BCSCHOOL

Empowering Digital Skills For The Jobs Of The Future

by

BC Soft
consulenza sistemi informativi

AiRED
Artificial Intelligence Ricerca e Sviluppo

Academy .NET

Interfacce e Classi Predefinite

Sommario

- IConvertible
- IEnumerator e IEnumerable
- ICloneable
- IComparable e IComparer
- IDictionary
- IList

Interfaccia IConvertible

Consente la **conversione dinamica dei tipi di dati** tramite la tecnica di programmazione basata sulle interfacce

Metodi di IConvertible:

```
public interface IConvertible
{
bool ToBoolean(IFormatProvider provider);
char ToChar(IFormatProvider provider);
……
UInt64 ToUInt64(IFormatProvider provider);
}
```

Interfaccia IConvertible

Per ottenere l'accesso a ogni membro dell'interfaccia **IConvertible** è necessario richiedere esplicitamente l'accesso all'interfaccia:

```
bool myBoll = true;
IConvertible i = (IConvertible) myBool;
```

Ora **i** è convertibile in qualsiasi tipo valido.

- Metodi IConvertible:
  - Questi metodi permettono la conversione da un tipo a un altro se compatibili
  - Se la conversione non è possibilie verrà generata un'eccezione **InvalidCastException**

IEnumerable

- **IEnumerable**: espone l'enumeratore, che supporta una semplice iterazione su una raccolta:

  GetEnumerator(): metodo pubblico che restituisce un enumeratore in grado di scorrere una raccolta

  Definito nel namespace **System.Collections**

IEnumerator

- **IEnumerator**: supporta una semplice iterazione su una raccolta:

  MoveNext(): metodo pubblico che fa avanzare l'enumeratore all'elemento successivo della raccolta

Esistono diverse classi che implementano le interfacce **IEnumerable e IEnumerator:**

- **Array**, **ArrayList**, **Stack**, **Queue**

IEnumerator - Esempio

```
public class Cars
    {
        // This class maintains an array of cars.
        private Car[ ] carArray;
        // Current position in array.
        // int pos = -1;
        public Cars()
        {
            carArray = new Car[4];
            carArray[0] = new Car("FeeFee", 200, 0);
            carArray[1] = new Car("Clunker", 90, 0);
            carArray[2] = new Car("Zippy", 30, 0);
            carArray[3] = new Car("Fred", 30, 0);
        }
    }
```

GetEnumerator()

Questo codice viene utilizzato Foreach e non GetEnumerator()

```
public class CarDriver
{
        public static void Main()
        {
            Cars carLot = new Cars();
            Console.WriteLine("Here are the cars in your lot");
            foreach (Cars c in carLot)
            {
                Console.WriteLine("-> Name: {0}", c.PetName);
                Console.WriteLine("-> Max speed: {0}", c.MaxSpeed);
                Console.WriteLine();
            }
        }
}
```

Implementazione GetEnumerator()

Ora si vedrà come utilizzare l'Enumerator piuttosto che il foreach:

```
public class Cars : IEnumerable
    {
        ……..
        // GetEnumerator() returns IEnumerator
        public IEnumerator GetEnumerator()
        {  // ??????? }
        ……..
    }
```

IEnumerable

IEnumerable.GetEnumerator() restituisce un oggetto che implementa **IEnumerator**:

```
public interface IEnumerator
{
bool MoveNext();
object Current { get; }
void Reset();
}
```

IEnumerable

Aggiornamento della classe Cars:

```
public class Cars : IEnumerable, IEnumerator
{
        ……..
        public IEnumerator GetEnumerator()
        {
            return   (IEnumerator) this;
        }
        ……..
}
```

IEnumerable

Implementazione **MoveNext()**

```
public bool MoveNext()
{
    if(pos < carArray.Length)
    {
        pos++;
        return true;
    }
    else
        return false;
}
```

```
public void Reset()
{
    pos = 0;
}
public object Current
{
get
{
    return carArray[pos];
}
}
```

IEnumerable

- I tipi possono essere iterati grazie il ciclo "**foreach**"

- Quando i membri **IEnumerator** vengono implementati in modo esplicito, è possibile fornire un metodo alternativo per accedere agli oggetti nel container

Interfaccia IClonable

Per evitare il problema **shallow copy** è necessario implementare l'interfaccia **Icloneable:**

```
public interface ICloneable
{
    object Clone();
}
```

Metodo Clone()

Il metodo **Clone()** deve essere ridefinito per adattarsi al tipo di oggetto.

```
public class Point : ICloneable
{
    …..
    public object Clone()
    { return new Point(this.x, this.y); }
    ….
}
```

Chiamata metodo Clone()

Evitare la shallow copy:

```
Point p1 = new Point(50, 50);
// p2 will point to the copy of p1
Point p2 = (Point)p1.Clone();
p2.x = 0; // will not affect p1
Console.WriteLine(p1);
Console.WriteLine(p2);
```

IComparable

Implementazione interfaccia **IComparable:**

Definisce un metodo di confronto generalizzato che un **tipo valore** o una **classe** implementa per creare un metodo di confronto specifico del tipo

CompareTo(object o): confronta l'istanza corrente con un altro oggetto dello stesso tipo

```
public interface IComparable
{
    int CompareTo(object o);
}
```

System.Array

- La **classe System.Array** definisce un metodo **Sort()** che ordina int, float, char, ecc.

- Tuttavia, se l'array contiene un set di Cars, fallisce.

```
Array.Sort (intArr); // OK
Array.Sort (myCars); // ArgumentException thrown
```

- Il vantaggio di **IComparable** è fornire un modo per ordinare i propri tipi

IComparable - Esempio

```
public class Student : IComparable
    {
        private int USN;
        public Student() {USN = 0; }
        public Student(int USN) { this.USN = USN; }
        public int RegNo
        {
            get
            { return USN; }
        }

        int IComparable.CompareTo(object o)
        {
            Student temp = (Student) o;
            if(this.USN > temp.USN) return 1;
            if(this.USN < temp.USN) return -1;
            else return 0;
        }
}
```

Valore di ritorno **CompareTo()**:
<0: minore dell' oggetto
0: uguale all'oggetto
>0: maggiore dell'oggetto

Ordinare oggetti

```
static void Main(string[] args)
    {
        Student[] cseStd = new Student[3];
        cseStd[0] = new Student(111);
        cseStd[1] = new Student(100);
        cseStd[2] = new Student(45);
        try
        {
            Array.Sort(cseStd);
        }
        catch(Exception e)
        {
            Console.WriteLine(e.StackTrace);
        }
        Console.WriteLine("Array after sorting");
        foreach(Student s in cseStd)
            Console.WriteLine(s.RegNo);
    }
```

System.Collection Namespace

ArrayList	**Array dinamico di oggetti**	**IList, ICollection, IEnumerable, ICloneable**
Hashtable	**Collezione identificata da chiavi**	**IDictionary, ICollection, IEnumerable, ICloneable**
Queue	**FIFO**	**ICollection, IEnumerable, ICloneable**
SortedList	**Simile a Dictionary, con accesso da indice**	**IDictionary, ICollection, IEnumerable, ICloneable**
Stack	**LIFO**	**ICollection e IEnumerable**

ArrayList

- Non è garantito che **ArrayList** venga ordinato. È necessario ordinare ArrayList prima di eseguire operazioni che richiedono l'ordinamento di ArrayList.

- La capacità di un **ArrayList** è data dal numero di elementi che può contenere. La capacità iniziale predefinita per un **ArrayList** è 0.

- Man mano che gli elementi vengono aggiunti a un **ArrayList**, la capacità viene automaticamente aumentata come richiesto tramite la riallocazione.

ArrayList

- La capacità può essere ridotta chiamando **TrimToSize** o impostando esplicitamente la proprietà **Capacity**.

- È possibile accedere agli elementi di questa raccolta utilizzando un indice intero. Gli indici in questa raccolta sono a base zero.

- ArrayList accetta un riferimento **null** come valore valido e consente elementi duplicati.

ArrayList - Esempio

```
using System;
using System.Collections;
public class SamplesArrayList
{
    public static void Main()
    {   // Creates and initializes a new ArrayList.
    ArrayList myAL = new ArrayList();
    myAL.Add("Hello");
    myAL.Add("World");
    myAL.Add("!");
    Console.WriteLine( " Count: {0}", myAL.Count );
    Console.WriteLine( " Capacity: {0}", myAL.Capacity );
    Console.Write( " Values:" );
    PrintValues( myAL );
}
public static void PrintValues( IEnumerable myList )
{
    foreach ( Object obj in myList )
        Console.Write( " {0}", obj ); Console.WriteLine();
}
```

```
myAL Count: 3
Capacity: 16
Values: Hello World !
```

System.Collections.Queue

- **Dequeue():** elimina e restituisce il primo oggetto dalla coda
- **Enqueue():** inserisce un oggetto nella coda
- **Peek()** - Restituisce il primo oggetto senza eliminarlo

```
Console.WriteLine("\n***** Queue Demo *****");
// Now make a Q with three items
Queue carWashQ = new Queue();
carWashQ.Enqueue(new Car("FirstCar", 0, 0));
carWashQ.Enqueue(new Car("SecondCar", 0, 0));
carWashQ.Enqueue(new Car("ThirdCar", 0, 0));

// Peek at first car in Q
Console.WriteLine("First in Q is {0}",
            ((Car)carWashQ.Peek()).PetName);
```

System.Collections.Queue

```
// Remove each item from Q
WashCar((Car)carWashQ.Dequeue());
WashCar((Car)carWashQ.Dequeue());
WashCar((Car)carWashQ.Dequeue());

// Try to de-Q again?
try
{
    WashCar((Car)carWashQ.Dequeue());
}
catch(Exception e)
{ Console.WriteLine("Error!! {0}", e.Message);}
```

System.Collections.Stack

```
Stack stringStack = new Stack();
stringStack.Push("One");
stringStack.Push("Two");
stringStack.Push("Three");
// Now look at the top item.
Console.WriteLine("Top item is: {0}", stringStack.Peek());
Console.WriteLine("Popped off {0}", stringStack.Pop());
Console.WriteLine("Top item is: {0}", stringStack.Peek());
Console.WriteLine("Popped off {0}", stringStack.Pop());
Console.WriteLine("Top item is: {0}", stringStack.Peek());
Console.WriteLine("Popped off {0}", stringStack.Pop());
try
{
    Console.WriteLine("Top item is: {0}", stringStack.Peek());
    Console.WriteLine("Popped off {0}", stringStack.Pop());
}
catch(Exception e)
{ Console.WriteLine("Error!! {0}\n", e.Message);}
```

Domande & approfondimenti

Academy .NET