Class Assignment – Yahtzee
Part D

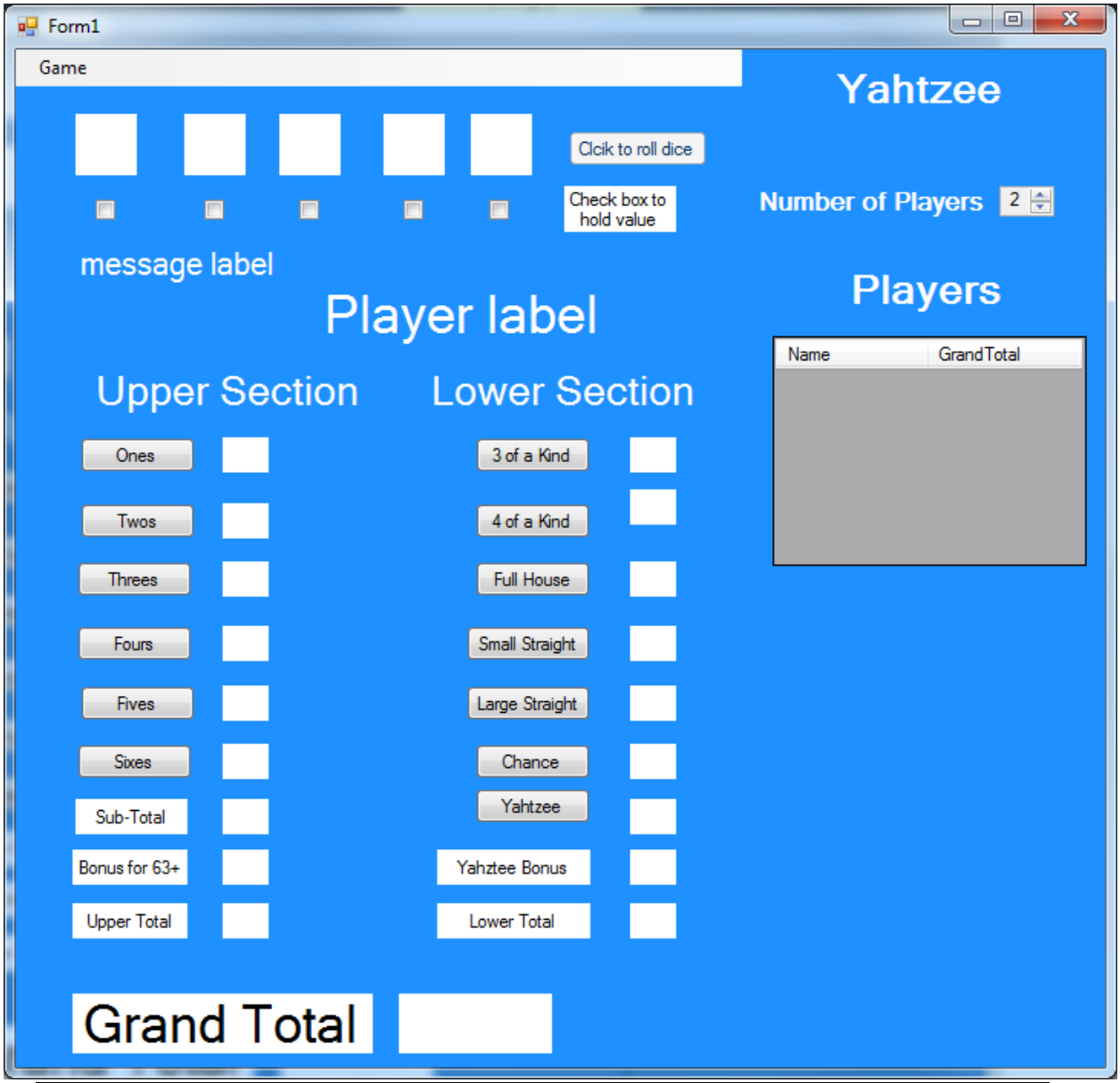
Due Date: 7th June, 2016 Weighting: 40%

The Objective of Part D

This objective of part D of the assignment is to add the additional classes that will provide the game logic to the assignment.

The game logic in Part D `will not implement the rules for **Yahtzee Bonus and Yahtzee Jokers.** We will refer to this version of the game as the **basic version**. It is expected that most student will implement the basic version to varying levels of completeness. A later specification will implement the additional rules.

Adding Controls to Panel2



As can be seen in the previous screenshot, Panel 2 has three (3) **Labels**, a **Numeric Up-Down** control and a **DataGridView** control. Place these controls onto the **Panel2** region of **Form1.**

You will need to add a **set block** to the **Property** of **Name** in the **Player class** as follows:

```
set {name = value;}
```

Add the following directive to **Game.cs;** `using System.ComponentModel;`

In Part C, **players** in the **Game class** was specified as an array of **Players**. For the **DataGridView** control to function we need to replace the array declaration of **players** with

```
private BindingList<Player> players;
```

and **players** is instantiated by the statement

```
players = new BindingList<Player>();
```

You will also need to add the following **Property** for **players**

```
public BindingList<Player> Players {
     get {
         return players;
     }
 }
```

A **BindingList** is a type of **List** with the additional functionality that when used with a **DataGridView** control, any update to the **BindingList** object is reflected in the **DataGridView** and any changes made to the editable fields of the **DataGridView** the corresponding data of the **BindingList** object are updated.

A **List** is a dynamic array that can grow to any size as elements are added. Elements are added to the end of the list via the **Add(object)** method, for example

```
players.Add(player); //where player is a Player object
```

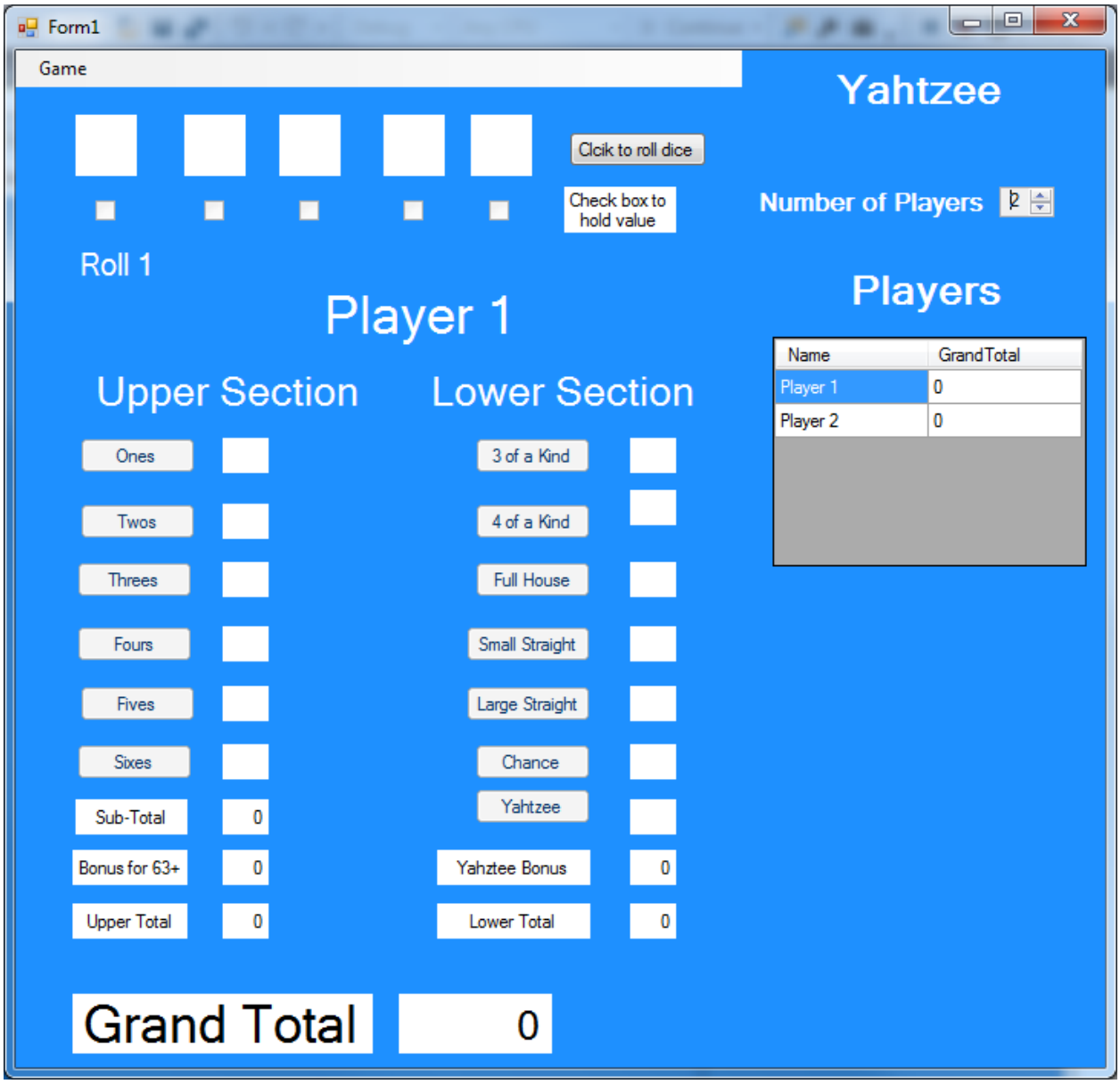
The number of elements in a List is accessed via the **Property Count**: `players.Count`

List elements are accessed by their index in the same way as array elements.

Until you are sure that your game plays correctly for two players, do not implement the event handler for the **Numeric Up-Down** control.

If you unfamiliar with a **DataGridView** control you will need to work through **Activity 4** in **Worksheet 8** for a quick introduction and all that you need to know for implementing the **DataGridView** in this assignment.

Once you have setup the **DataGridView** control and made the changes discussed above at the start of the game your GUI should look similar to the following screenshot.



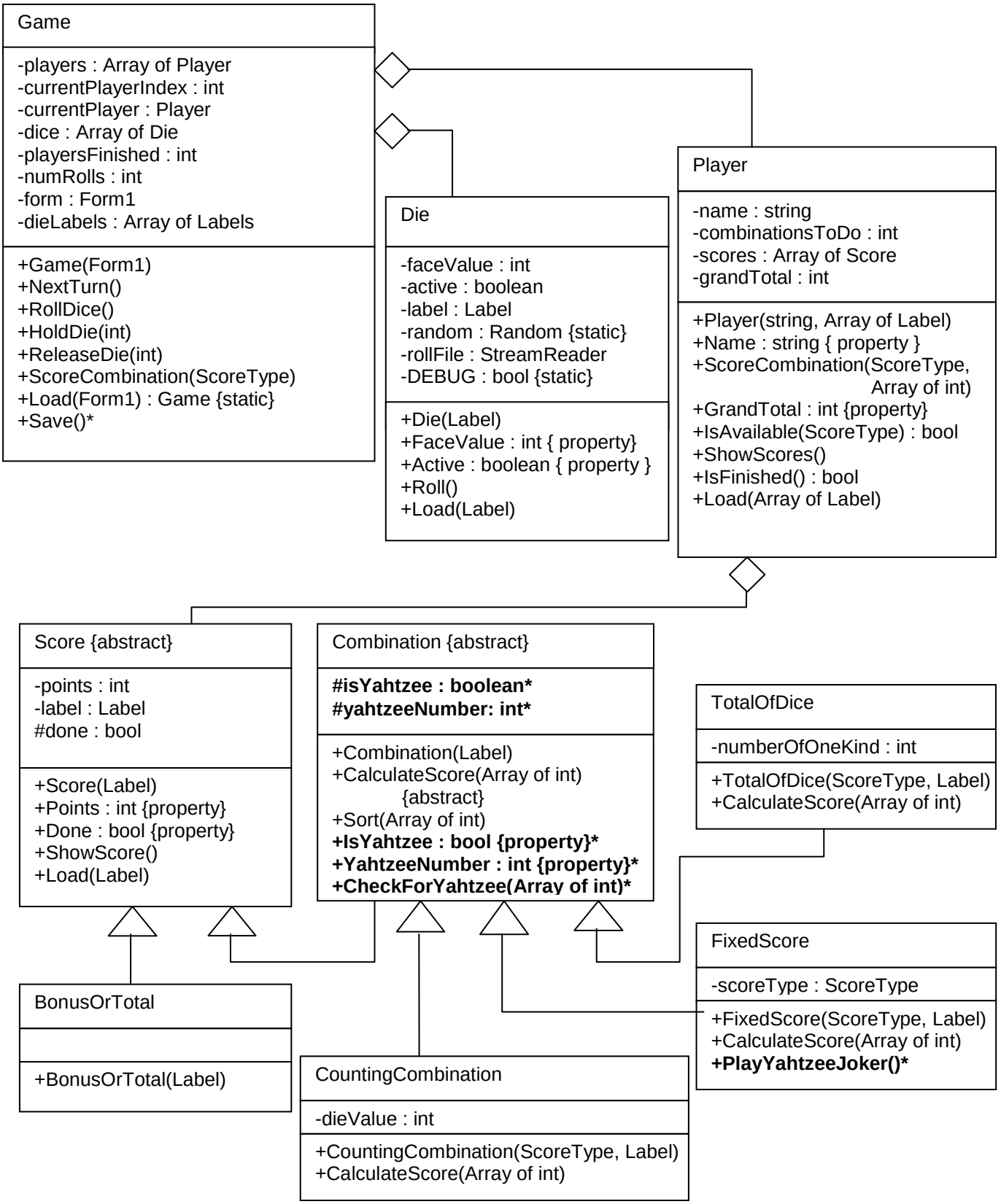
After you have added the event handler for the **Numeric Up-Down** control, as the last activity of this part, whenever the user changes the number of players then the **DataGridView** will show that number of players.

UML Partial System Diagram

The UML diagram on the next page shows the classes which provide the logic of the game. There are nine (9) classes shown in the UML diagram. The details shown in the class diagram give the public methods and the essential private instance variables for each class.

The **triangle shaped arrowhead** indicates a **subclass** relationship. For example, the **Combination class** is a subclass of the **Score class**. Another example, **TotalOfDice class** is a subclass of the **Combination class**.

You will need to declare constants in some of these classes as well as private helper methods. But you MUST use this design when implementing your project.
A brief explanation of the various subclasses of **Score** and **Combination** classes and their instance variables and methods follows.



NOTE: The **bolded** instance variables and method (also denoted with an **asterisk (*)**) are only needed for the advanced version of the game; these variables and methods do not need to be added in part D.

+ : public; - : private; # : protected

As advised in Part B and C, start by adding instance variables and method headings only for each and every class. Any class which has a **Label** object as an instance variable or as a parameter will need to add the following **using** directive: `using System.Windows.Forms;`

Do not attempt to write the bodies of any methods until a skeleton of each class exists and the project compiles without error.

Combination class

An abstract class which is a subclass of **Score.** It represents single dice combination.

Instance Variables

The two instance variables will only be used in the advanced version of the game. They have no purpose in the basic version of the game played in Part D.

Methods

Combination(Label) is a constructor which only needs to pass **Label** parameter to its parent's constructor where the **Label** object represents a **Score** on the GUI.

CalculateScore(Array of int) is an abstract method which must be implemented by all the subclasses of this class. Each subclass will know how to calculate its own score. The parameter to this method is the dice values that are to be used when calculating the score.

The **Sort** method is a **"service"** method. This method offers some functionality that may be used by **Combination** subclasses. This method will sort an array of integers into ascending order. This method can call the Array System Sort method or if you are interested you can implement your own sort algorithm.

IsYahtzee, YahtzeeNumber and CheckForYahtzee are for the advanced version of the game, so need not be included in the basic version.

CountingCombination class

This is a subclass of the **`Combination`** class. It represents a scoring combination that counts the number of a single **die** value in the five dice. These combinations are the six scoring combinations in the **Upper Section** on the GUI.

Instance Variable

dieValue is the die value that this combination will count to calculate the score for this combination.

Methods

The **CountingCombination** constructor has two parameters. The first is the **ScoreType** that represents this scoring combination. From the **ScoreType**, the **dieValue** that this combination needs to count can be determined. The second parameter is the **Label** that represents this **Score object** on the GUI.

CalculateScore will calculate the score for this combination given the dice values (the parameter). To calculate the score for this **CountingCombination**, you will count how many of the dice values match the **dieValue** for this **CountingCombination** and then multiply that count by the **dieValue**.

FixedScore class

This is a subclass of the **Combination** class. It represents all the scoring combinations that have a fixed number as their score. This includes the **Small Straight**, **Large Straight**, **Full House** and **Yahtzee** scoring combinations.

Instance Variable

scoreType is the **ScoreType** value representing this dice combination.

Methods

The **FixedScore** constructor has two parameters. The first is the **ScoreType** value for this scoring combination and the second is the **Label** representing this **score** on the GUI.

CalculateScore(Array of int) will calculate the score for this **Combination** given the dice values. Each of the possible fixed **scoreType** have a constant value as their score. This method will check if the dice values are valid for the particular scoring combination. If they are, then the points value for that combination will be awarded. If they are not, then a points value of zero will be given. **Hint:** It is easier to calculate if the required combination has been achieved if the dice values are sorted.

PlayYahtzeeJoker is part of the advanced game implementation and need not be implemented in the basic version.

TotalOfDice class

This is a subclass of the **Combination** class. It represents all the scoring combinations where the score is the total of the dice face values. These include the **"3 of a Kind"**, **"4 of a Kind"** and **Chance** combinations.

Instance Variables

numberOfOneKind has how many of the same die value must be rolled to score any points for this combination. Note: **numberOfOneKind** will be zero for the **Chance** combination, as this combination allows ANY combination of dice values.

Methods

TotalOfDice constructor will use the **ScoreType** parameter to determine what the value of **numberOfAKind** will be for this combination. The **Label** parameter is required by the parent constructor.

CalculateScore(Array of int) as in the previous two classes, will calculate how many points will be awarded for this combination, given the dice values. The score for all of the **TotalOfDice** combinations will be the sum of the five dice face values. This score will only be awarded if the required number of dice have the same die value.

For the **Chance** combination, no checking is required, and the score will always be the sum of the dice values. For the **"3 of a Kind"** and **"4 of a Kind"** combinations, there must be 3 or 4 of the dice with the same value for the points to be awarded, otherwise the score will be zero. **Hint:** it will be easier to check for these two combinations if the dice values are sorted.

BonusOrTotal class

This is a subclass of the **Score** class. It represents a score for the player which is either one of the bonuses or a sub-total or total score of a Section.

Method

BonusOrTotal(Label) is a constructor that calls the parent's constructor.

Back to completing Player class of Part C

The constructor of **Player** has two parameters, the first is the **name** of this player; the second is the array of **Label** representing the score totals on the GUI. This constructor is called from **Game** class to initialise each individual player element in its **players BindingList.**

Each element of the array **scores** needs to be instantiated using the corresponding **scoreTotals** label of **Form1**. **Hint:** Use a for loop to iterate over **ScoreType** from **ScoreType.Ones** to **ScoreType.GrandTotal** and within the loop body use a switch on **ScoreType** to call the correct subclass constructor, eg **CountingCombination, TotalOfDice, FixedScore** or **BonusOrTotal**.

ShowScores() displays all the scores for this player on their associated **ScoreTotals** labels.

ScoreCombination(…) calculates the score for a specified scoring combination (the **ScoreType** parameter), given the five face values of the dice (the array of int parameter).

The **ScoreType** is used to determine which **CalculateScore method** will be called. The **calculated score** then needs to be added to the correct **Section Total** and in case of a **CountingCombination** to the **Sub-Total** as well.

In order for the **DataGridView** to show a player's grand total after a score has been calculated the following method needs to be added to **Form1.cs**

```
private void UpdatePlayersDataGridView() {
     game.Players.ResetBindings();
 }
```

This method needs to be called from within **Form1.cs** as the last statement in the event handler of the **scoreButtons**.

Assignment Checkpoint

If you have implemented the sub-classes correctly and completed the implementation of the **Player** class your program should be able to play Yahtzee for two players and their scores being displayed for the correct scoring combination, with section totals and grand totals showing the correct values for the stage of the game. and the **DataGridView** showing the running total for each player.

You can implement the event handler for the **Numeric Up-Down** control, for a maximum of 6 players and a minimum of 1 player.

In week 13 you need to have part D completed to allow time to finish the remaining part of the assignment which will involve adding supplied code fragments to your classes as well as attempting to implement the rules for **Yahtzee Bonus and Yahtzee Jokers.**