Projecto de Sistemas Industriais

C#.NET

Parte II

Sumário – C#.NET

- Estruturas
- Operador Overloading
- Interfaces
- Vectores e Classe Collections

Exemplos retirados do livro:
Programming C#, Jesse Liberty – O'Reilly

C# - Estruturas

- Uma estrutura consiste num tipo de dados definido pelo utilizador.
- As estruturas são semelhantes às classes, pois possuem construtor, propriedades, métodos, operadores, etc..
- Diferem das classes na medida em que não possuem destrutores nem permitem herança.
- Geralmente opta-se pelo uso de estruturas quando se pretende representar tipos de dados pequenos, simples e similares no seu comportamento e características.
- Definição técnica de uma estrutura:

  [*attributes*] [*acess-modifiers*] **struct** identifier [:*interface-list*]

  {*struct members*};

C# - Estruturas

```
using System;
namespace MyStruct
{
     public struct Location
     {
          private int xVal;
          private int yVal;

          public Location (int xCoordinate, int yCoordinate)
          {
               xVal = xCoordinate;
               yVal = yCoordinate;
          }

          public int x
          {
               get
               {
                    return xVal;
               }
               set
               {
                    xVal = value;
               }
          }
```

```
        public int y
        {
            get
            {
                return yVal;
            }
            set
            {
                yVal = value;
            }
        }

        public override string ToString()
        {
            return (String.Format("{0}, {1}", xVal, yVal));
        }
}
```

C# - Estruturas

```
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    public class Tester
    {
            public void myFunc (Location loc)
            {
                    loc.x = 50;
                    loc.y = 100;
                    Console.WriteLine("In MyFunc loc: {0}", loc);
            }
            static void Main()
            {
                    Location loc1 = new Location(200,300);
                    Console.WriteLine("Loc1 location: {0}", loc1);
                    Tester t = new Tester();
                    t.myFunc(loc1);
                    Console.WriteLine("Loc1 location: {0}", loc1);
            }
    }
}
```

C# - Estruturas

- Sem destrutor ou construtor por defeito:
  - Estruturas não possuem destrutores, nem possuem construtor por defeito sem parametros.
  - Se não for fornecido construtor, será fornecido um por defeito que colocará todos os membros a zero. Se existir um construtor, é obrigatório inicializar os membros.
- Sem inicialização:
  - Não é permitido inicializar os membros dentro da estrutura. Sendo assim é ilegal proceder à seguinte codificação:

```
privare int xVal = 50;

privare int xVal = 50;
```

  - Isto tem que ser executado dentro da classe.

C# - Operador Overloading

- O operador overloading permite que classes definidas pelo utilizador possuam as mesmas funcionalidades que os tipos de dados pré-definidos.
- Como exemplo suponha que possui uma classe para representar fracções, Garantir que esta classe possui todas as funcionalidades que tipos de dados pré-definidos, significa que a classe definida por si tem capacidade de executar as mesmas operações que as classes de dados pré-definidos. (Exemplo: adicionar duas fracções, multiplicar duas fracções, e converter fracções em dados do tipo int).

  Fraction theSum = firstFraction + secondFraction;

- Como o operador + já existe, será necessário efectuar um *overloading* do mesmo, de forma a permitir efectuar uma soma de objectos da nossa classe.

C# - Operador Overloading

Operador Comparação ==

- É bastante comum efectuar o *overloading* do operador ==, no entanto a linguagem C# insiste que será necessário efectuar o mesmo para o operador !=.

  O mesmo se passa em relação aos operadores < e >, será necessário implementar o *overloading* dos operadores <= e =>.

- Sendo assim é preferível que se efectue antes um override ao método Virtual **Equals()**. Isto permite que a classe seja polimórfica e fornece compatibilidade com outras linguagens .NET que não suportam o **overload** de operadores (mas suportam o método *overloading*).

- A classe objecto implementa o método **Equals()** da seguinte forma:

  public virtual bool *Equals (****object*** *o)*

Operador Comparação ==

- Ao efectuar o override do método, garante-se que a classe Fraction actue de forma polimórfica com outros objectos.
- Dentro do método é necessário garantir que o objecto a comparar é um objecto do tipo Fraction.

```
public override bool Equals(object o)
{
    if (! (o is Fraction))
    {
            return false;
    }
    return this == (Fraction) o;
}
```

- O operador **is** é utilizado para garantir que o objecto é compatível com o operando. Caso o objecto seja Fraction a operação será avaliada.

C# - Operador Overloading

Operadores de Conversão

- A linguagem C# permite conversão implicita e explícita.
- Por exemplo converter um int num long consiste numa conversão implítica, pois o tamanho de um int cabe num long.
- A operação inversa traduz-se por uma conversão explícita.

```
int myInt = 5;
long myLong;
myLong = myInt          // implicita
myInt = (int) myLong    // explícita
```

C# - Operador Overloading

```
namespace MyOverloading
{
    using System;

    public class Fraction
    {
        private int numerator;
        private int denominator;

        public Fraction (int numerator, int denominator)
        {
                Console.WriteLine("In Fraction Constructor(int, int)");
                this.numerator = numerator;
                this.denominator = denominator;
        }

        public Fraction(int wholeNumber)
        {
                Console.WriteLine("In Fraction Constructor (int)");
                numerator = wholeNumber;
                denominator = 1;
        }
```

C# - Operador Overloading

```
public static implicit operator Fraction(int theInt)
{
        Console.WriteLine("In implicit conversion to Fraction");
        return new Fraction (theInt);
}

public static explicit operator int(Fraction theFraction)
{
        Console.WriteLine("In implicit conversion to int");
        return (theFraction.numerator / theFraction.denominator);
}

public static bool operator == (Fraction lhs, Fraction rhs)
{
        Console.WriteLine("In operator ==");
        if ((lhs.denominator == rhs.denominator)&& (lhs.numerator == rhs.numerator))
        {
                return true;
        }
        // colocar código quando as frações são diferentes
        return false;
}

public static bool operator != (Fraction lhs, Fraction rhs)
{
        return !(lhs==rhs);
}
```

C# - Operador Overloading

```
public override bool Equals(object o)
{
        Console.WriteLine("In method Equals");
        if (! (o is Fraction))
        {
                return false;
        }
        return this == (Fraction) o;
}
public static Fraction operator + (Fraction lhs, Fraction rhs)
{
        Console.WriteLine("In operator +");
        if (lhs.denominator == rhs.denominator)
        {
                return new Fraction (lhs.numerator+rhs.numerator, lhs.denominator);
        }
        // simlistic solution for unlike fractions
        // 1/2 + 3/4 == (1*4) + (3*2) / (2*4) = 10/8
        int firstProduct = lhs.numerator * rhs.denominator;
        int secondProduct = lhs.denominator * rhs.numerator;
        return new Fraction (firstProduct+secondProduct, lhs.denominator*rhs.denominator);
}
public override string ToString()
{
        string s = numerator.ToString() + "/"+denominator.ToString();
        return s;
}
}
```

C# - Operador Overloading

```
public class Tester
{

    static void Main()
    {
            Fraction f1 = new Fraction(3,4);
            Console.WriteLine("f1: {0}", f1.ToString());

            Fraction f2 = new Fraction(2,4);
            Console.WriteLine("f2: {0}", f2.ToString());

            Fraction f3 = f1+f2;
            Console.WriteLine("f1 + f2 = f3: {0}", f3.ToString());

            Fraction f4 = f3+5;
            Console.WriteLine("f3 + 5 = f4: {0}", f4.ToString());

            Fraction f5 = new Fraction(2,4);
            if (f5 == f2)
            {
                    Console.WriteLine("f5: {0} == f2: {1}", f5.ToString(), f2.ToString());
            }
    }
}
}
```

C# - Interfaces

- Uma interface consiste basicamente num contracto com o cliente sobre o comportamento de uma determinada classe ou estrutura.
- Quando uma classe implementa uma interface, isto significa, que essa classe garante que suporta todos os métodos, propriedades e eventos da referida interface.
- Sintáticamente, uma interface é semelhante a uma classe que possui somente métodos abstractos.
- A implementação de uma interface implica relações.
- Um carro **é um** veículo, no entanto pode implementar a capacidade *PodeSerCompradoComRecursoCredito*.

[*attributes*] [*access-modifier*] interface *interface-name* [:*base-list*]

{*interface-body*}

C# - Interfaces

```
namespace MyInterface
{
     using System;
     // declare the interface
     interface IStorable
     {
         // no access modifiers, methods are public
         // no implementation
         void Read(  );
         void Write(object obj);
         int Status { get; set; }

     }
     // Take our interface out for a spin
     public class Tester
     {

         static void Main(  )
         {
                 // access the methods in the Document object
                 Document doc = new Document("Test Document");
                 doc.Status = -1;
                 doc.Read(  );
                 Console.WriteLine("Document Status: {0}", doc.Status);
         }
     }
```

C# - Interfaces

```
// create a class which implements the IStorable interface
public class Document : IStorable
{
    public Document(string s){
            Console.WriteLine("Creating document with: {0}", s);
    }
    // implement the Read method
    public void Read(  ){
            Console.WriteLine(
                    "Implementing the Read Method for IStorable");
    }
    // implement the Write method
    public void Write(object o){
            Console.WriteLine(
                    "Implementing the Write Method for IStorable");
    }
    // implement the property
    public int Status{
            get {
                    return status;
            }
            set{
                    status = value;
            }
    }
    // store the value for the property
    private int status = 0;
}
```

Implementando mais do que uma interface

- É permitido que uma classe implemente mais do que uma interface, bastando para isso indicar o nome das interfaces, separadas por uma vígula.

```
public class Document : IStorable, ICompressible
```

Extensão de interfaces

- É possível a uma interface herdar as caraterísticas de outra interface.

```
interface ILoggedCompressible : ICompressible
```

- As classes são livres de implementar a interface que lhes convém mais, no entanto classes que implementem a interface ILoggedCompressible têm que implementar os métodos de ambas as interfaces.

Junção de Interfaces.

- É possível desenvolver novas interfaces a partir da junção de interfaces já existentes e opcionalmente adicionar novos métodos ou propriedades.

```
interface IStorableCompressible : IStorable, ILoggedCompressible
{
     void LogOriginalSize();
}
```

- Esta interface combina os métodos das outras duas interfaces e implementa um novo método que permite guardar o tamanho original do item antes da compressão.

C# - Interfaces

```
namespace MyInterfaceCombining
{
     using System;
     interface IStorable {
         void Read();
         void Write(object obj);
         int Status { get; set; }

     }
     // here's the new interface
     interface ICompressible {
         void Compress();
         void Decompress();
     }
     // Extend the interface
     interface ILoggedCompressible : ICompressible {
         void LogSavedBytes();
     }
     // Combine Interfaces
     interface IStorableCompressible : IStorable, ILoggedCompressible {
         void LogOriginalSize();
     }
     // yet another interface
     interface IEncryptable {
         void Encrypt();
         void Decrypt();
     }
```

C# - Interfaces

```
public class Document : IStorableCompressible, IEncryptable
{
    // the document constructor
    public Document(string s)
    {
            Console.WriteLine("Creating document with: {0}", s);

    }
    // implement IStorable
    public void Read()
    {
            Console.WriteLine("Implementing the Read Method for IStorable");
    }
    public void Write(object o)
    {
            Console.WriteLine("Implementing the Write Method for IStorable");
    }
    public int Status
    {
            get{
                    return status;
            }
            set{
                    status = value;
            }
    }
```

C# - Interfaces

```
    // implement ICompressible
    public void Compress() {
            Console.WriteLine("Implementing Compress");
    }
    public void Decompress() {
            Console.WriteLine("Implementing Decompress");
    }
    // implement ILoggedCompressible
    public void LogSavedBytes() {
            Console.WriteLine("Implementing LogSavedBytes");
    }
    // implement IStorableCompressible
    public void LogOriginalSize() {
            Console.WriteLine("Implementing LogOriginalSize");
    }
    // implement IEncryptable
    public void Encrypt() {
            Console.WriteLine("Implementing Encrypt");

    }
    public void Decrypt() {
            Console.WriteLine("Implementing Decrypt");

    }
    // hold the data for IStorable's Status property
    private int status = 0;
}
```

C# - Interfaces

```
public class Tester
{
    static void Main()
    {
            // create a document object
            Document doc = new Document("Test Document");

            // cast the document to the various interfaces
            IStorable isDoc = doc as IStorable;
            if (isDoc != null)
            {
                    isDoc.Read();
            }
            else
                    Console.WriteLine("IStorable not supported");

            ICompressible icDoc = doc as ICompressible;
            if (icDoc != null)
            {
                    icDoc.Compress();
            }
            else
                    Console.WriteLine("Compressible not supported");
```

C# - Interfaces

```
            IStorableCompressible isc = doc as IStorableCompressible;
            if (isc != null)
            {
                    isc.LogOriginalSize();  // IStorableCompressible
                    isc.LogSavedBytes();    // ILoggedCompressible
                    isc.Compress();         // ICompressible
                    isc.Read();             // IStorable

            }
            else
            {
                    Console.WriteLine("StorableCompressible not supported");
            }

            IEncryptable ie = doc as IEncryptable;
            if (ie != null)
            {
                    ie.Encrypt();
            }
            else
                    Console.WriteLine("Encryptable not supported");
        }
    }

}
```

Implementação explícita de uma Interface.

- A implementação explícita torna-se necessário quando uma classe implementa duas interfaces que contêm pelo menos um método com o mesmo nome.

```
interface IStorable{
      void Write();
      void Read()
}
interface ITalk{
      void Talk();
      void Read()
}
```

```
void ITalk.Read()
{
      Console.WriteLine("Implementig ITalk.Read");
}
```

C# - Vectores

- Um vector (***Array***) consiste numa coleção indexada de dados do mesmo tipo.
- Relativamente à liguagem C#, os vectores diferem das outras linguagens na medida em que todos os dados constituintes do vector são objectos.
- Declaração de um vector: *type* [] *array-name*;
- Os parênteses rectos indicam ao compilador que se trata de um vector.
- A instanciação de um vector é efectuada recorrendo à palavra **new**:

```
int[] myIntArray;

myIntArray = new int[5];
```

- A declaração anterior permitiu declarar um vector de 5 elementos inteiros, e alocar memória para os mesmos.

Valores por defeito

- Ao criar um vector de elementos do tipo valor, cada elemento do vector será inicializado a zero.
- Se o vector for do tipo referência cada elemento do vector será inicializado a **null**, ou seja, se tentar aceder a um elemento do vector sem o ter inicializado primeiro, provocará uma excepção.
- O seguinte exemplo:

```
Button[] myButtonArray = new Button [3];
```

Não cria um array com três objectos Button, mas sim um vector de 3 elementos do tipo Button. Para conter objectos deste género, será necessário primeiro inicializar cada elemento com um objecto deste tipo.

Aceder aos elementos

- A indexação de um vector é inicializada a zero, e cada elemento terá que ser acedido pela sua posição dentro de [].

- Como referido anteriormente, os vectores são objectos, logo possuem propriedades, sendo uma delas a **length** (tamanho). Como os vectores são indexados a zero, é necessário ter em consideração que a última posição do vector é **length-1**

C# - Vectores

```
namespace MyArrayFor
{
    using System;

    public class Employee
    {
        private int empID;

        public Employee(int empID)
        {
                this.empID = empID;
        }

        public override string ToString()
        {
                return empID.ToString();
        }
    }
```

C# - Vectores

```
public class Tester
{

    static void Main()
    {
            int[] intArray;
            Employee[] empArray;
            intArray = new int[5];
            empArray = new Employee[3];

            //populate the array
            for (int i = 0; i<empArray.Length;i++)
            {
                    empArray[i]= new Employee(i+5);
            }

            for (int i = 0; i<intArray.Length;i++)
            {
                    Console.WriteLine(intArray[i].ToString());
            }

            for (int i = 0; i<empArray.Length;i++)
            {
                    Console.WriteLine(empArray[i].ToString());
            }
    }
}
}
```

C# - Vectores - foreach

```
public class Tester
{

    static void Main()
    {
            int[] intArray;
            Employee[] empArray;
            intArray = new int[5];
            empArray = new Employee[3];

            //populate the array
            for (int i = 0; i<empArray.Length;i++)
            {
                    empArray[i]= new Employee(i+5);
            }

            foreach (int i in intArray)
            {
                    Console.WriteLine(intArray[i].ToString());
            }

            foreach (Employee e in empArray)
            {
                    Console.WriteLine(e.ToString());
            }
    }
}
}
```

Parâmetros - params

- Permite a passagem de diversos parâmetros para um método, sem ser necessário especificar no método quantos parâmetros são.

```
namespace MyArrayParams
{
     using System;
     public class Tester
     {
         static void Main ()
         {
                 Tester t = new Tester();
                 t.DisplayVals(5,6,7,8);
                 int [] explicitArray = new int[5] {1,2,3,4,5};
                 t.DisplayVals(explicitArray);
         }
         public void DisplayVals (params int[] intVals)
         {
                 foreach (int i in intVals)
                 {
                         Console.WriteLine("Displays {0}", i);
                 }
         }
     }
}
```

Regulares

- Um vector multidimensional regular é um vector que possui o mesmo número de colunas para todas as linhas.
- Num array multimensional de duas dimensões, a 1ª consiste no número de linhas enquanto a 2ª dimensão representa o número de colunas.
- Declaração de um vector de duas dimensões:

```
int [,] myRectangleArray = new int [2,3];
```

- A existência de uma vírgula indica que o vector é de duas dimensões (duas vírgulas indica 3 dimensões e por aí adiante).
- Neste género de vectores também é possível utilizar a inicialização através de {}:

```
int [,] myRectangleArray = { {0,1,2}, {3,4,5}, {6,7,8}, {9,10,11}};
```

C# - Vectores Multidimensionais

Jagged

- Um vector vector jagged consiste num vector de vectores, onde cada linha não tem obrigatoriamente de ter o mesmo número de colunas das anteriores.
- Ao criar um vector jagged, define-se unicamente o número de linhas do mesmo. Cada linha irá conter então um outro vector, o qual terá o tamanho desejado.
- Definição de um vector jagged:

```
int [][] myMyJagged = new int [rows][];
```

C# - Vectores - jagged

```
namespace MyJaggedArray
{
    using System;

    public class Tester
    {
        static void Main()
        {
                const int rows = 4;

                // declare the jagged array as 4 rows high
                int[][] jaggedArray = new int[rows][];

                // the first row has 5 elements
                jaggedArray[0] = new int[5];

                // a row with 2 elements
                jaggedArray[1] = new int[2];

                // a row with 3 elements
                jaggedArray[2] = new int[3];

                // the last row has 5 elements
                jaggedArray[3] = new int[5];
```

C# - Vectores - jagged

```
            // Fill some (but not all) elements of the rows
            jaggedArray[0][3] = 15;
            jaggedArray[1][1] = 12;
            jaggedArray[2][1] = 9;
            jaggedArray[2][2] = 99;
            jaggedArray[3][0] = 10;
            jaggedArray[3][1] = 11;
            jaggedArray[3][2] = 12;
            jaggedArray[3][3] = 13;
            jaggedArray[3][4] = 14;

            for (int i = 0;i < 5; i++){
                    Console.WriteLine("jaggedArray[0][{0}] = {1}", i,jaggedArray[0][i]);
            }

            for (int i = 0;i < 2; i++){
                    Console.WriteLine("jaggedArray[1][{0}] = {1}", i,jaggedArray[1][i]);
            }

            for (int i = 0;i < 3; i++){
                    Console.WriteLine("jaggedArray[2][{0}] = {1}",  i,jaggedArray[2][i]);
            }
            for (int i = 0;i < 5; i++){
                    Console.WriteLine("jaggedArray[3][{0}] = {1}",  i,jaggedArray[3][i]);
            }
        }
    }
}
```

C# - Vectores - ArrayList

- O problema com o tipo vector é o seu tamanho fixo. Se não for possível ter um conhecimento antecipado de quantos objectos será necessário armazenar, corre-se o risco de declarar ou um vector muito grande, ou um vector que não preenche as necessidades.

- Sendo assim a linguagem C# possui uma classe de dados denominada ArrayList, cujo tamanho pode ser dinâmicamente incrementado à medida das necessidades.

- Esta classe possui métodos e propriedades que ajudam bastante na sua utilização.

C# - Vectores - ArrayList

```
namespace MyArrayList
{
    using System;
    using System.Collections;
    // a simple class to store in the array
    public class Employee
    {
        public Employee(int empID)
        {
                this.empID = empID;
        }
        public override  string ToString()
        {
                return empID.ToString();
        }
        public int EmpID
        {
                get
                {
                        return empID;
                }
                set
                {
                        empID = value;
                }
        }
        private int empID;
    }
```

C# - Vectores - Dicionários

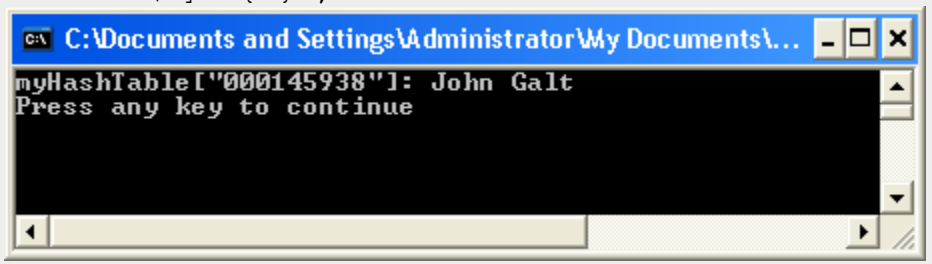
- Um dicionário consiste numa colecção de dados que associa uma **chave** a um **valor**.
- A Framework .NET permite que um dicionário possa associar qualquer tipo de chave (string, integer, object, etc) a qualquer tipo de valores (string, integer, object, etc).
- A característica mais importante de um dicionário é que seja de fácil utilização, isto é, que seja fácil de inserir novos valores e rápido a devolver os mesmos.

Hashtables

- É um tipo de dicionário optimizado para aceder a valores rápidamente.
- Numa *hash table* cada valor é armazenado num **bucket**, o qual é numerado de forma semelhante ao **offset** de um vector.
- A chave de uma *hash table* não necessita de ser inteira.

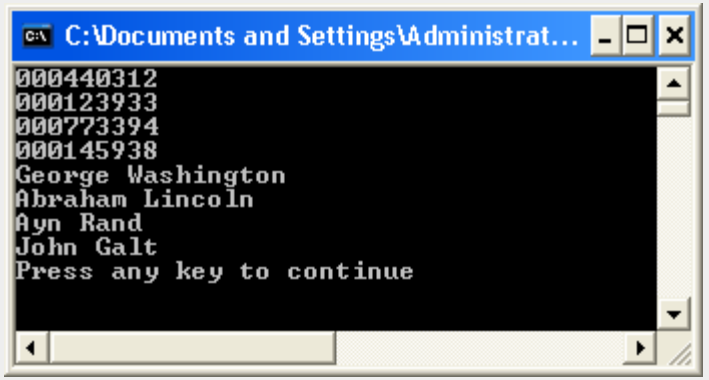
C# - Vectores - hashtables

```
namespace MyHash
{
     using System;
     using System.Collections;

     public class Tester
     {
         static void Main(  )
         {
                 // Create and initialize a new Hashtable.
                 Hashtable hashTable = new Hashtable(  );
                 hashTable.Add("000440312", "Jesse Liberty");
                 hashTable.Add("000123933", "Stacey Liberty");
                 hashTable.Add("000145938", "John Galt");
                 hashTable.Add("000773394", "Ayn Rand");

                 // access a particular item
                 Console.WriteLine("myHashTable[\"000145938\"]: {0}",
                         hashTable["000145938"]);
         }
     }
}
```



C# - Vectores - hashtables

```
namespace MyHashProperties
{
     using System;
     using System.Collections;
     public class Tester
     {
         static void Main(  )
         {
                 // Create and initialize a new Hashtable.
                 Hashtable hashTable = new Hashtable(  );
                 hashTable.Add("000440312", "George Washington");
                 hashTable.Add("000123933", "Abraham Lincoln");
                 hashTable.Add("000145938", "John Galt");
                 hashTable.Add("000773394", "Ayn Rand");

                 // get the keys from the hashTable
                 ICollection keys = hashTable.Keys;
                 // get the values
                 ICollection values = hashTable.Values;
                 // iterate over the keys ICollection
                 foreach(string key in keys){
                         Console.WriteLine("{0} ", key);
                 }
                 // iterate over the values collection
                 foreach (string val in values){
                         Console.WriteLine("{0} ", val);
                 }
         }
     }
}
```



C# - Vectores - IDictionaryEnumerator

```
namespace MyDictionaryEnumerator
{
    using System;
    using System.Collections;
    public class Tester
    {
        static void Main(  )
        {
                // Create and initialize a new Hashtable.
                Hashtable hashTable = new Hashtable(  );
                hashTable.Add("000440312", "George Washington");
                hashTable.Add("000123933", "Abraham Lincoln");
                hashTable.Add("000145938", "John Galt");
                hashTable.Add("000773394", "Ayn Rand");

                // Display the properties and values of the Hashtable.
                Console.WriteLine( "hashTable" );
                Console.WriteLine( "  Count:    {0}", hashTable.Count );
                Console.WriteLine( "  Keys and Values:" );
                PrintKeysAndValues( hashTable );
        }
        public static void PrintKeysAndValues( Hashtable table )
        {
                IDictionaryEnumerator enumerator = table.GetEnumerator(  );
                while ( enumerator.MoveNext(  ) )
                        Console.WriteLine( "\t{0}:\t{1}",enumerator.Key, enumerator.Value );
                Console.WriteLine(  );
        }
    }
}
```