K – Looping Structures and Collections

Code Samples - Documentation

# Examples

The following examples are used to illustrate this topic.

1. **Math** – The Math class is expanded to produce a collection of Integers for the sequence of Fibonacci numbers.
2. **PhoneBook** – The PhoneBook class provides simple management of a collection of PhoneNumber objects. The PhoneBook allows for adding and retrieving phone numbers; it supports search for phone numbers by the telephone number or by a last name.
3. **ClassList** – The ClassList example maintains a list of students for a particular course by offering methods to add and remove students from a course. In addition, the ClassList ensures that students are not added to the course twice (based on the student’s Id).
4. **MultipleChoiceMarker** – This class is used for marking multiple choice exams. It takes a collection of MultipleChoice objects as the answer key when it is first created. It provides a method to mark the student’s answers.
5. **BankStatement** – This class represents a bank statement for a BankAccount for a given month. The statement allows BankTransaction objects to be added, and performs deposits and withdrawals on the BankAccount. The statement reports the starting and ending balance and also summarizes the total amount deposited and withdrawn for the month.
6. **DeckOfCards** –The DeckOfCards class represents a complete deck of cards. When the deck is first created, a card is created for each suit. The DeckOfCards supports a method to draw a card.

# Overview

This topic provides further examples of looping logic, but involves the use of *collections*. All of these samples make use of the List<T> class to maintain a collection or group of objects.

The List<T> class is a *Generic* class, meaning that the item in the angled brackets - <T> - is a placeholder for the name of the actual class that is being managed in the List. For example, to have a list of Integers we would declare that as List<Integer>. Likewise, if we wanted a list of Student objects, it would be declared as List<Student>.

The List<T> class supports a number of methods and properties for working with the collection.

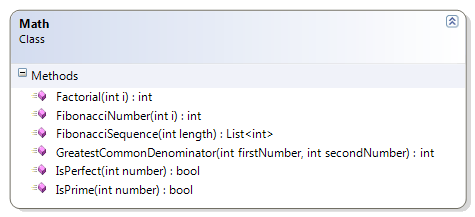
* **Add()** – Used to add an item to the collection
* **[***index***]** – Used to retrieve an item in the collection
* **Remove()** – Used to remove an item from the collection
* **Count** – Identifies the number of items in the collection

# Math

## Problem Statement

This sample leverages the existing calculation of a number in the Fibonacci sequence to generate the sequence itself.

* **FibonacciSequence(Length : Integer)** – This method generates the sequence of Fibonacci numbers to a specific length in the sequence.



## Code Solution

public static List<int> FibonacciSequence(int length)

{

List<int> fib = new List<int>();

if (length >= 1)

{

for (int index = 0; index < length; index++)

fib.Add(FibonacciNumber(index + 1));

}

return fib;

}

# PhoneBook

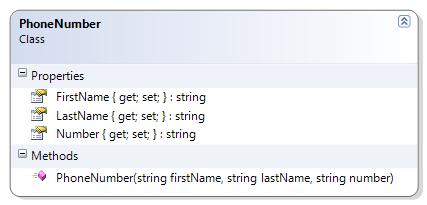
## ProblemStatement

The PhoneBook class provides simple management of a collection of PhoneNumber objects. The PhoneBook allows for adding and retrieving phone numbers; it supports search for phone numbers by the telephone number or by a last name.

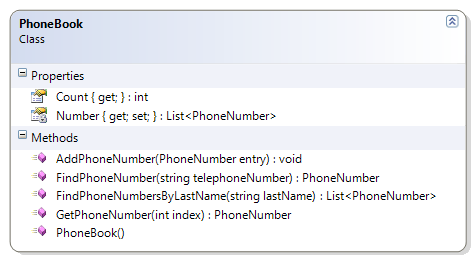
* **Constructor** – Create a new List<PhoneNumber> for the numbers field.
* **AddPhoneNumber(Entry : PhoneNumber)** – Add a PhoneNumber to the collection. Ensure that the PhoneNumber exists (is not *null*) before adding; if the PhoneNumber is null, throw an exception.
* **GetPhoneNumber(Index : Integer) : PhoneNumber** – Get an item from a particular position in the collection.
* **GetCount() : Integer** – Get the size of the collection, which is a count of how many PhoneNumber objects are in the collection.
* **FindPhoneNumber(TelephoneNumber : String) : PhoneNumber** – Look through the collection and return a PhoneNumber with a matching telephone number. If none is found, return *null*.
* **FindPhoneNumbersByLastName(LastName : String) : List<PhoneNumber>** - Look through the collection for PhoneNumber objects with a matching last name. Add those objects to a new collection and return the collection of matching PhoneNumbers. If no items are found, return an empty collection.

## Supporting Classes

The PhoneNumber class used in this sample is provided for you. The following class diagram shows the design of this class.



## PhoneBook Class



## Code Solution

public class PhoneBook

{

private List<PhoneNumber> Number { get; set; }

public PhoneBook()

{

Number = new List<PhoneNumber>();

}

public void AddPhoneNumber(PhoneNumber entry)

{

if (entry == null)

throw new System.Exception("The phone number entry cannot be null");

Number.Add(entry);

}

public PhoneNumber GetPhoneNumber(int index)

{

return Number[index];

}

public int Count

{

get

{

return Number.Count;

}

}

public PhoneNumber FindPhoneNumber(string telephoneNumber)

{

PhoneNumber found = null;

foreach (PhoneNumber item in Number)

{

if (item.Number.Equals(telephoneNumber))

{

found = item;

break;

}

}

return found;

}

public List<PhoneNumber> FindPhoneNumbersByLastName(string lastName)

{

List<PhoneNumber> found = new List<PhoneNumber>();

foreach (PhoneNumber item in Number)

{

if (item.LastName.Equals(lastName))

{

found.Add(item);

}

}

return found;

}

}

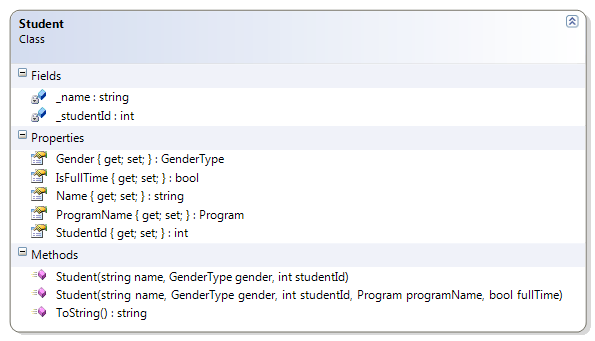
# ClassList

## Problem Statement

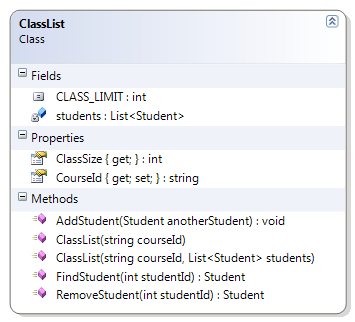
The ClassList example maintains a list of students for a particular course by offering methods to add and remove students from a course. In addition, the ClassList ensures that students are not added to the course twice (based on the student’s Id). The following parts of the ClassList must be coded to complete the solution.

* **Constructor** – Set the course id and the collection of students. Ensure that the supplied arguments are valid
  + CourseId cannot be empty or null, and must be trimmed of leading or trailing spaces
  + The collection object cannot be null and cannot have more students than the constant CLASS\_LIMIT
  + There cannot be any duplicate students in the collection (where a duplicate is defined as two or more Student objects with identical Ids)
* **AddStudent()** – Add the supplied student object to the collection. Ensure that
  + The Student object is not null
  + The class limit is not exceeded
  + The Student object does not already exist in the collection (that is, there are no duplicates allowed, based on the student’s id)
* **FindStudent()** – Search the collection for a Student with a matching Id. If none is found, return null.
* **RemoveStudent()** – Search for a Student with a matching Id; if one is found, remove it from the collection.

## Student



## ClassList



## Code Solution

public class ClassList

{

public const int CLASS\_LIMIT = 25;

public string CourseId { get; private set; }

private List<Student> students;

public ClassList(string courseId, List<Student> students)

{

if (String.IsNullOrEmpty(courseId) || string.IsNullOrEmpty(courseId.Trim()))

throw new Exception("Course Id is required");

if (students == null)

throw new Exception("Students cannot be a null list");

if (students.Count > CLASS\_LIMIT)

throw new Exception("Class Limit Exceeded");

for (int index = 0; index < students.Count - 1; index++)

{

int id = students[index].StudentId;

for (int innerLoop = index + 1; innerLoop < students.Count; innerLoop++)

if (students[innerLoop].StudentId == id)

throw new Exception(

"Duplicate student Ids not allowed in the class list");

}

this.CourseId = courseId.Trim();

this.students = students;

}

public ClassList(string courseId) :

this(courseId, new List<Student>())

{

}

public int ClassSize

{

get

{

return students.Count;

}

}

public void AddStudent(Student anotherStudent)

{

if (anotherStudent == null)

throw new Exception("Cannot add null student");

if (students.Count >= CLASS\_LIMIT)

throw new Exception("Class Limit Exceeded - Cannot add student");

for (int index = 0; index < students.Count - 1; index++)

{

int id = students[index].StudentId;

if (anotherStudent.StudentId == id)

throw new Exception(

"Duplicate student Ids not allowed in the class list");

}

students.Add(anotherStudent);

}

public Student FindStudent(int studentId)

{

Student found = null;

for (int index = 0; index < students.Count && found == null; index++)

if (students[index].StudentId == studentId)

found = students[index];

return found;

}

public Student RemoveStudent(int studentId)

{

Student found = FindStudent(studentId);

if (found != null)

students.Remove(found);

return found;

}

}

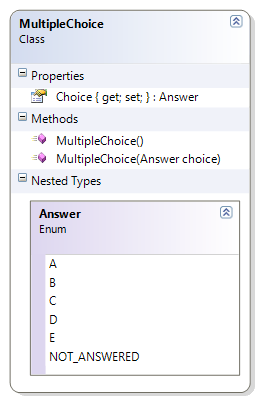
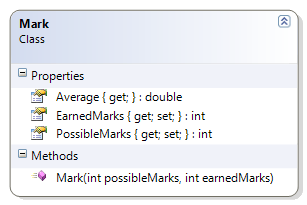
# MultipleChoiceMarker

## Problem Statement

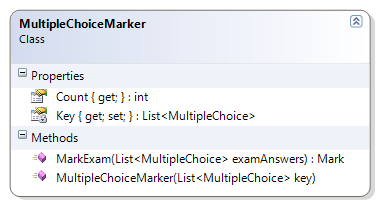
This class is used for marking multiple choice exams. It takes a collection of MultipleChoice objects as the answer key when it is first created. It provides a method to mark the student’s answers. The following methods must be coded to complete the solution.

* **MarkExam()** – This method takes the supplied exam answers and compares them against the answers in the marking key. It then constructs a Mark object, based on the earned marks and the possible marks (each answer is worth one mark). Before marking the exam, the method must ensure that
  + The supplied collection of multiple choice answers is not null
  + The supplied collection of multiple choice answers has the same number of answers as the MultipleChoiceMarker’s answer key

## MultipleChoice and Mark Classes

## MultipleChoiceMarker



## Code Solution

public class MultipleChoiceMarker

{

private List<MultipleChoice> Key { get; set; }

public MultipleChoiceMarker(List<MultipleChoice> key)

{

if (key == null)

throw new Exception("Answer key cannot be null");

this.Key = key;

}

public Mark MarkExam(List<MultipleChoice> examAnswers)

{

if (examAnswers == null)

throw new Exception("Cannot mark null answers");

if (examAnswers.Count != Key.Count)

throw new Exception(

"The number of student answers does not match the number of items in the answer key");

int possible = Key.Count;

int earned = 0;

// Calculate earned marks

for (int index = 0; index < Key.Count; index++)

{

if (examAnswers[index] != null)

if (Key[index].Choice == examAnswers[index].Choice)

earned++;

}

Mark examMark = new Mark(possible, earned);

return examMark;

}

public int Count

{

get

{

return Key.Count;

}

}

}

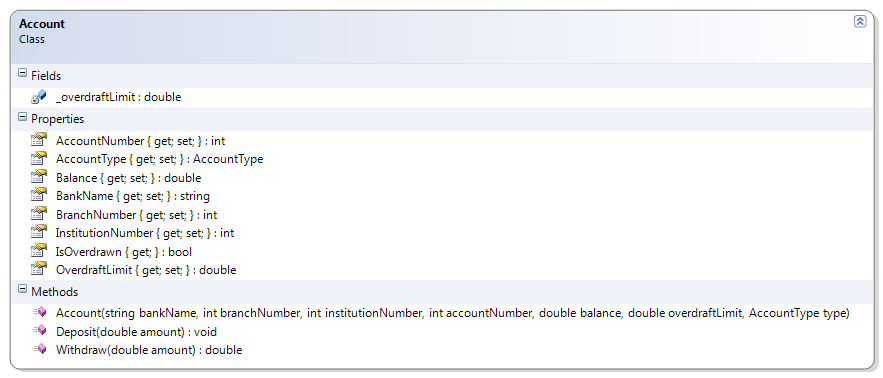
# BankStatement

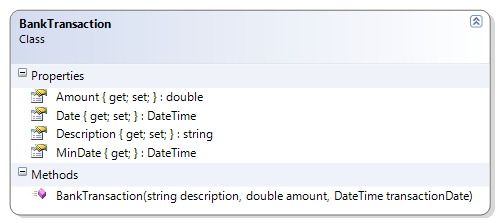
## Problem Statement

This class represents a bank statement for a BankAccount for a given month. The statement allows BankTransaction objects to be added, and performs deposits and withdrawals on the BankAccount. The statement reports the starting and ending balance and also summarizes the total amount deposited and withdrawn for the month. The following methods must be coded to complete the solution.

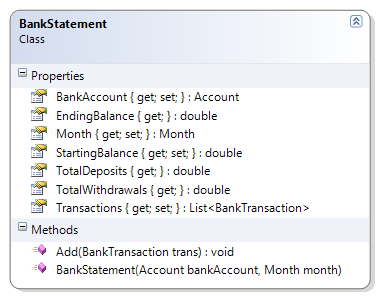
* **GetTotalDeposits()** – Loop through the collection of transactions to total all those transactions with a positive amount.
* **GetTotalWithdrawals()** – Loop through the collection of transactions to total all those transactions with a negative amount.

## Account and BankTransaction





## Bank Statement



## Code Solution

public class BankStatement

{

public Account BankAccount { get; private set; }

public double StartingBalance { get; private set; }

public Month Month { get; private set; }

public List<BankTransaction> Transactions { get; private set; }

public double EndingBalance

{

get

{

return BankAccount.Balance;

}

}

public BankStatement(Account bankAccount, Month month)

{

if (bankAccount == null)

throw new System.Exception("Bank account cannot be null");

this.BankAccount = bankAccount;

this.StartingBalance = bankAccount.Balance;

this.Month = month;

this.Transactions = new List<BankTransaction>();

}

public void Add(BankTransaction trans)

{

if (trans == null)

throw new System.Exception("Cannot add null transactions");

double amount = trans.Amount;

if (amount < 0)

BankAccount.Withdraw(-amount); // Amount is "negated" to make positive

else

BankAccount.Deposit(amount);

Transactions.Add(trans);

}

public double TotalDeposits

{

get

{

double total = 0;

for (int index = 0; index < Transactions.Count; index++)

{

BankTransaction aTransaction = Transactions[index];

if (aTransaction.Amount > 0)

total += aTransaction.Amount;

}

return total;

}

}

public double TotalWithdrawals

{

get

{

double total = 0;

foreach (BankTransaction aTransaction in Transactions)

if (aTransaction.Amount < 0)

total += aTransaction.Amount;

return total;

}

}

}

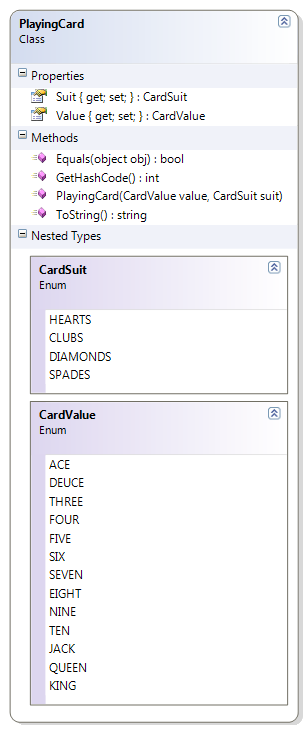
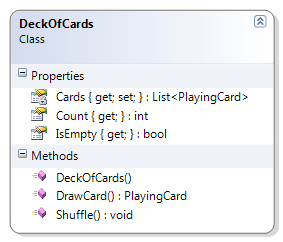
# DeckOfCards

## Problem Statement

The DeckOfCards class represents a complete deck of cards. When the deck is first created, a card is created for each suit. The DeckOfCards supports a method to draw a card. The following methods must be coded to complete the solution.

* **Constructor** – Create all the cards for all the CardSuit values and all the CardValue values.
* **DrawCard()** – Return the card at the “top” of the deck (that is, at position zero). If the deck is empty, return a null.
* **Shuffle()** – Mix up the order of the PlayingCards in the Cards list. Shuffle should work regardless of the number of cards still in the deck.

## PlayingCard and DeckOfCards

## Code Solution

public class DeckOfCards

{

private List<PlayingCard> Cards { get; set; }

public int Count

{ get { return Cards.Count; } }

public bool IsEmpty

{ get { return Count == 0; } }

public DeckOfCards()

{

Cards = new List<PlayingCard>();

foreach (PlayingCard.CardSuit suit in System.Enum.GetValues(typeof(PlayingCard.CardSuit)))

foreach (PlayingCard.CardValue value in System.Enum.GetValues(typeof(PlayingCard.CardValue)))

Cards.Add(new PlayingCard(value, suit));

}

public PlayingCard DrawCard()

{

PlayingCard card = null;

if (Cards.Count != 0)

{

card = Cards[0];

Cards.Remove(card);

}

return card;

}

public void Shuffle()

{

for (int counter = 0; counter < 100; counter++)

{

int index = Rnd.Next(Cards.Count);

PlayingCard card = Cards[0];

Cards.RemoveAt(0);

Cards.Insert(index, card);

}

}

}