L – Arrays – Not Sorted

Code Samples - Documentation

# Examples

The following examples are used to illustrate this topic.

1. **Math** – The Math class is modified to produce the sequence of Fibonacci numbers as an array.
2. **PhoneBook** – The PhoneBook class provides simple management of an array 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. The PhoneBook is initialized with a physical limit for the arrays, and the actual number of entries in the phone book is tracked as phone numbers are added to the phone book.
3. **ClassList** – The ClassList example maintains an array 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 internal array is of a fixed size (ClassList.CLASS\_LIMIT) as well as the logical size of the array. If the ClassList is provided with an array of students in its constructor, that list is “copied” to the internal array of the ClassList object.
4. **MultipleChoiceMarker** – This class is used for marking multiple choice exams. It takes an array 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 The statement reports the starting and ending balance and also summarizes the total amount deposited and withdrawn for the month.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

All of these classes are used to illustrate working with un-sorted arrays.

# Math

## Problem Statement

This sample leverages the existing calculation of a number in the Fibonacci sequence to generate the sequence itself. The sequence is returned as an array of integers.

* **FibonacciSequence(Length : Integer)** – This method generates the sequence of Fibonacci numbers to a specific length in the sequence; the results are returned as an array of integers. If the specified length is negative, throw an exception.

## Code Solution

public static int[] FibonacciSequence(int length)

{

int[] fib = new int[0];

if (length >= 1)

{

fib = new int[length];

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

fib[index] = FibonacciNumber(index + 1);

}

return fib;

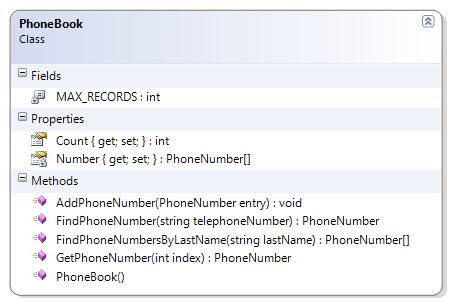
}

# PhoneBook

## ProblemStatement

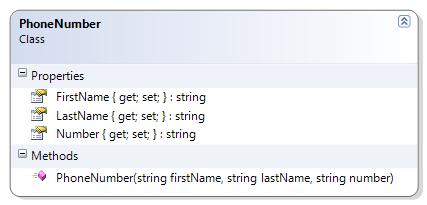
The PhoneBook class provides simple management of an array of PhoneNumber objects. The PhoneBook allows for adding and retrieving phone numbers; it supports search for phone numbers by the telephone number or by a last name. The PhoneBook is initialized with a physical limit for the arrays, and the actual number of entries in the phone book is tracked as phone numbers are added to the phone book.

* **Constructor** – Create a new array of PhoneNumber objects for the numbers field.
* **AddPhoneNumber(Entry : PhoneNumber)** – Add a PhoneNumber to the array. Ensure that the PhoneNumber exists (is not *null*) before adding; if the PhoneNumber is null, throw an exception. If there isn’t any room to add the PhoneNumber, then throw an exception.
* **GetPhoneNumber(Index : Integer) : PhoneNumber** – Get an item from a particular position in the array.
* **Count : Integer** – Get the logical size of the array.
* **FindPhoneNumber(TelephoneNumber : String) : PhoneNumber** – Look through the array and return a PhoneNumber with a matching telephone number. If none is found, return *null*.
* **FindPhoneNumbersByLastName(LastName : String) : PhoneNumber[ ]** - Look through the array for PhoneNumber objects with a matching last name. Add those objects to a new array and return it. If no items are found, return an empty array.



## Supporting Classes

**ArrayFullException** – This class is used to indicate that the array is full and no more items can be added to the array.



## Code Solution

public class PhoneBook

{

private const int MAX\_RECORDS = 1000;

private PhoneNumber[] Number { get; set; }

public int Count { get; private set; }

public PhoneBook()

{

Number = new PhoneNumber[MAX\_RECORDS];

}

public void AddPhoneNumber(PhoneNumber entry)

{

if (entry == null)

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

if (Count == MAX\_RECORDS)

throw new ArrayFullException("The phone book cannot hold any more Phone Numbers");

Number[Count] = entry;

Count++;

}

public PhoneNumber GetPhoneNumber(int index)

{

return Number[index];

}

public PhoneNumber FindPhoneNumber(string telephoneNumber)

{

PhoneNumber found = null;

int foundIndex = -1;

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

{

if (Number[index].Number.Equals(telephoneNumber))

{

foundIndex = index;

break;

}

}

if (foundIndex >= 0)

found = Number[foundIndex];

return found;

}

public PhoneNumber[] FindPhoneNumbersByLastName(string lastName)

{

PhoneNumber[] found = new PhoneNumber[MAX\_RECORDS];

// Find the numbers

int foundCounter = 0;

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

{

PhoneNumber item = Number[index];

if (item.LastName.Equals(lastName))

{

found[foundCounter] = item;

foundCounter++;

}

}

// Trim the array

PhoneNumber[] foundTrimmed = new PhoneNumber[foundCounter];

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

{

foundTrimmed[index] = found[index];

}

return foundTrimmed;

}

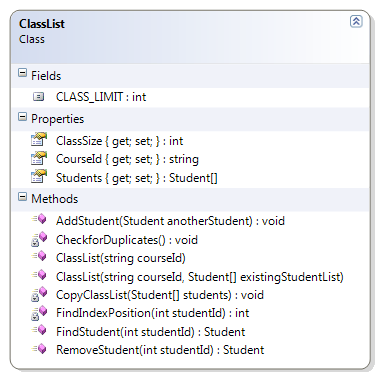
}

# ClassList

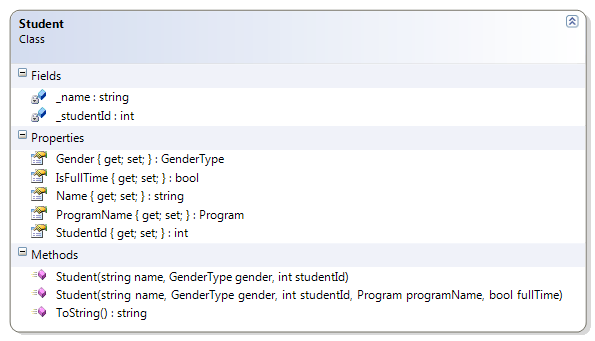
## Problem Statement

The ClassList example maintains an array 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 internal array is of a fixed size (ClassList.CLASS\_LIMIT) as well as the logical size of the array. If the ClassList is provided with an array of students in its constructor, that list is “copied” to the internal array of the ClassList object.

* **Constructor** – Set the course id and the array 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 array of students cannot be null and cannot have more students than the constant CLASS\_LIMIT
  + There cannot be any duplicate students in the array (where a duplicate is defined as two or more Student objects with identical Ids)
* **AddStudent()** – Add the supplied student object to the array. Ensure that
  + The Student object is not null
  + The class limit is not exceeded
  + The Student object does not already exist in the array (that is, there are no duplicates allowed, based on the student’s id)
* **FindStudent()** – Search the array 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 array.



## Supporting Classes



## Code Solution

public class ClassList

{

public const int CLASS\_LIMIT = 25;

public string CourseId { get; private set; }

public Student[] Students { get; private set; }

public int ClassSize { get; private set; }

public ClassList(string courseId) :

this(courseId, new Student[CLASS\_LIMIT])

{

}

public ClassList(string courseId, Student[] existingStudentList)

{

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

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

if (existingStudentList == null)

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

if (existingStudentList.Length > CLASS\_LIMIT)

throw new Exception("Class Limit Exceeded");

CopyClassList(existingStudentList);

CheckforDuplicates();

this.CourseId = courseId.Trim();

}

private void CopyClassList(Student[] students)

{

ClassSize = 0;

this.Students = new Student[CLASS\_LIMIT];

for (int i = 0; i < students.Length; i++)

{

if (students[i] != null)

{

this.Students[ClassSize] = students[i];

ClassSize++;

}

else

break; // Treat the first null instance as a "flag" indicating the end of actual student objects in the array

}

}

private void CheckforDuplicates()

{

for (int index = 0; index < ClassSize - 1; index++)

{

int id = Students[index].StudentId;

for (int innerLoop = index + 1; innerLoop < ClassSize; innerLoop++)

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

throw new Exception(

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

}

}

public void AddStudent(Student anotherStudent)

{

if (anotherStudent == null)

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

if (ClassSize >= CLASS\_LIMIT)

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

for (int index = 0; index < ClassSize - 1; index++)

{

int id = Students[index].StudentId;

if (anotherStudent.StudentId == id)

throw new Exception(

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

}

Students[ClassSize] = anotherStudent;

ClassSize++;

}

public Student FindStudent(int studentId)

{

Student found = null;

int foundPosition = FindIndexPosition(studentId);

if (foundPosition >= 0)

found = Students[foundPosition];

return found;

}

private int FindIndexPosition(int studentId)

{

int foundPosition = -1;

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

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

foundPosition = index;

return foundPosition;

}

public Student RemoveStudent(int studentId)

{

Student found = null;

int foundPosition = FindIndexPosition(studentId);

if (foundPosition >= 0)

{

found = Students[foundPosition];

Students[foundPosition] = Students[ClassSize - 1];

ClassSize--;

}

return found;

}

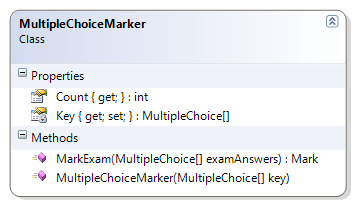
}

# MultipleChoiceMarker

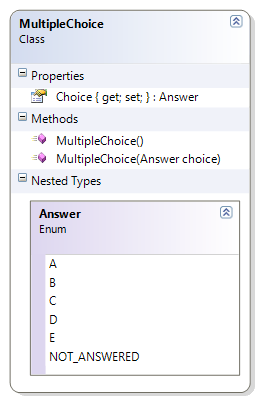
## Problem Statement

This class is used for marking multiple choice exams. It takes an array 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 array of multiple choice answers is not null
  + The supplied array of multiple choice answers has the same number of answers as the MultipleChoiceMarker’s answer key



## Supporting Classes



## Code Solution

public Mark MarkExam(MultipleChoice[] examAnswers)

{

if (examAnswers == null)

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

if (examAnswers.Length != Key.Length)

throw new Exception(

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

int possible = Key.Length;

int earned = 0;

// Calculate earned marks

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

{

if (examAnswers[index] != null)

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

earned++;

}

Mark examMark = new Mark(possible, earned);

return examMark;

}

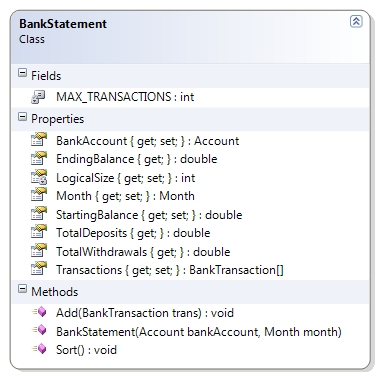
# BankStatement

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.

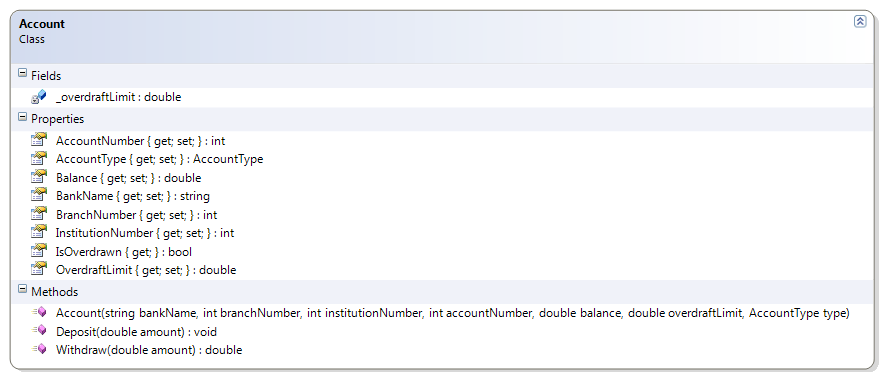
## Problem Statement

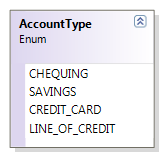
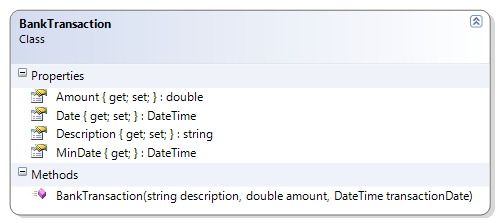
This class represents a bank statement for a BankAccount for a given month. The statement allows BankTransaction objects to be added, and performs deposits and withdrawals on the BankAccount. The statement reports the starting and ending balance and also summarizes the total amount deposited and withdrawn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

* **Add()** – Add a BankTransaction to the array of transactions. If the BankTransaction is null, throw an exception. Otherwise, process the transaction as either a deposit or a withdrawal, depending on if the amount is positive or negative, and then add it to the array of transactions.
* **GetTotalDeposits()** – Loop through the array of transactions to total all those transactions with a positive amount.
* **GetTotalWithdrawals()** – Loop through the array of transactions to total all those transactions with a negative amount.
* **Sort()** – Sorts the array of transactions by date.



## Supporting Classes





## Code Solution

public class BankStatement

{

private const int MAX\_TRANSACTIONS = 500;

private int LogicalSize { get; set; }

public Account BankAccount { get; private set; }

public double StartingBalance { get; private set; }

public Month Month { get; private set; }

public BankTransaction[] Transactions { get; private set; }

public double EndingBalance

{

get

{

return BankAccount.Balance;

}

}

public BankStatement(Account bankAccount, Month month)

{

if (bankAccount == null)

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

this.BankAccount = bankAccount;

this.StartingBalance = bankAccount.Balance;

this.Month = month;

this.Transactions = new BankTransaction[MAX\_TRANSACTIONS];

LogicalSize = 0;

}

public void Add(BankTransaction trans)

{

if (trans == null)

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

if (LogicalSize == MAX\_TRANSACTIONS)

throw new ArrayFullException("The bank statement cannot accept further transactions; the array is full");

double amount = trans.Amount;

if (amount < 0)

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

else

BankAccount.Deposit(amount);

Transactions[LogicalSize] = trans;

LogicalSize++;

}

public double TotalDeposits

{

get

{

double total = 0;

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

{

BankTransaction aTransaction = Transactions[index];

if (aTransaction.Amount > 0)

total += aTransaction.Amount;

}

return total;

}

}

public double TotalWithdrawals

{

get

{

double total = 0;

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

{

BankTransaction aTransaction = Transactions[index];

if (aTransaction.Amount < 0)

total += aTransaction.Amount;

}

return total;

}

}

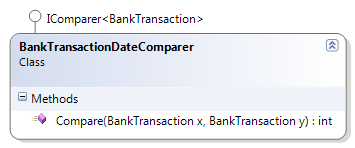
public void Sort()

{

Array.Sort<BankTransaction>(Transactions, new BankTransactionDateComparer());

}

}



public class BankTransactionDateComparer : IComparer<BankTransaction>

{

#region IComparer<BankTransaction> Members

public int Compare(BankTransaction x, BankTransaction y)

{

return x.Date.CompareTo(y.Date);

}

#endregion

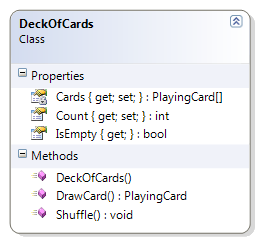
}

# 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.
* **IsEmpty** – Returns true if the logical size of the deck of cards has reached zero, otherwise it returns false.



## Supporting Classes

The PlayingCard class, along with its two enumerated types CardSuit and CardValue, are the basis for the DeckOfCards class.

## Code Solution

public class DeckOfCards

{

private PlayingCard[] Cards { get; set; }

public int Count { get; private set; }

public bool IsEmpty { get { return Count == 0; } }

public DeckOfCards()

{

Cards = new PlayingCard[52];

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

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

{

Cards[Count] = new PlayingCard(value, suit);

Count++;

}

}

public PlayingCard DrawCard()

{

PlayingCard card = null;

if (Count != 0)

{

// get the first card

card = Cards[0];

// "pull" the other items up in the array

for (int i = 0; i < Count - 1; i++)

{

Cards[i] = Cards[i + 1];

}

// "clear" the last element & adjust the count

Cards[Count - 1] = null;

Count--;

}

return card;

}

public void Shuffle()

{

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

{

int index = Rnd.Next(Count);

PlayingCard card = Cards[0];

Cards[0] = Cards[index];

Cards[index] = card;

}

}

}