Programming with C#

# Generic Collections Practice

1. Start Microsoft Visual Studio 2013 if it is not already running.
2. Open the Cards project, which is located in the \Microsoft Press\Visual CSharp Step By Step\Chapter 18\Windows *X*\Cards folder in your Documents folder.
3. This project contains an updated version of the project from the Arrays practice that deals hands of cards by using arrays. The *PlayingCard* class is modified to expose the value and suit of a card as read-only properties. In the Chapter 18 folder, locate either the Windows 7 or Windows 8 folder.
4. Display the Pack.cs file in the Code and Text Editor window. Add the following *using* directive to the top of the file:
5. **using System.Collections.Generic;**
6. In the *Pack* class, change the definition of the *cardPack* two-dimensional array to a *Dictionary<Suit, List< PlayingCard>>* object, as shown here in bold:

class Pack  
{   
 ...   
 **private Dictionary<Suit, List<PlayingCard>> cardPack;** ...  
}

1. The original application used a two-dimensional array for representing a pack of cards. This code replaces the array with a *Dictionary*, where the key specifies the suit, and the value is a list of cards in that suit.
2. Locate the *Pack* constructor. Modify the first statement in this constructor to instantiate the *cardPack* variable as a new *Dictionary* collection rather than an array, as shown here in bold:

public Pack()  
{   
 **this.cardPack = new Dictionary<Suit, List<PlayingCard>>(NumSuits);** ...  
}

1. Although a *Dictionary* collection will resize itself automatically as items are added, if the col­lection is unlikely to change in size, you can specify an initial size when you instantiate it. This helps to optimize the memory allocation, although the *Dictionary* collection can still grow if this size is exceeded. In this case, the *Dictionary* collection will contain a collection of four lists (one list for each suit), so it is allocated space for four items (*NumSuits* is a constant with the value 4).
2. In the outer *for* loop, declare a *List<PlayingCard>* collection object called *cardsInSuit* that is big enough to hold the number of cards in each suit (use the *CardsPerSuit* constant), as fol­lows in bold:

public Pack()  
{   
 this.cardPack = new Dictionary<Suit, List<PlayingCard>>(NumSuits);   
 for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)   
 {   
 **List<PlayingCard> cardsInSuit = new List<PlayingCard>(CardsPerSuit);** for (Value value = Value.Two; value <= Value.Ace; value++)   
 {   
 ...   
 }   
 }  
}

1. Change the code in the inner *for* loop to add new *PlayingCard* objects to this collection rather than the array, as shown in bold in the following code:

for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)   
 {   
 List<PlayingCard> cardsInSuit = new List<PlayingCard>(CardsPerSuit);   
 for (Value value = Value.Two; value <= Value.Ace; value++)   
 {   
 **cardsInSuit.Add(new PlayingCard(suit, value));** }   
 }

1. After the inner *for* loop, add the *List* object to the *cardPack Dictionary* collection, specifying the value of the *suit* variable as the key to this item:

for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)   
{   
 List<PlayingCard> cardsInSuit = new List<PlayingCard>(CardsPerSuit);   
 for (Value value = Value.Two; value <= Value.Ace; value++)   
 {   
 **cardsInSuit.Add(new PlayingCard(suit, value));** }   
 **this.cardPack.Add(suit, cardsInSuit);** }

1. Find the *DealCardFromPack* method.

This method picks a card at random from the pack, removes the card from the pack, and returns this card. The logic for selecting the card does not require any changes, but the state­ments at the end of the method that retrieve the card from the array must be updated to use the *Dictionary* collection, instead. Additionally, the code that removes the card from the array (it has now been dealt) must be modified; you need to search for the card in the list and then remove it from the list. To locate the card, use the *Find* method and specify a predicate that finds a card with the matching value. The parameter to the predicate should be a *PlayingCard* object (the list contains *PlayingCard* items).

1. The updated statements occur after the closing brace of the second *while* loop, as shown in bold in the following code:

public PlayingCard DealCardFromPack()   
{   
 Suit suit = (Suit)randomCardSelector.Next(NumSuits);   
 while (this.IsSuitEmpty(suit))   
 {   
 suit = (Suit)randomCardSelector.Next(NumSuits);   
 }

Value value = (Value)randomCardSelector.Next(CardsPerSuit);   
 while (this.IsCardAlreadyDealt(suit, value))   
 {   
 value = (Value)randomCardSelector.Next(CardsPerSuit);   
 }   
  
 **List<PlayingCard> cardsInSuit = this.cardPack[suit];   
 PlayingCard card = cardsInSuit.Find(c => c.CardValue == value);  
 cardsInSuit.Remove(card);** return card;  
 }

1. Locate the *IsCardAlreadyDealt* method.
2. This method determines whether a card has already been dealt by checking whether the corresponding element in the array has been set to *null*. You need to modify this method to determine whether a card with the specified value is present in the list for the suit in the *cardPack Dictionary* collection.
3. To determine whether an item exists in a *List<T>* collection, you use the *Exists* method. This method is similar to *Find* inasmuch as it takes a predicate as its argument. The predicate is passed each item from the collection in turn, and it should return true if the item matches some specified criteria, and false otherwise. In this case, the *List<T>* collection holds *PlayingCard* objects, and the criteria for the *Exists* predicate should return true if it is passed a *PlayingCard* item with a suit and value that matches the parameters passed to the *IsCardAlreadyDealt* method.
4. Update the method, as shown in the following example, in bold:

private bool IsCardAlreadyDealt(Suit suit, Value value)  
{   
 **List<PlayingCard> cardsInSuit = this.cardPack[suit];   
 return (!cardsInSuit.Exists(c => c.CardSuit == suit && c.CardValue == value));**}

1. Display the Hand.cs file in the Code and Text Editor window. Add the following *using* directive to the list at the top of the file:

**using System.Collections.Generic;**

1. The *Hand* class currently uses an array called *cards* to hold the playing cards for the hand. Modify the definition of the *cards* variable to be a *List<PlayingCard>* collection, as shown here in bold:

class Hand  
{   
 public const int HandSize = 13;   
 **private List<PlayingCard> cards = new List<PlayingCard>(HandSize);** ...  
}

1. Find the *AddCardToHand* method.
2. This method currently checks to see whether the hand is full; if it is not, it adds the card provided as the parameter to the *cards* array at the index specified by the *playingCardCount* variable.
3. Update this method to use the *Add* method of the *List<PlayingCard>* collection, instead.
4. This change also removes the need to explicitly keep track of how many cards the collection holds because you can use the *Count* property of the *cards* collection, instead. Therefore, remove the *playingCardCount* variable from the class and modify the *if* statement that checks whether the hand is full to reference the *Count* property of the *cards* collection.
5. The completed method should look like this, with the changes highlighted in bold:

public void AddCardToHand(PlayingCard cardDealt)  
{   
 if (**this.cards.Count** >= HandSize)   
 {   
 throw new ArgumentException("Too many cards");   
 }   
 **this.cards.Add(cardDealt);  
}**

1. On the Debug menu, click Start Debugging to build and run the application.
2. When the Card Game form appears, click Deal.

**Note** Remember that in the Windows Store apps version of this application, the Deal button is located on the app bar.

1. Verify that the cards are dealt and that the populated hands appear as before. Click Deal again to generate another random set of hands.
2. Stop the application and close Visual Studio.