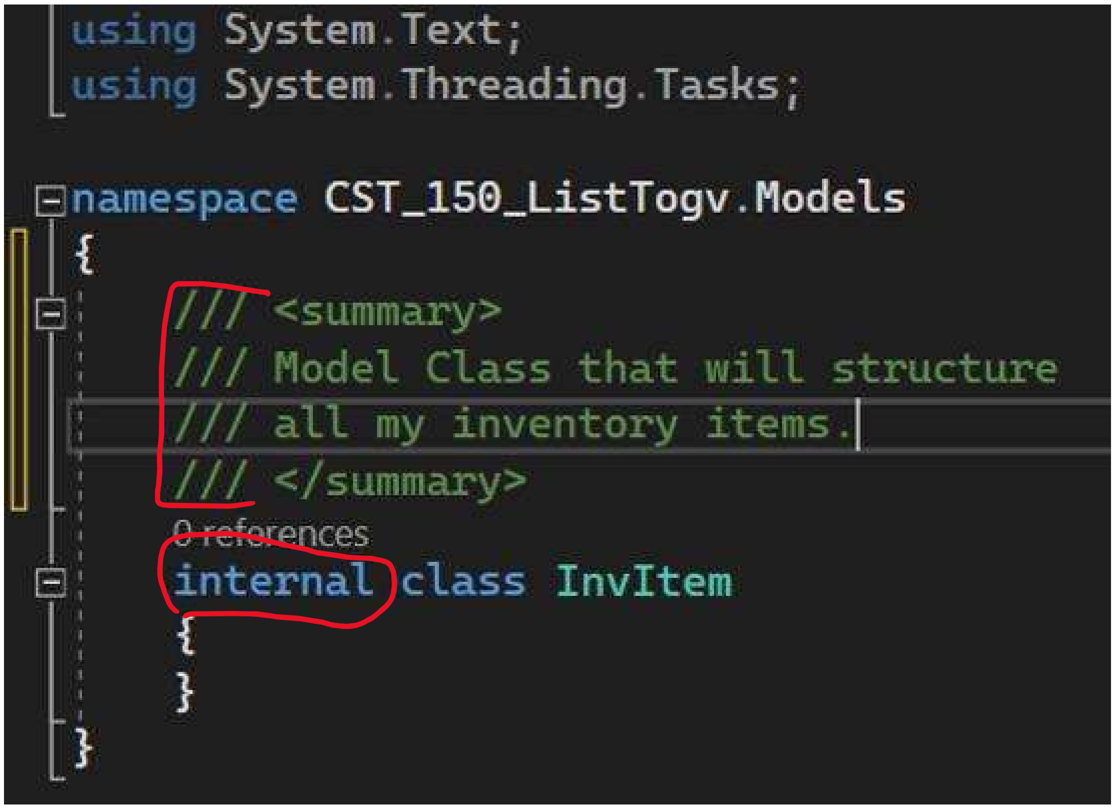
f. In the "InvItem.cs" class, add the summary comments as shown in Figure 7.

g. What is the internal keyword as shown in Figure 7?

h. In C#, the internal keyword can be used on a class or its members.

i. It is **one of the C# access modifiers.**

j. Internal types or members are accessible only within files in the same assembly.



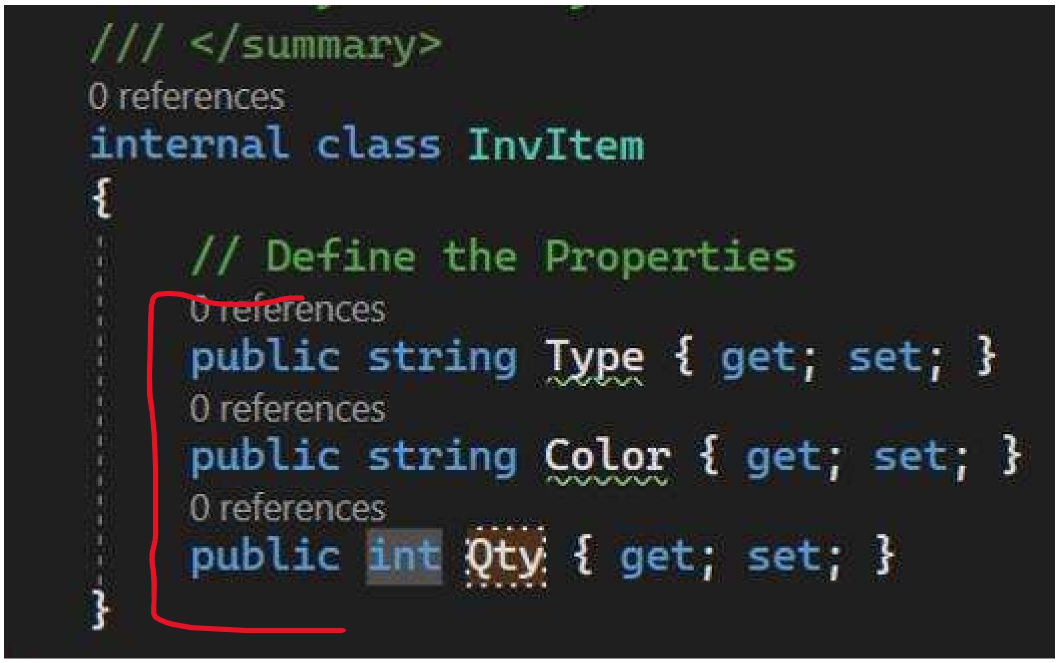
```
using System.Text;
using System.Threading.Tasks;

namespace CST_150_ListTogv.Models
{
    /// <summary>
    /// Model Class that will structure
    /// all my inventory items.
    /// </summary>
    0 references
    internal class InvItem
    {
    }
}
```

Figure 7: InvItem.cs

k. In the "InvItem.cs" class, define the properties as shown in Figure 8.

l. This activity has 3 properties; however, the milestone is required to have a minimum of 5 properties.

m. These properties should match the columns in the text file that we will be reading from.



```
/// </summary>
0 references
internal class InvItem
{
    // Define the Properties
    0 references
    public string Type { get; set; }
    0 references
    public string Color { get; set; }
    0 references
    public int Qty { get; set; }
}
```

Figure 8: Define the Properties

n. The model class only needs to have a parameterized constructor that initializes the properties.

o. Right click on the mouse the line you want to insert the parameterized constructor followed by Quick Actions.

p. Then, generate constructor as shown in Figure 9.

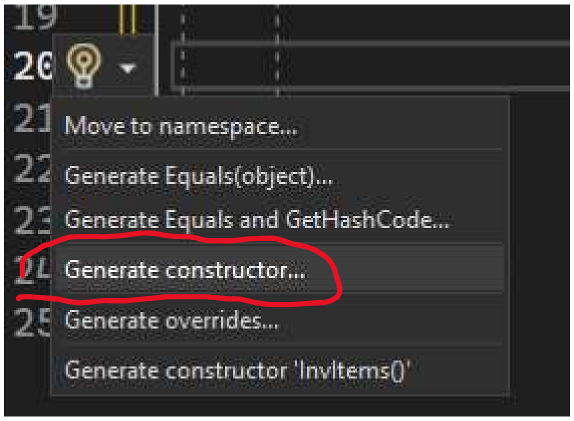
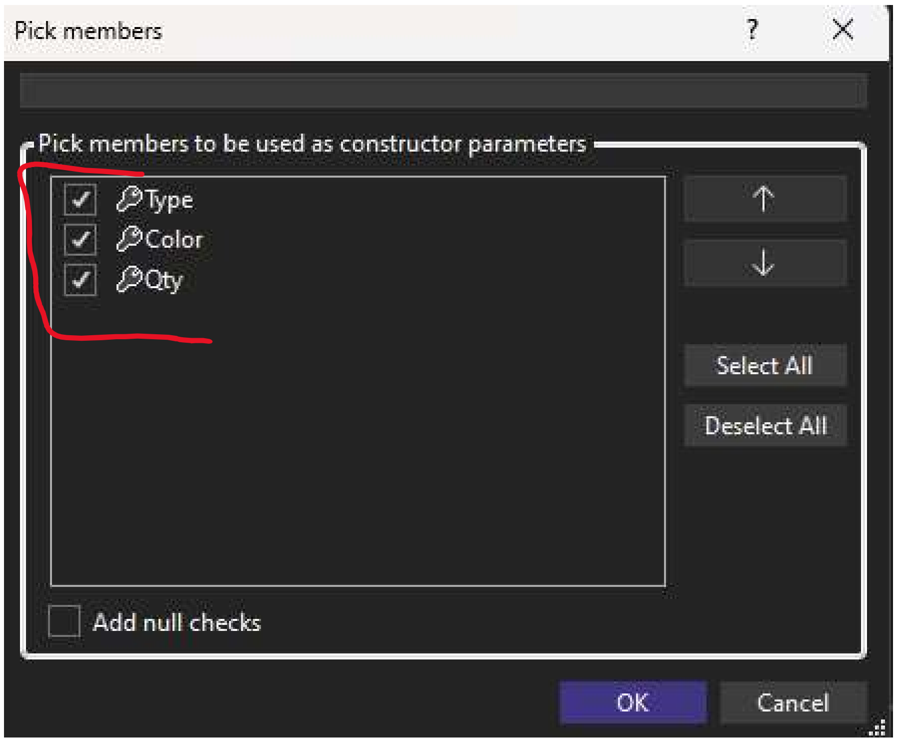

Move to namespace...
Generate Equals(object)...
Generate Equals and GetHashCode...
Generate constructor...
Generate overrides...
Generate constructor 'InvItems()'

Figure 9: Generate Constructor

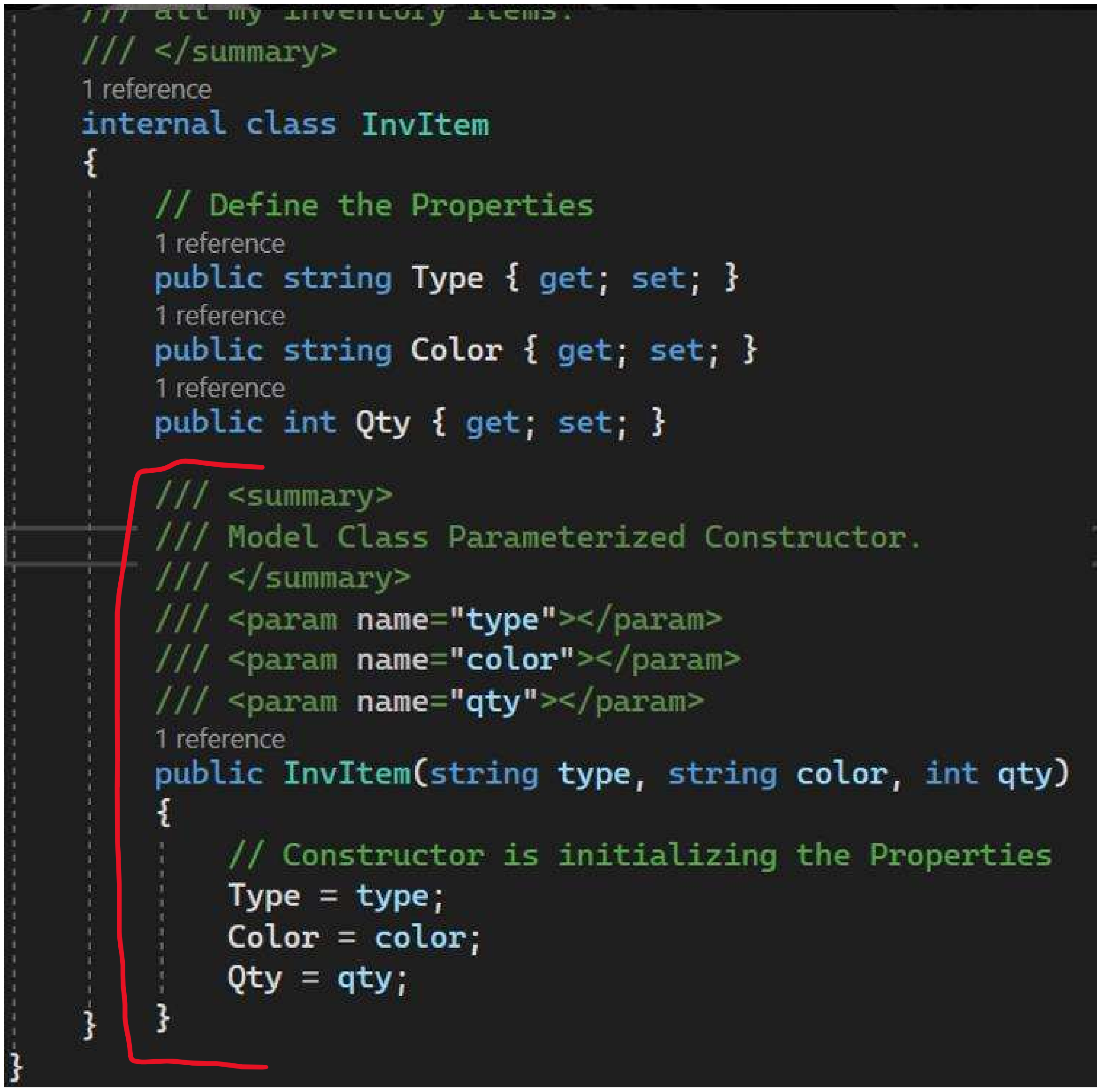
q. Since we want to initialize all the properties, be sure all are checked as shown in Figure 10.

r. Then, click OK to generate the Parameterized Constructor as shown in Figure 11.



- [✓] Type
- [✓] Color
- [✓] Qty
- [ ] Add null checks

Figure 10: Check all boxes.



```
    /// </summary>
    1 reference
    internal class InvItem
    {
        // Define the Properties
        1 reference
        public string Type { get; set; }
        1 reference
        public string Color { get; set; }
        1 reference
        public int Qty { get; set; }
        /// <summary>
        /// Model Class Parameterized Constructor.
        /// </summary>
        /// <param name="type"></param>
        /// <param name="color"></param>
        /// <param name="qty"></param>
        1 reference
        public InvItem(string type, string color, int qty)
        {
            // Constructor is initializing the Properties
            Type = type;
            Color = color;
            Qty = qty;
        }
    }
}
```

Figure 11: Model Class

4. Configure the Presentation Layer.
   a. This application will have the Presentation Layer on the root of the folders in the Solution Explorer.
   b. For "Form1.cs," rename this to "FrmInventory.cs" as shown in Figure 12.
   c. Then, change the "Text" property of the form to "Inventory" as shown in Figure 13.
   d. To complete the form, add a datagridview control and name it "gvInv."

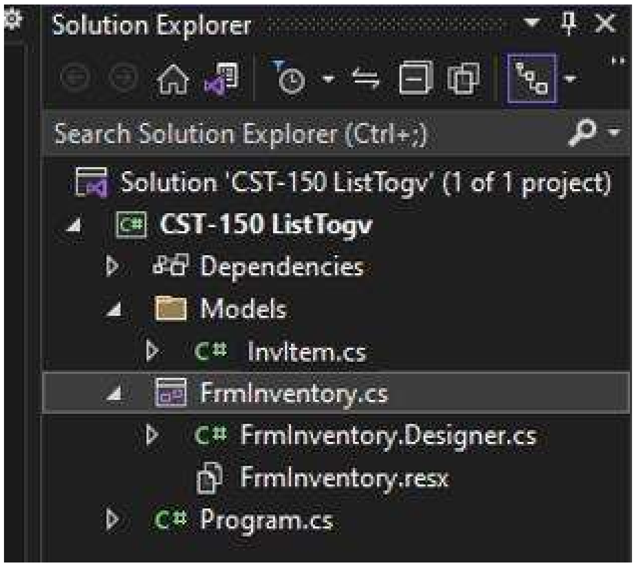


Figure 12: FrmInventory.cs

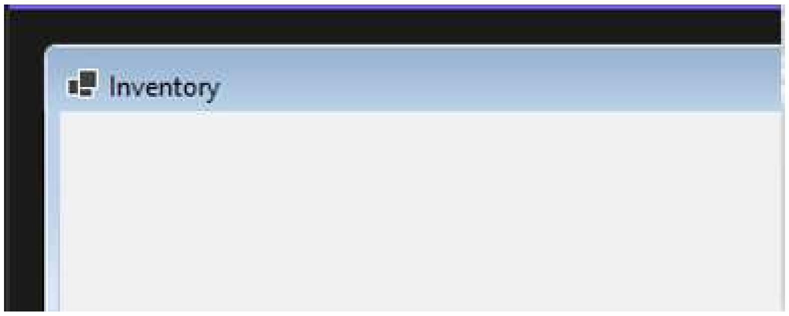


Figure 13: Text Property updated.

5. Create the Business Layer.
    a. Create a business layer that will read in our txt file.
    b. Then, add a new class to the business layer and call it "Inventory.cs" using PascalCasing as shown in Figure 14.

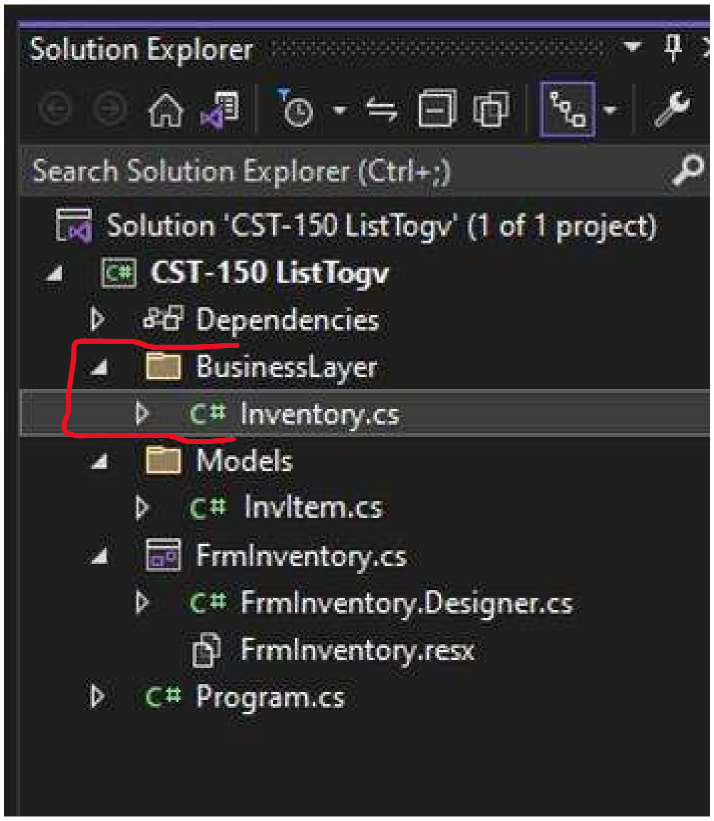


Figure 14: Business Layer

c. Using the Topic6.txt file provided by the instructor, place this text file in the data directory in the bin folder as shown in Figure 15.

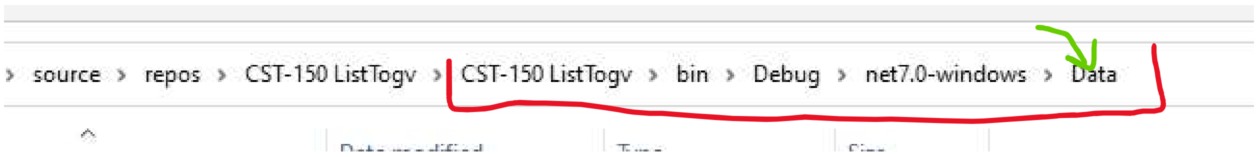
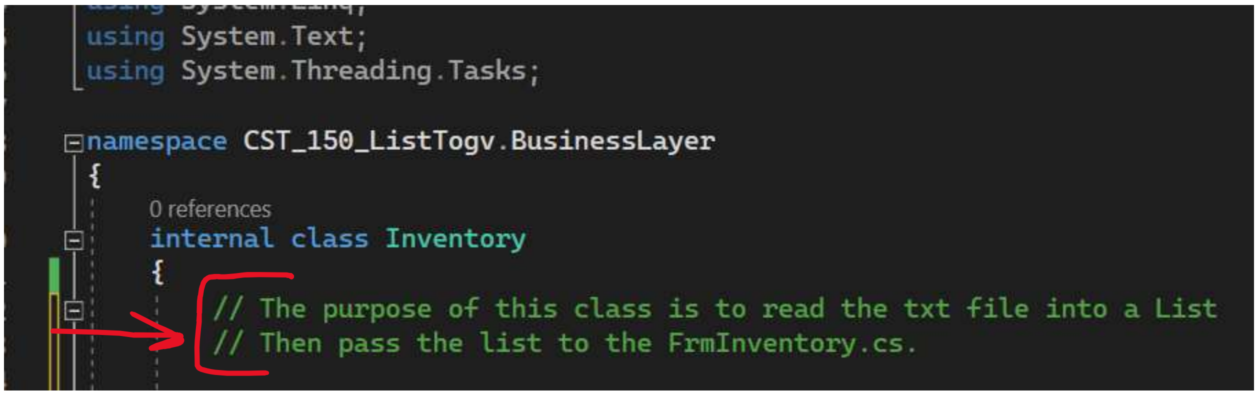

source › repos › CST-150 ListTogv › CST-150 ListTogv › bin › Debug › net7.0-windows › Data

Figure 15: Data folder

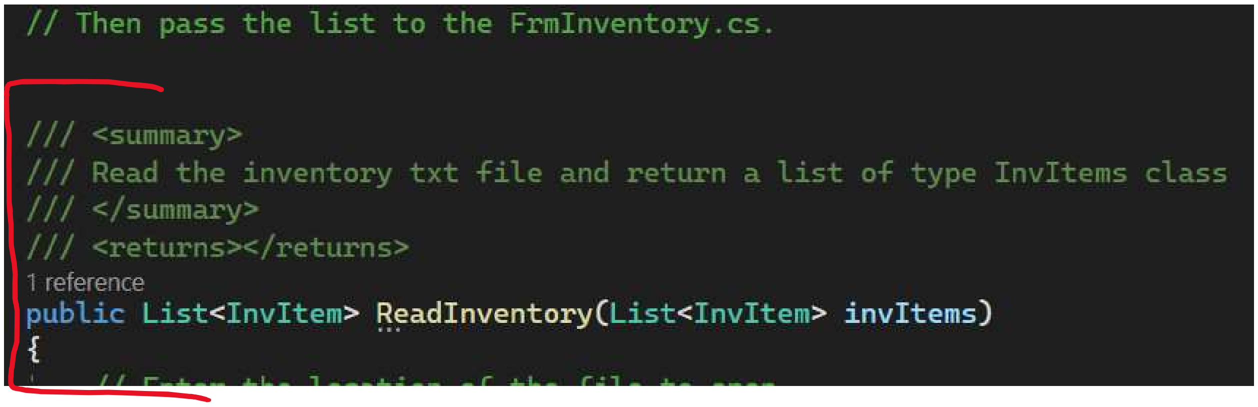
d. In Inventory.cs class, create a method that will read in the txt file and return a List of type InvItem class as shown in Figure 17.

e. Since the master inventory List is created in the FrmInventory.cs file, be sure to pass this master inventory List into the Parameter of this method.



```
using System.Text;
using System.Threading.Tasks;

namespace CST_150_ListTogv.BusinessLayer
{
    0 references
    internal class Inventory
    {
        // The purpose of this class is to read the txt file into a List
        // Then pass the list to the FrmInventory.cs.
```

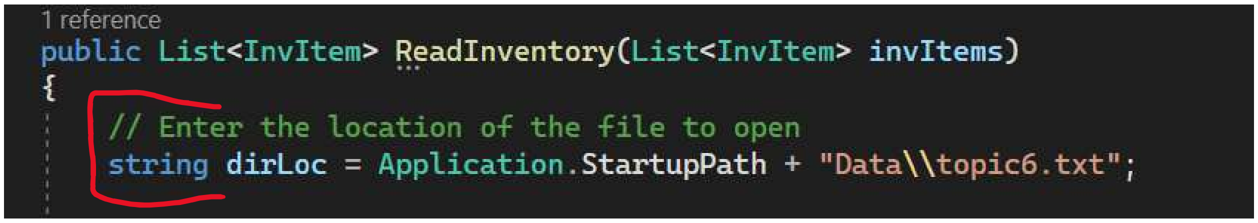
Figure 16: Purpose of this class.



```
// Then pass the list to the FrmInventory.cs.


/// <summary>
/// Read the inventory txt file and return a list of type InvItems class
/// </summary>
/// <returns></returns>
1 reference
public List<InvItem> ReadInventory(List<InvItem> invItems)
{
```

Figure 17: Create method.

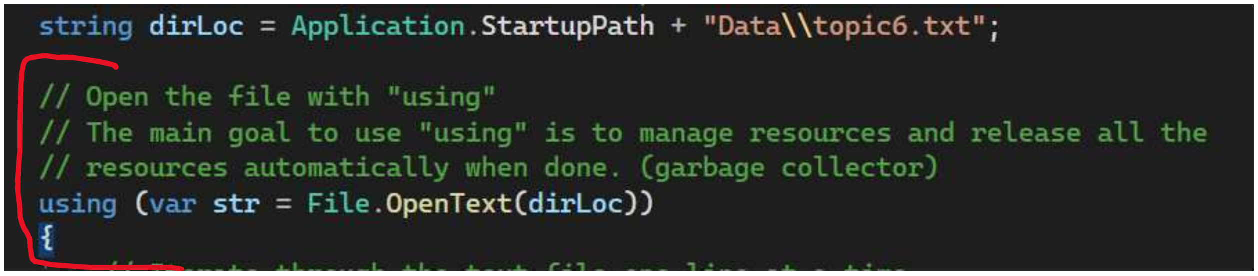
f. The first step is to identify where the text file is located as shown in Figure 18.



```
1 reference
public List<InvItem> ReadInventory(List<InvItem> invItems)
{
    // Enter the location of the file to open
    string dirLoc = Application.StartupPath + "Data\\topic6.txt";
```

Figure 18: Identify where the txt file is located.

g. Then, open the file with "using" statement as shown in Figure 19. Be sure to write all the comments to fully understand "using."



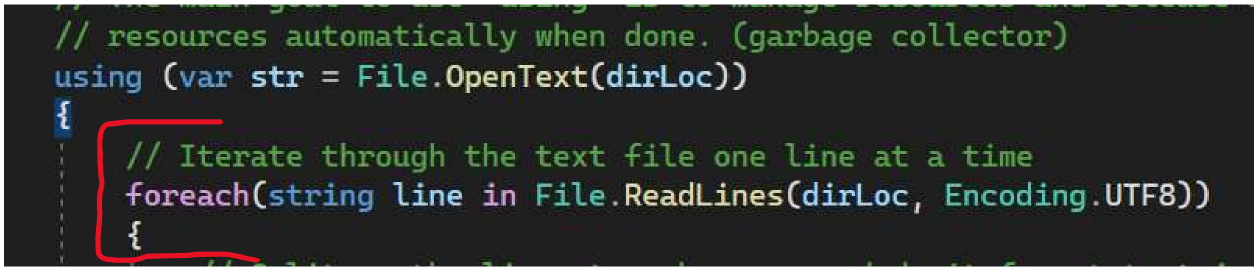
```
string dirLoc = Application.StartupPath + "Data\\topic6.txt";

// Open the file with "using"
// The main goal to use "using" is to manage resources and release all the
// resources automatically when done. (garbage collector)
using (var str = File.OpenText(dirLoc))
{
```

Figure 19: Open txt file.

h. Now, iterate over each line in the text file as shown in Figure 20.

Note: Activity 3 reads the entire txt file in one statement, so this time we are changing that process to read in one line at a time.

i. Encoding.UTF8 →ASCII only encodes English characters, so if you used other languages whose alphabet does not consist of English characters, the text would not properly display on your screen. Thus, UTF-8 was created to address ASCII's shortcomings; it can translate almost every language in the world.



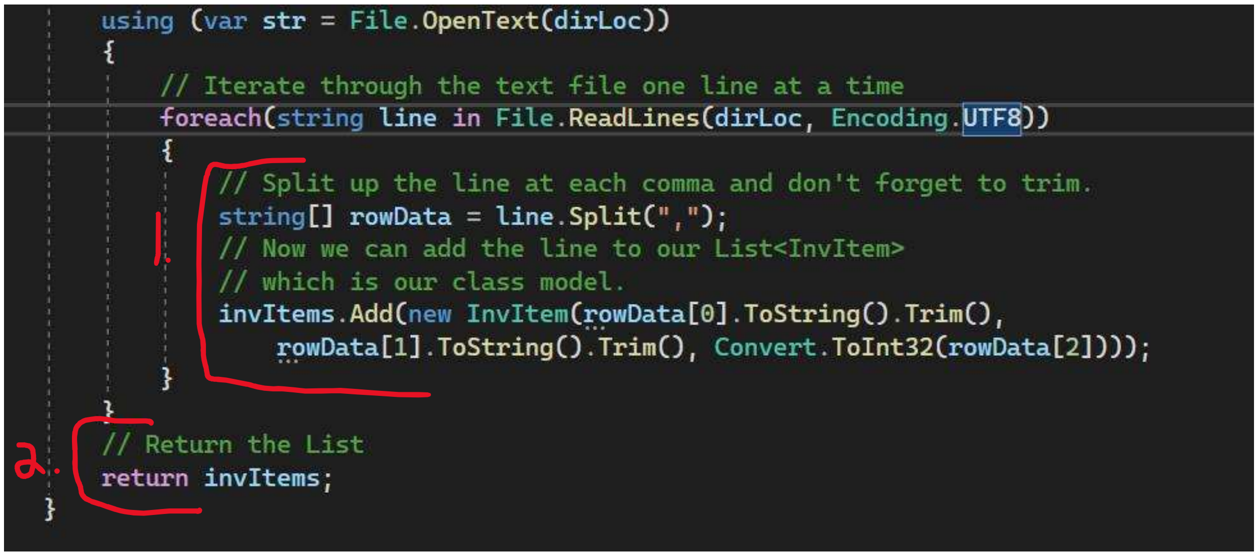
```
// resources automatically when done. (garbage collector)
using (var str = File.OpenText(dirLoc))
{
    // Iterate through the text file one line at a time
    foreach(string line in File.ReadLines(dirLoc, Encoding.UTF8))
    {
```

Figure 20: Read txt file one line at a time.

j. Split up the lines and create the List as shown in Figure 21.

k. Then, return the inventory list as shown in the figure as number 2.

l. **Note:** When lines are added to the list, there is a Convert statement being used. As an extra item, work on moving this convert statement before the line is added to the list. Then, manage the exception that could potentially occur with the convert in this new section of code and replace the current Convert statement with the variable created in the exception handling.

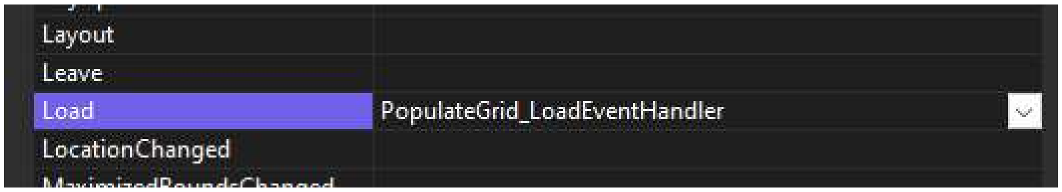


```
    using (var str = File.OpenText(dirLoc))
    {
        // Iterate through the text file one line at a time
        foreach(string line in File.ReadLines(dirLoc, Encoding.UTF8))
        {
            // Split up the line at each comma and don't forget to trim.
            string[] rowData = line.Split(",");
            // Now we can add the line to our List<InvItem>
            // which is our class model.
            invItems.Add(new InvItem(rowData[0].ToString().Trim(),
                rowData[1].ToString().Trim(), Convert.ToInt32(rowData[2])));
        }
    }
    // Return the List
    return invItems;
}
```

Figure 21: Create the List.

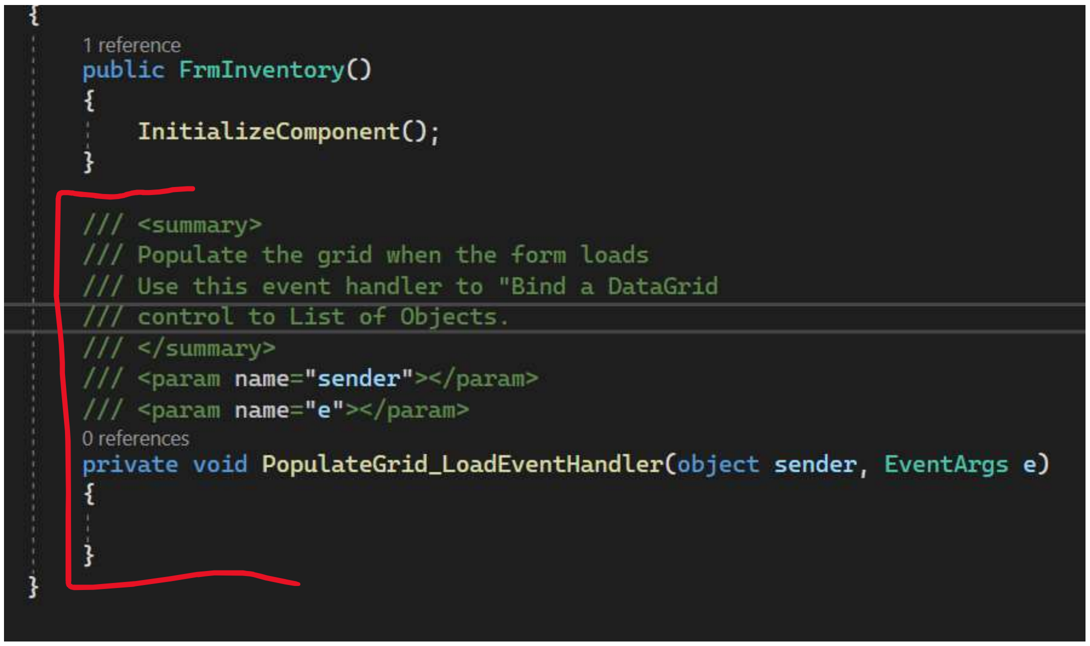
6. Write code behind FrmInventory.

   a. Populate the datagridview when the form loads as the event handler and shown in Figure 22 and Figure 23.



Layout	
Leave	
Load	PopulateGrid_LoadEventHandler
LocationChanged	

Figure 22: Form Load Event Handler

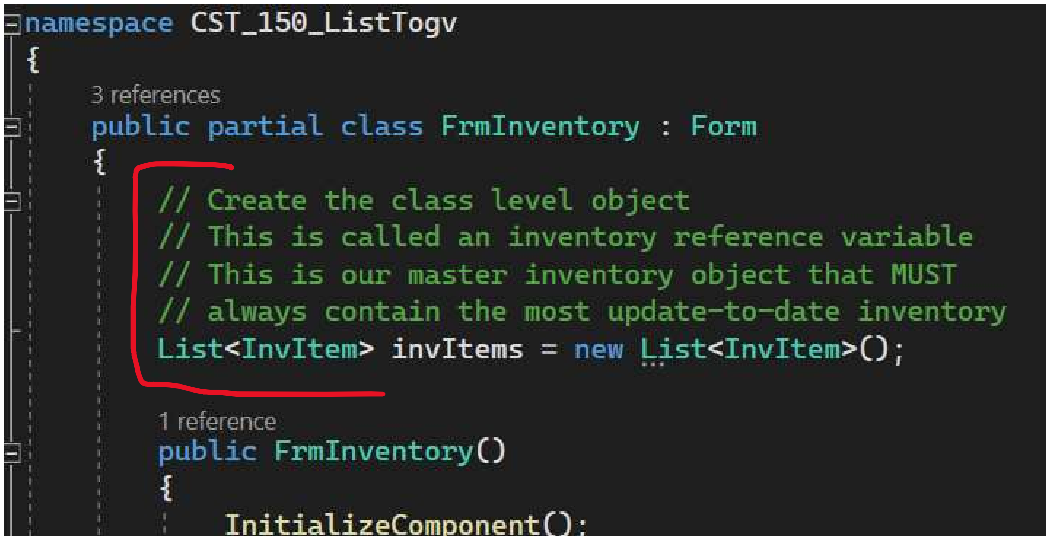


```
{
    1 reference
    public FrmInventory()
    {
        InitializeComponent();
    }

    /// <summary>
    /// Populate the grid when the form loads
    /// Use this event handler to "Bind a DataGrid
    /// control to List of Objects.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    0 references
    private void PopulateGrid_LoadEventHandler(object sender, EventArgs e)
    {

    }
}
```

Figure 23: Comment the Event Handler

b. At the class level, instantiate the model class as shown in Figure 24.

c. Be sure to write all the comments. This List is the Master Inventory List that will be used throughout the entire application.



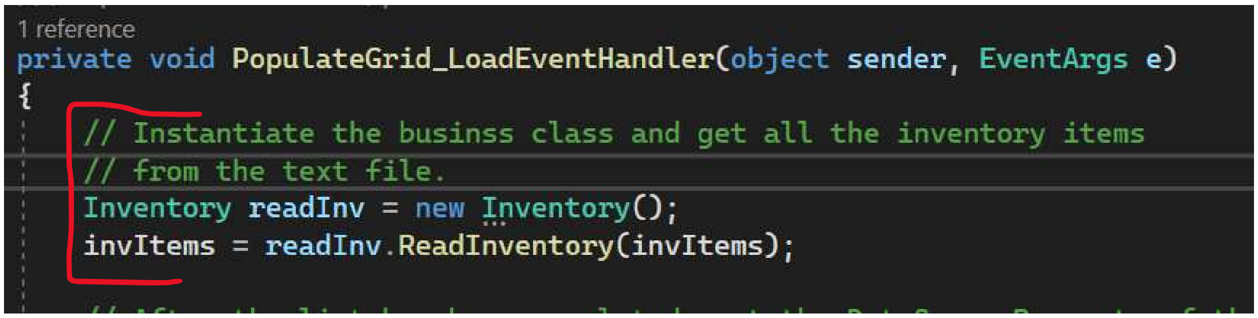
```
namespace CST_150_ListTogv
{
    3 references
    public partial class FrmInventory : Form
    {
        // Create the class level object
        // This is called an inventory reference variable
        // This is our master inventory object that MUST
        // always contain the most update-to-date inventory
        List<InvItem> invItems = new List<InvItem>();

        1 reference
        public FrmInventory()
        {
            InitializeComponent();
```

Figure 24: Create the master inventory list.

d. Back in the PopulateGrid event handler.

e. Inside the event handler, get the inventory as shown in Figure 25.

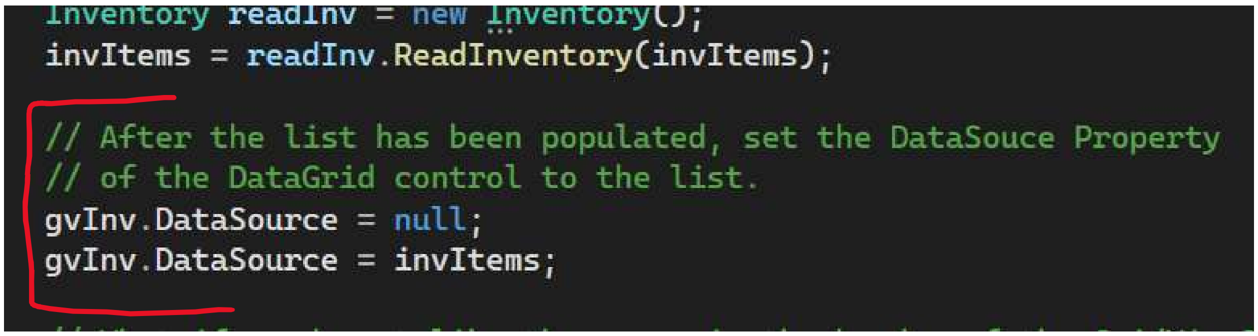
f. Set a breakpoint at the instantiation line. Then, using the debug tools, step through the code watching the execution go to the inventory class and read the text file.



```
1 reference
private void PopulateGrid_LoadEventHandler(object sender, EventArgs e)
{
    // Instantiate the businss class and get all the inventory items
    // from the text file.
    Inventory readInv = new Inventory();
    invItems = readInv.ReadInventory(invItems);
```

Figure 25: Get the Inventory

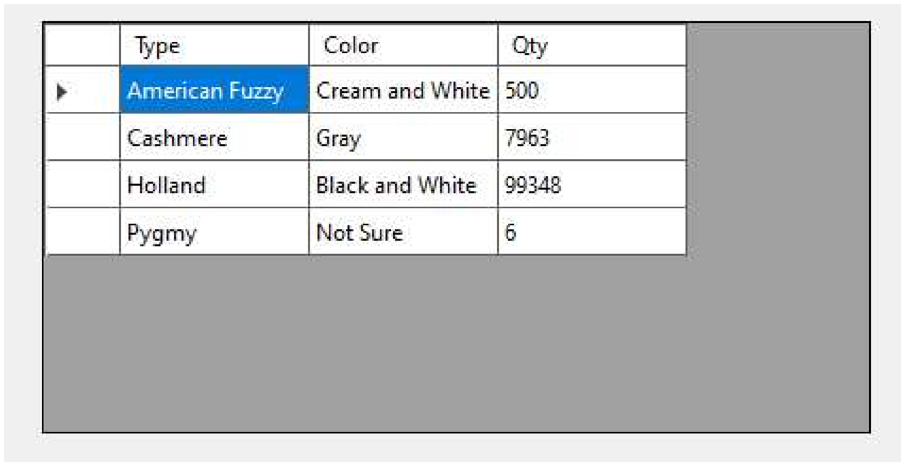
g. All we have to do once we have the inventory is bind the object to the grid data source as shown in Figure 26. The null ensures there is no data source already bound to the data grid view.



```
Inventory readInv = new Inventory();
invItems = readInv.ReadInventory(invItems);

// After the list has been populated, set the DataSouce Property
// of the DataGrid control to the list.
gvInv.DataSource = null;
gvInv.DataSource = invItems;
```

Figure 26: Bind the Data

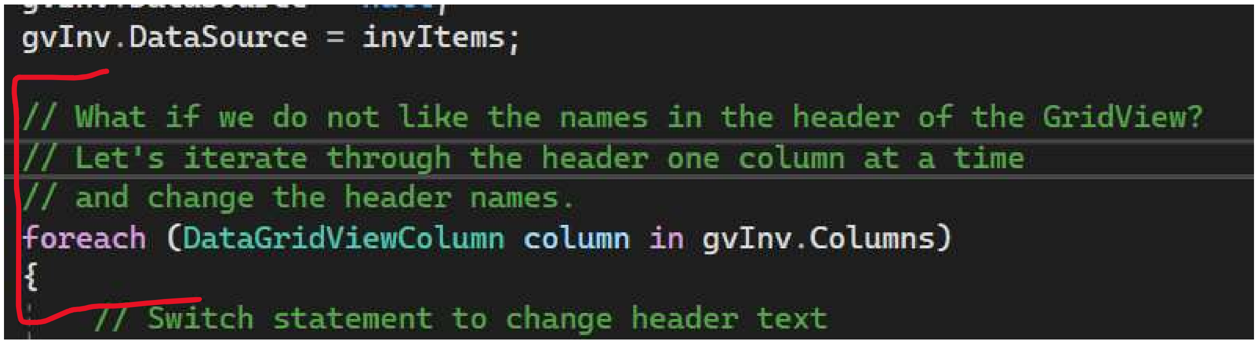
h. The result should look like Figure 27, with headers that are not named as we would like and numbers that are not right aligned. The result will be different based on the text file provided by the instructor.



	Type	Color	Qty
▶	American Fuzzy	Cream and White	500
	Cashmere	Gray	7963
	Holland	Black and White	99348
	Pygmy	Not Sure	6

Figure 27: Result when ran.

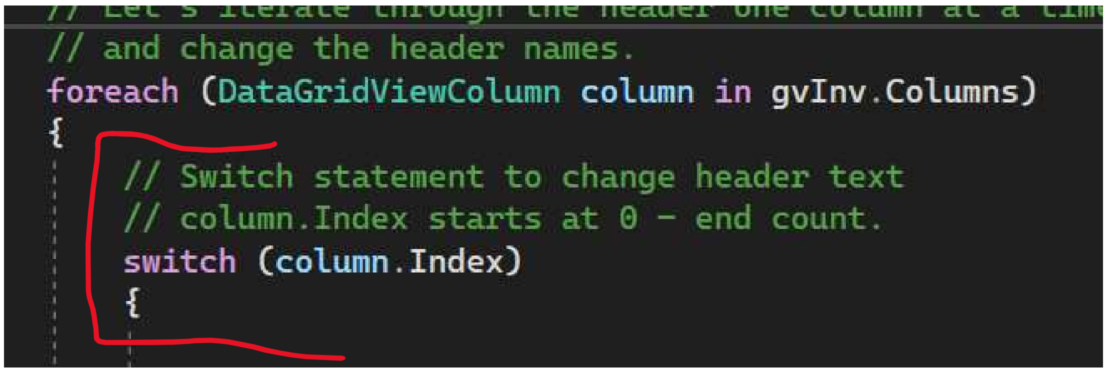
7. Work with Data Grid View using code behind (format columns).
   a. What if we want to change the header text and align the columns?
   b. Use a foreach loop with Switch Statement to iterate through the columns in the data grid view as shown in Figure 28.



```
gvInv.DataSource = invItems;

// What if we do not like the names in the header of the GridView?
// Let's iterate through the header one column at a time
// and change the header names.
foreach (DataGridViewColumn column in gvInv.Columns)
{
    // Switch statement to change header text
```

Figure 28: Iterate over the grid columns.

   c. Using the column index from the grid, we can use a switch statement to change and format the header text as shown in Figure 29.

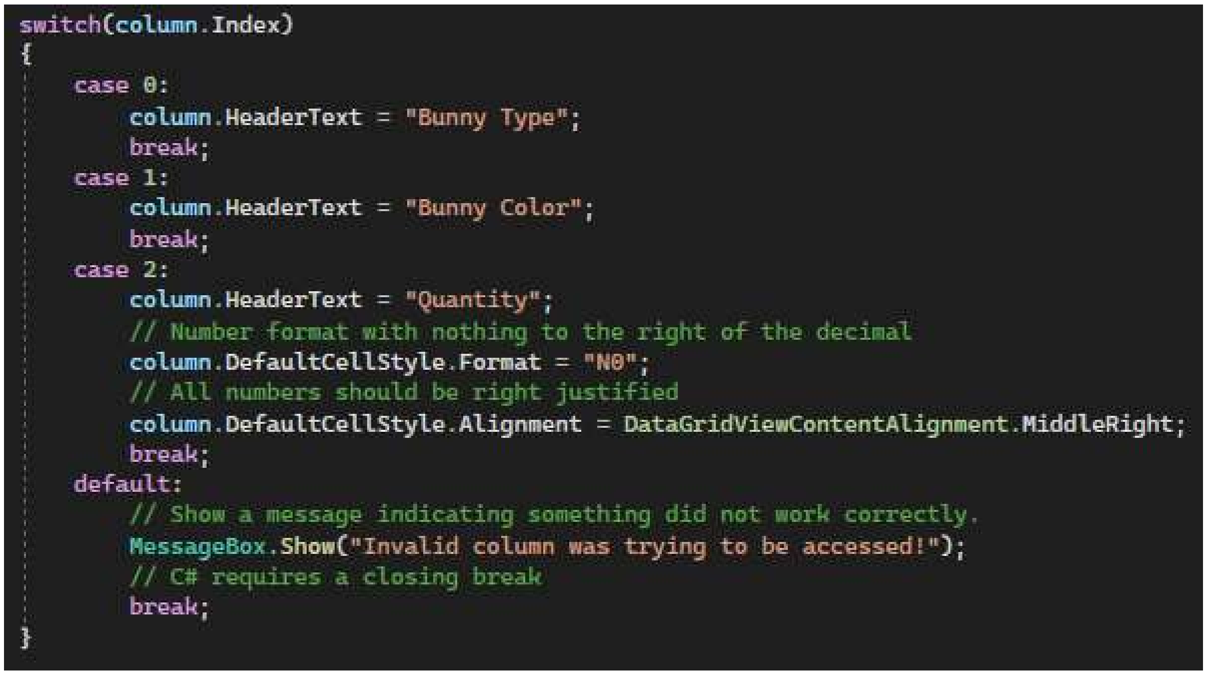


```
// Let's iterate through the header one column at a time
// and change the header names.
foreach (DataGridViewColumn column in gvInv.Columns)
{
    // Switch statement to change header text
    // column.Index starts at 0 - end count.
    switch (column.Index)
    {
```

Figure 29: Column Index

d. Now, we can individually change the Header Text by identifying the column index as shown in Figure 30.

e. Also, format the Quantity so they are comma formatted and positioned to the right of the cell.



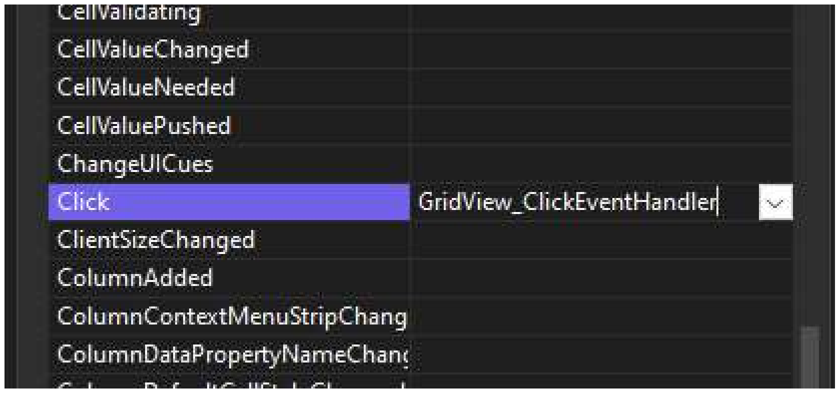
```
switch(column.Index)
{
    case 0:
        column.HeaderText = "Bunny Type";
        break;
    case 1:
        column.HeaderText = "Bunny Color";
        break;
    case 2:
        column.HeaderText = "Quantity";
        // Number format with nothing to the right of the decimal
        column.DefaultCellStyle.Format = "N0";
        // All numbers should be right justified
        column.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
        break;
    default:
        // Show a message indicating something did not work correctly.
        MessageBox.Show("Invalid column was trying to be accessed!");
        // C# requires a closing break
        break;
}
```

Figure 30: Modify each column.

8. Work with Data Grid View using code behind (Select Row).

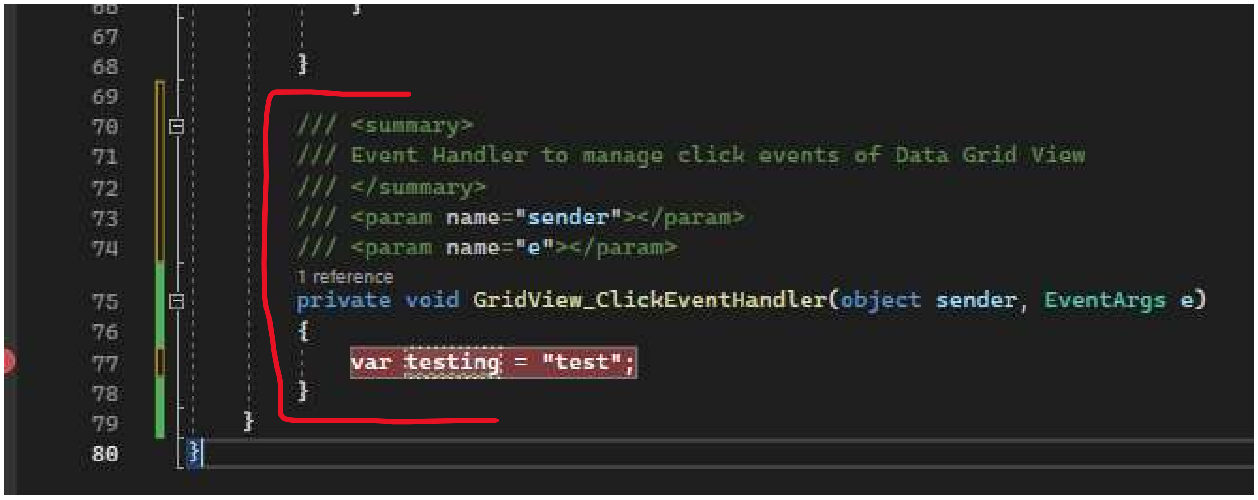
   a. How to get the selected row in a DataGridView: select the Data Grid View control in the form and create a Click Event Handler named "GridView_ClickEventHandler" as shown in Figure 31.

b. Then, create a break on a test line to verify clicking the DataGridView triggers this event as shown in Figure 32.



CellValidating	
CellValueChanged	
CellValueNeeded	
CellValuePushed	
ChangeUICues	
Click	GridView_ClickEventHandler
ClientSizeChanged	
ColumnAdded	
ColumnContextMenuStripChang	
ColumnDataPropertyNameChang	

Figure 31: Grid Click Event Handler

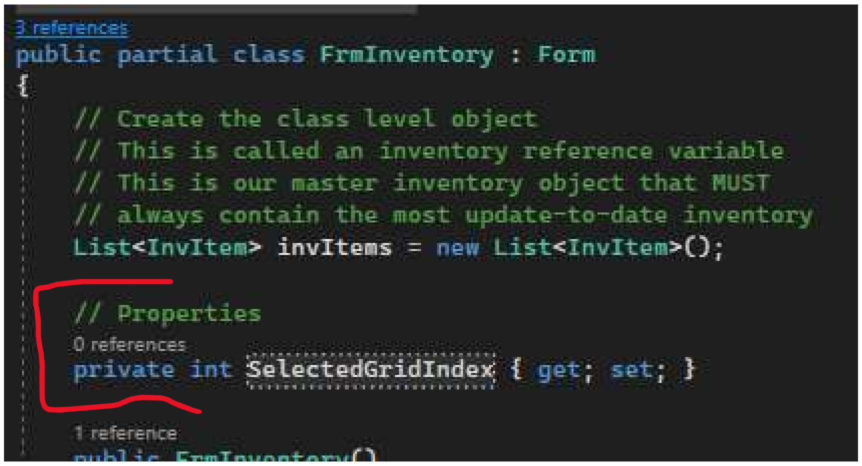


```
        }
    }

        /// <summary>
        /// Event Handler to manage click events of Data Grid View
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        1 reference
        private void GridView_ClickEventHandler(object sender, EventArgs e)
        {
            var testing = "test";
        }
    }
}
```

Figure 32: Test to verify operation.

c. Now let's get the row, so we know where in the list to work from.

d. First, define a property that will hold the SelectedGridIndex at the class level as shown in Figure 33.

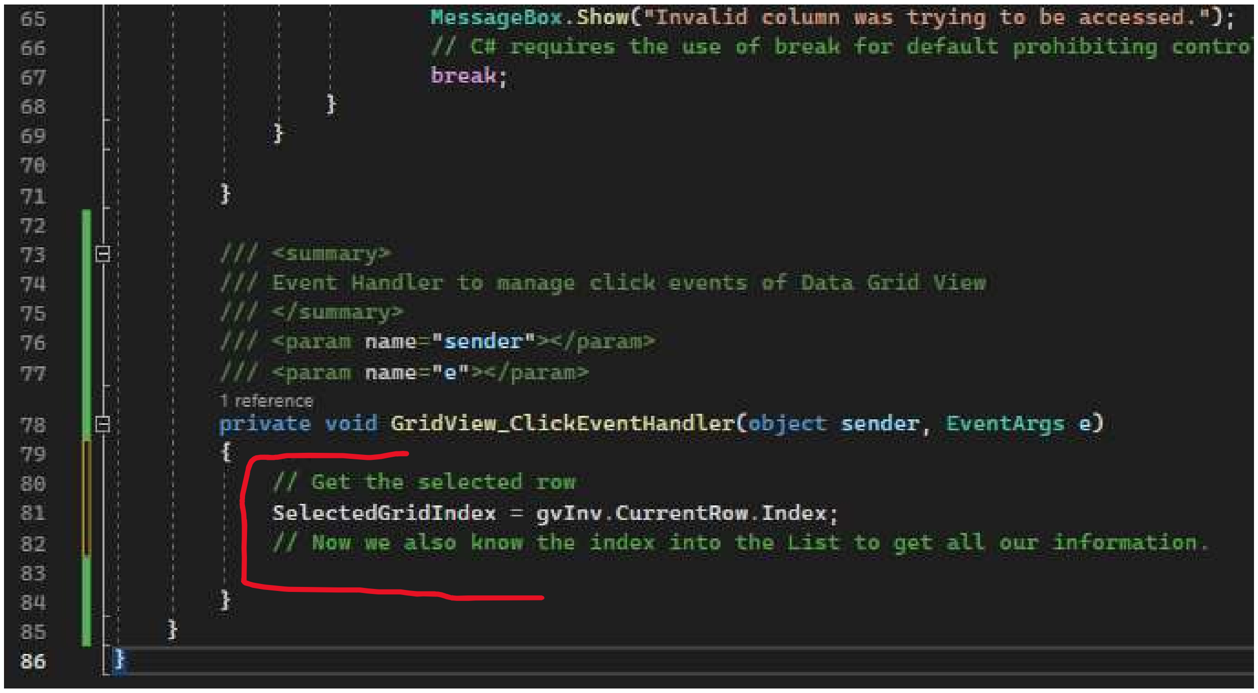


```
3 references
public partial class FrmInventory : Form
{
    // Create the class level object
    // This is called an inventory reference variable
    // This is our master inventory object that MUST
    // always contain the most update-to-date inventory
    List<InvItem> invItems = new List<InvItem>();

    // Properties
    0 references
    private int SelectedGridIndex { get; set; }

    1 reference
    public FrmInventory()
```

Figure 33: Add Property.

e. In the GridView, click event handler and remove the test line we put in earlier.

f. Get the current selected row index of the data grid view and place the index into the property as shown in Figure 34.



```
                        MessageBox.Show("Invalid column was trying to be accessed.");
                        // C# requires the use of break for default prohibiting contro
                        break;
                }
            }

        }

        /// <summary>
        /// Event Handler to manage click events of Data Grid View
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        1 reference
        private void GridView_ClickEventHandler(object sender, EventArgs e)
        {
            // Get the selected row
            SelectedGridIndex = gvInv.CurrentRow.Index;
            // Now we also know the index into the List to get all our information.

        }
    }
}
```

Figure 34: Get Selected Row

9. Work with Data Grid View using code behind (Increment Quantity Value).

   a. Sample exercise showing how to increment the quantity of the selected row.

   b. On the inventory form, place a button above the data grid view as shown in Figure 35.

c. Name this button "btnIncQty" and change the text property to "Inc Qty."

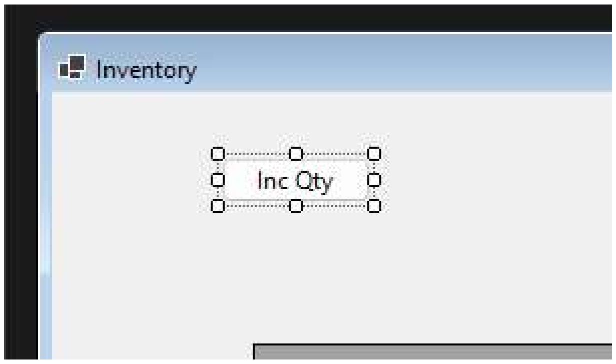
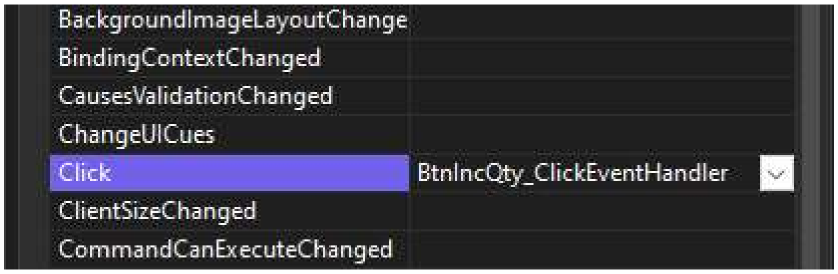


Figure 35: Button on form.

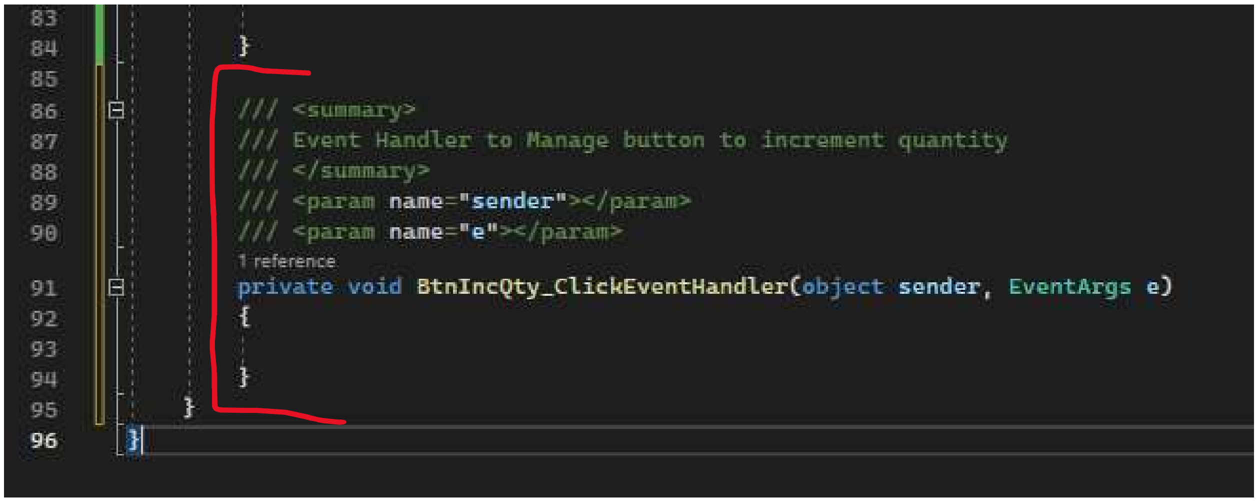
d. With the button selected, create a click event handler as shown in Figure 36.

e. Then, comment the method using the /// summary comment as shown in Figure 37. This is the method that will manage incrementing the qty.



BackgroundImageLayoutChange	
BindingContextChanged	
CausesValidationChanged	
ChangeUICues	
Click	BtnIncQty_ClickEventHandler
ClientSizeChanged	
CommandCanExecuteChanged	

Figure 36: Button click event handler.



```
        }

        /// <summary>
        /// Event Handler to Manage button to increment quantity
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        1 reference
        private void BtnIncQty_ClickEventHandler(object sender, EventArgs e)
        {

        }
    }
}
```

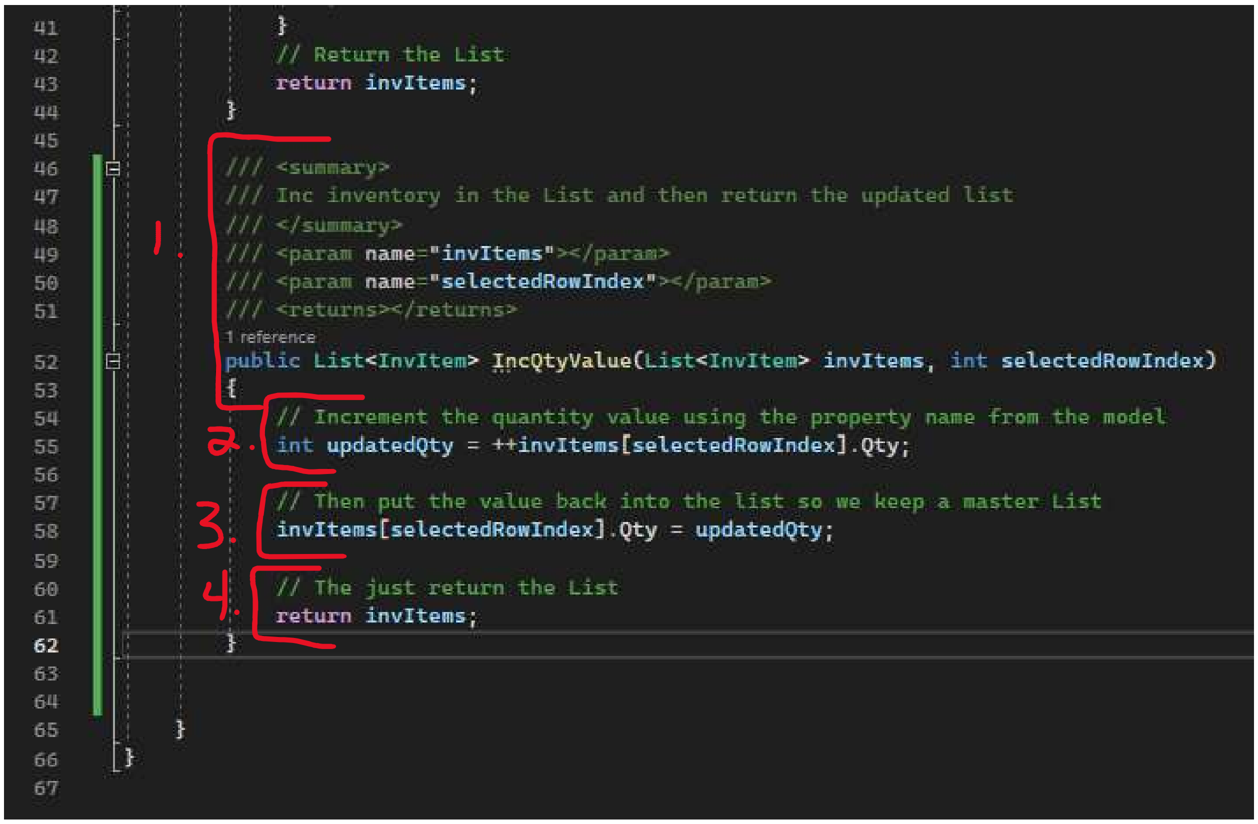
Figure 37: Add comments to click event handler.

f. First item: we need a method that will actually perform the incrementing.

g. Since incrementing the qty is logic, we need to move into the Inventory class and place all logic in the Business Layer.

h. Create a method to increment the qty column in the List as shown in Figure 38.

i. Be sure the List maintains the master inventory. We need to pass this List into the method and then return the updated List after the qty is incremented.



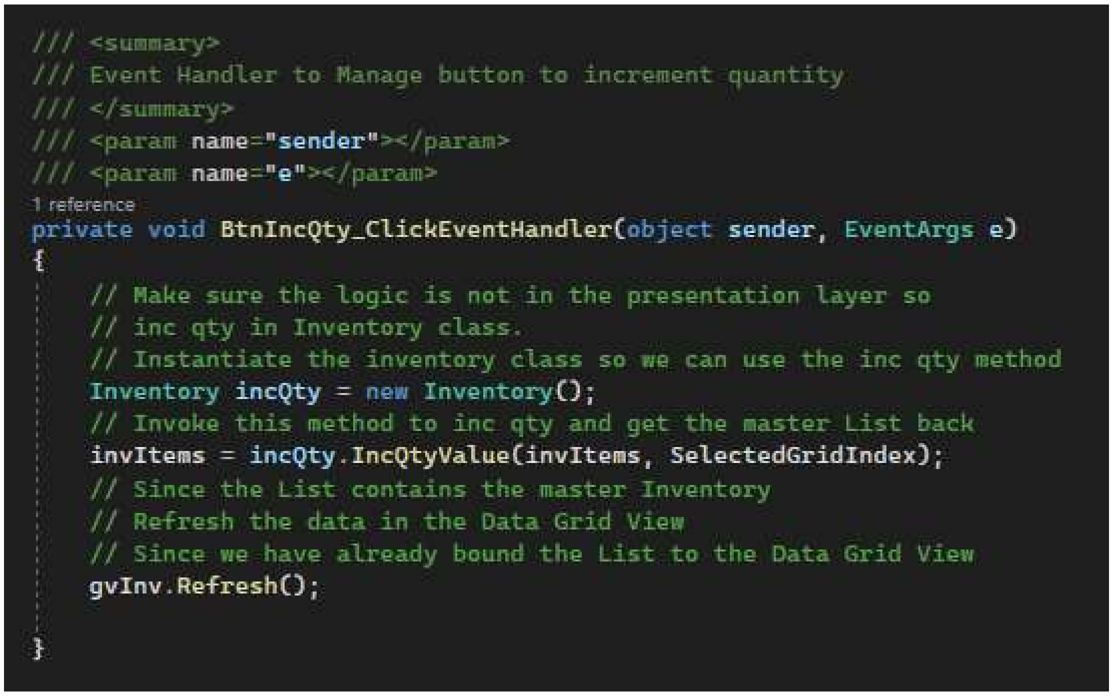
```
            }
            // Return the List
            return invItems;
        }

        /// <summary>
        /// Inc inventory in the List and then return the updated list
        /// </summary>
        /// <param name="invItems"></param>
        /// <param name="selectedRowIndex"></param>
        /// <returns></returns>
        public List<InvItem> IncQtyValue(List<InvItem> invItems, int selectedRowIndex)
        {
            // Increment the quantity value using the property name from the model
            int updatedQty = ++invItems[selectedRowIndex].Qty;

            // Then put the value back into the list so we keep a master List
            invItems[selectedRowIndex].Qty = updatedQty;

            // The just return the List
            return invItems;
        }


    }
}
```

Figure 38: Increment Inventory

j. Back to the Qty Inc click event handler in the presentation layer.

k. All we do is invoke the new method in the inventory class and get the master Inventory List back as shown in Figure 39.

l. Then, refresh this new data back to the data grid view; since the list was already bound to the DataSource earlier, all we need to do is just refresh the data. If we added or removed a row, then we must re-bind the data as a refresh will not work.



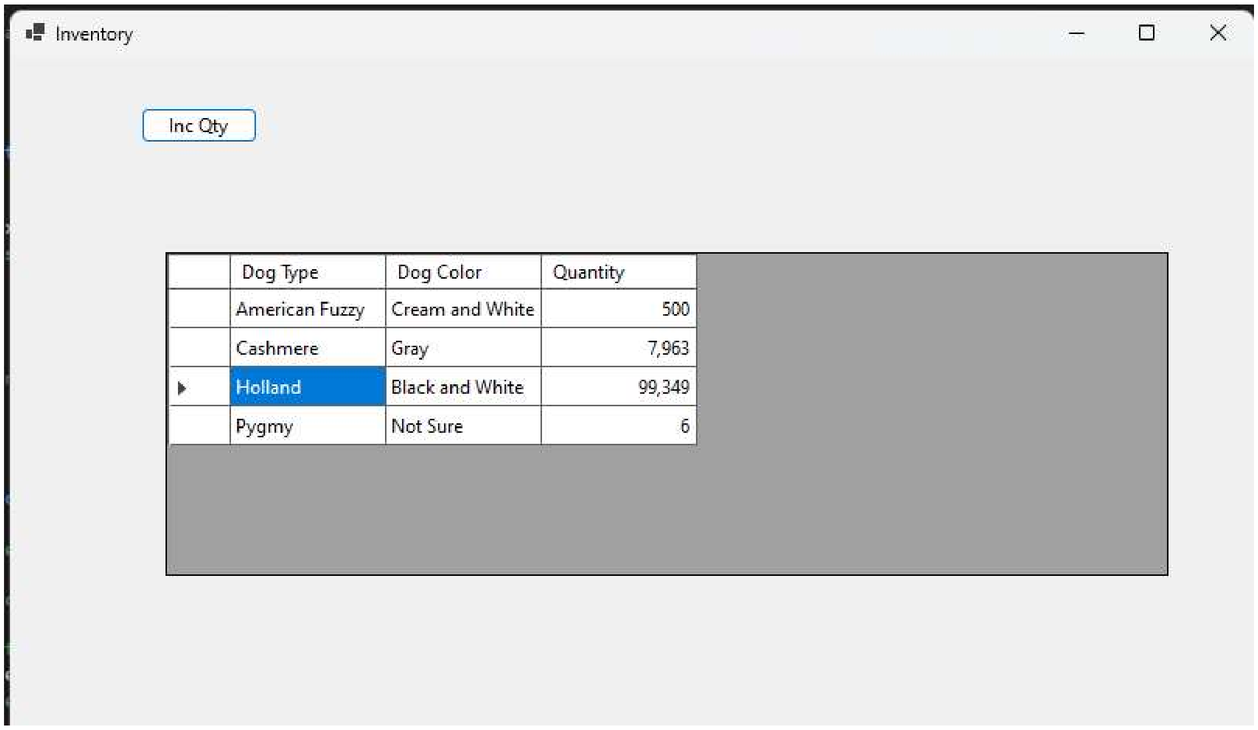
```
/// <summary>
/// Event Handler to Manage button to increment quantity
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnIncQty_ClickEventHandler(object sender, EventArgs e)
{
    // Make sure the logic is not in the presentation layer so
    // inc qty in Inventory class.
    // Instantiate the inventory class so we can use the inc qty method
    Inventory incQty = new Inventory();
    // Invoke this method to inc qty and get the master List back
    invItems = incQty.IncQtyValue(invItems, SelectedGridIndex);
    // Since the List contains the master Inventory
    // Refresh the data in the Data Grid View
    // Since we have already bound the List to the Data Grid View
    gvInv.Refresh();

}
```

Figure 39: Click Event Handler to Inc Qty

m. Run the app and verify the new button functions correctly as shown in Figure 40. Your screen may look different based on the text file provided by the instructor.

n. Verify the qty increments in the selected inventory row.

o. Verify the columns are formatted correctly.



	Dog Type	Dog Color	Quantity
	American Fuzzy	Cream and White	500
	Cashmere	Gray	7,963
▶	Holland	Black and White	99,349
	Pygmy	Not Sure	6

Figure 40: Final Result

10. Refactor the process used to save updated inventory to the text file.
    a. Read Chapter 5 Section 6, "Using Files for Data Storage," in the textbook and learn how to write the updated inventory back into the text file.
        - File Access Methods
        - Writing Data to a File with a StreamWriter Object
        - Writing Data with the Write Method
        - Handling File-Related Exceptions
        - Appending Data to an Existing File.

11. Submit the Activity as described in the digital classroom.

Part 2

Tic Tac Toe

Overview

Tic Tac Toe: Implement Programming Problem 8 is found in Chapter 7 of the textbook.

Submit the Activity as described in the digital classroom.