Drukuj Zamknij

Tworzenie kolekcji dedykowanej w C#

2003-11-30 01:48:58
Mariusz Matrejek - Wydział Informatyki i Zarządzania, Politechnika Wrocławska

Artykuł wymaga wiedzy z zakresu następujących zagadnień: czym są interfejsy

Kolekcja w .NET to nic innego jak zbiór obiektów w formie tablicy z oprogramowanymi funkcjami pozwalającymi na efektywną interakcję (chyba najmniej ważnym przykładem, lecz najbardziej obrazowym jest użycie instrukcji **foreach**). Wielokrotnie w trakcie pisania programów spotkałem się z sytuacjami, gdzie kolekcja rozwiązałaby problem, który najczęściej sprowadzał się do oprogramowania kilku metod nadającym tablicom namiastkę funkcjonalności kolekcji. Dostępna kolekcja **ArrayList**, nie do końca odpowiadała moim potrzebom. Przede wszystkim operuje ona na elementach typu **Object**, co w konsekwencji prowadzi do marnotrawstwa związanego z ciągłym rzutowaniem elementów kolekcji w celu wywołania ich metod. W końcu zdecydowałem się poświęcić, jak się okazało wcale nie tak dużo czasu i stworzyć własną kolekcję, nadając jej nazwę „**kolekcji dedykowanej**" (w tłumaczeniu angielskim określilibyśmy ją jako „**strongly-typed collection**"). Jej uniwersalność, mimo konieczności oprogramowania całkiem sporej liczby metod (z części zresztą zrezygnowałem, uznając je za zbędne) zaskoczyła mnie. Raz napisany kod kolekcji możemy przystosowywać do obsługi innych obiektów zmieniając niewielkie fragmenty kodu. Uzyskujemy też lepszą kontrolę typów oraz większą efektywność, związaną z brakiem rzutowania. W niniejszym artykule chciałbym podzielić się z Wami moimi doświadczeniami.

W tym artykule stworzymy klasę będącą kolekcją (a dokładniej lista) punktów (x,y). Kolekcja będzie posiadać funkcjonalność podobną do kolekcji **ArrayList** z przestrzeni nazw System.Collections.

Spróbujmy zatem prześledzić proces jej tworzenia.

Przygotowania

Zobacz też: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollections.asp [System.Collections]

Na początku tworzymy klasę, którą nazwiemy **MarPoint**. Nie będę wdawał się w szczegóły jej implementacji. Ważne jest, aby metoda **Equals** posiadała formę podaną w listingu, ponieważ jest ona używana później przy definiowaniu metody **Contains** kolekcji. Kod klasy znajduje się w solucji załączonej do artykułu.

Następnie przystępujemy do tworzenia własnej kolekcji. Kolejno wykonamy następujące kroki:

1. Utworzymy własny indeksator (stworzymy obiekt będący tablicą obiektów **MarPoint**)
2. Zamienimy naszą klasę na kolekcję (zaimplementujemy interfejsy **IEnumerator** i **IEnumarable**)
3. Przygotujemy naszą kolekcję do inicjacji z pojemnością ograniczoną lub nieograniczoną.
4. „Zaimplementujemy" interfejs **IList**.
5. Dodamy implementacje interfejsu **ICollection**.

Zanim przystąpimy do realizacji tych punktów musimy stworzyć szkielet klasy. Decelowo powinien on mieć postać:

```
public class MarPointList : IEnumerator, IEnumerable, ICollection
{
```

```
}
```

Na potrzeby testowania kolejnych etapów sugeruję dodawanie dziedziczenia kolejnych interfejsów dopiero w momencie ich faktycznej implementacji.

Tworzenie własnego indeksatora

Zobacz też: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vclrfindexedpropertiespg.asp [indeksatory]

Dostęp do grupy obiektów tego samego typu na zasadzie podawania indeksu jest bardzo wygodny. Na początek wprowadzimy w naszej klasie kilka pól. Właściwą zawartość naszej kolekcji deklarujemy jako tablicę punktów:

```
private MarPoint[] _elements;
```

Wprowadzamy też informacje o aktualnej pojemności maksymalnej kolekcji **(_capacity)**, która może się zmieniać w zależności od ustawień pola **_isFixedSize**. ograniczonej wielkości kolekcji w celu późniejszej kontroli zasięgu, bieżącej ilości elementów kolekcji (**_size**) oraz, czy kolekcja jest *„tylko-do-odczytu”.* Pole **_index** potrzebne jest do zapamiętania, na którym elemencie kolekcji operujemy w danym momencie.

```
private int _capacity;
private int _index;
private int _size;
private bool _isFixedSize = false;
private bool _isReadOnly = false;
```

Cześć pól udostępniamy jako własność kolekcji za pomocą akcesorów.

```
public int Capacity
{
     get
     {
          return _capacity;
     }
}

public bool IsReadOnly
{
     get
     {
          return _isReadOnly;
     }
     set
     {
          _isReadOnly = value;
```

```
     }
}

public bool IsFixedSize
{
     get
     {
          return _isFixedSize;
     }
     set
     {
          _isFixedSize = value;
     }
}
```

Następnie oprogramować musimy dostęp do elementu kolekcji za pomocą indeksu (czyli tak jak dla zwykłej tablicy). Dokonujemy tego wprowadzając pole **this** w następujący sposób:

```
public MarPoint this[int pos]
{
     get
     {
          if (pos < 0 || pos >= _capacity)
               throw new IndexOutOfRangeException("Out of range!");
          else return _elements[pos];
     }
     set
     {
     if (_isReadOnly)
               throw new NotSupportedException("Operation not supported!");
          else
          {
          if (pos < 0 || pos >= _capacity)
                    throw new IndexOutOfRangeException("Out of range!");
               else
               {
               _elements[pos] = value;
                    _size++;
               }
          }
}
}
```

Jak widzimy nasze akcesory posiadają zabezpieczenia przed nieprawidłowymi odwołaniami do kolekcji (np. próba zapisu do kolekcji *„tylko-do-odczytu”*).

Aby wszystko działało poprawnie musimy zdefiniować jeszcze odpowiednie konstruktory. Zakładamy, że konstruktor będzie miał dwie wersje. Bezargumentową, która stworzy kolekcję o nieograniczonym rozmiarze (póki co zarezerwujemy „wystarczająco” dużą tablicę, później zoptymalizujemy kolekcję w celu lepszego wykorzystania pamięci). Druga wersja tworzy kolekcję o wyznaczonym rozmiarze:

```
public MarPointList()
{
     _elements = new MarPoint[1000];
     _capacity = 1000;
     _size = 0;
     _isFixedSize = true;
     _isReadOnly = false;
}

public MarPointList(int capacity)
{
     _elements = new MarPoint[capacity];
     _capacity = capacity;
     _isFixedSize = true;
     _isReadOnly = false;
     _size = 0;
}
```

W tym momencie nasz obiekt kolekcji jest czymś w rodzaju zubożonej tablicy. Idźmy więc dalej powiększając jego funkcjonalność.

Implementacja interfejsów IEnumerable i IEnumerator

Zobacz też: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollectionsienumeratorclasstopic.asp [IEnumerator]

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollectionsienumerableclasstopic.asp [IEnumerable]

Oba interfejsy zdefiniowane są w przestrzeni nazw **System.Collections**.

Interfejs **IEnumerable** służy do pozyskania interfejsu **IEnumerator** dla wybranej kolekcji, co umożliwia później łatwe iterowanie po niej za pomocą instrukcji **foreach**. Interfejs ten wymaga zaimplementowania tylko jednej metody **GetEnumerator()**:

```
public IEnumerator GetEnumerator()
{
     return (IEnumerator)this;
}
```

Interfejs **IEnumerator** definiuje własność **Current**, wskazującą na bieżący obiekt w kolekcji oraz metody **MoveNext()** - przesunięcie w kolekcji do przodu o jeden element i **Reset()** – przesunięcie na początek kolekcji. Przy czym metoda **MoveNext()** zwraca wartość logiczną w zależności od tego, czy w kolekcji można się jeszcze przemieścić dalej, czy też nie.

```
public bool MoveNext()
{
     if (_index < _capacity-1 && IsEmpty(_index+1) == false)
     {
          _index++;
          return true;
     }
     else return false;
}

public void Reset()
{
     _index = -1;
}
```

Ponieważ elementy kolekcji w założeniu występują zaraz po sobie koniec kolekcji wyznacza jej pierwszy pusty element. Aby go zidentyfikować implementujemy metodę pomocniczą **IsEmpty():**

```
private bool IsEmpty(int index)
{
     if (_elements[index] == null)
     {
          return true;
     }
     else return false;
}
```

Jeszcze tylko własność...

```
public object Current
{
     get
     {
          if (_index < 0) return null;
          else return (object)_elements[_index];
     }
}
```

... i nasza kolekcja jest już do wykorzystania z instrukcją **foreach**. Instrukcja ta rozpoczyna iterację od indeksu o wartości 0. Aby przechodziła całą kolekcję należy na końcu kodów konstruktorów dodać instrukcję **Reset();** Iteracja kończy się na ostatnim elemencie kolekcji, który pozostaje elementem aktywnym (kolekcja nie jest resetowana).

W ten sposób uzyskaliśmy już możliwość swobodnej iteracji po kolekcji. Przydałoby się jednak dołożyć kilka bardziej użytecznych metod.

Implementacja interfejsu ICollection

Zobacz też: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollectionsicollectionclasstopic.asp [ICollection]

Zanim jednak do tego dojdziemy zaimplementujemy interfejs **ICollection**. Interfejs (nota bene dziedziczony przez wspominany później **IList**) jest dziedziczony przez wszystkie kolekcje z przestrzeni nazw **System.Collections**. Definiuje on trzy własnośći: **Count** (określa liczbę elementów w kolekcji), **IsSynchronized** oraz **SyncRoot**. Dwie ostatnie własności wykorzystywane są w aplikacjach wielowątkowych. Iteracja **foreach** wykonywana w jednym wątku rzuci wyjątek z chwilą, gdy do danej kolekcji spróbuje odwołać się inny wątek. Po szczegóły odsyłam jednak do MSDN. Implementacja wspomnianych własności wygląda następująco:

```
public int Count
{
     get
     {
          return _size;
     }
}

public bool IsSynchronized
{
     get
     {
          return false;
     }
}

public object SyncRoot
{
     get
     {
          return this;
     }
}
```

Interfejs deklaruje również metodę **CopyTo()**, która pozwala na skopiowanie kolekcji do podanej tablicy. Tablicę wypełnia się poczynając od podanego indeksu:

```
public void CopyTo(Array array, int index)
{
     //czy tablica istnieje?
     if (object.ReferenceEquals(array, null))
          throw new ArgumentNullException("Array is null!");

     //czy indeks jest poprawny?
     if (index < 0)
          throw new IndexOutOfRangeException("Index is out of range!");

     //tablica musi być 1-wymiarowa
     if (array.Rank > 1)
           throw new ArgumentException("Array is not 1D!");

//i wystarczająco duża
     if (array.Length < _size)
          throw new NotSupportedException("Destination array is to small!");

     //kopiuj!
     foreach (object o in this)
     {
          array.SetValue(o, index);
          index++;
     }
}
```

Tablica, którą wypełniamy musi być tablicą jednowymiarową. Po zaimplementowaniu interfejsu **ICollection** możemy przejść dalej.

„Implementacja" interfejsu IList

Zobacz też: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollectionsilistclasstopic.asp [IList]

Gdy pierwszy raz implementowałem własną kolekcję wprowadziłem wiele potrzebnych mi metod. Następnie zauważyłem, że dużą część z nich deklaruje interfejs **IList** znajdujący się w przestrzeni nazw **System.Collections**. Implementacja tego interfejsu ograniczyłaby jednak naszą kolekcję (np. wymagane jest aby indeksator zwracał wartości typu **Object**, a nie typu **MarPoint**, co wymuszałoby późniejsze rzutowanie). Dlatego też zaimplementujemy dopowiedniki metod z **IList**, dostosowane do naszych potrzeb. Będą to: **Add** (dodanie elementu na koniec kolekcji), **Clear** (wyczyszczenie kolekcji), **Contains** (sprawdzenie, czy kolekcja zawieta dany element), **IndexOf** (pozycja danego elementu w kolekcji), **Insert** (wstawienie elementu na wskazanej pozycji), **Remove** (usunięcie elementu z końca kolekcji) i **RemoveAt** (usunięcie elementu na podanej pozycji). Ich kod jest następujący:

```
public int Add(MarPoint element)
{
     if (_isReadOnly)
```

```
            throw new NotSupportedException("Operation not supported!");
      else if (_size == _capacity && _isFixedSize)
            throw new NotSupportedException("Collection is full!");
      else
      {
            //znajdź koniec I wstaw
            _index = _size-1;
            if ((_index < _capacity - 1 && _isFixedSize)
                  ||(_isFixedSize == false))
            {
                  //może trzeba powiększyć kolekcję
                  if (_isFixedSize == false && _size == _capacity)
                  {
                        ExtendCollection(_capacity*2);                              }

                  _index++;
                  _elements[_index] = element;
                  _size++;
                  _index = -1;

                  return _index;                                                }
            else return -1;
      }
      return -1;
}

public void Clear()
{
      if (_isReadOnly)
            throw new NotSupportedException("Operation not supported!");
      else
      {
            _elements = new MarPoint[_capacity];
            _size = 0;
            Reset();
      }
}

public bool Contains(MarPoint element)
{
      for (int i = 0; i<_size; i++)
      {
            if (this[i] == element) return true;
```

```
        }
        return false;
    }

    public int IndexOf(MarPoint element)
    {
        for (int i = 0; i < _size; i++)
        {
            if (this[i].Equals(element)) return i;
        }
        return -1;
    }

    public void Insert(MarPoint element, int index)
    {
        //indeks ok?
        if (index > _capacity - 1)
        throw new IndexOutOfRangeException("Out of range!");
        else if (_isReadOnly)
        {
            throw new NotSupportedException("Operation not supported!");
        }
        else if (_size == _capacity && _isFixedSize)
        {
            throw new NotSupportedException("Collection is full!");
        }
        else
        {
            if (index == _size) this.Add(element);
            else //zrób miejsce
            {
                //powiększyć kolekcję?
                if (_isFixedSize == false && _size == _capacity)
                {
                    ExtendCollection(_capacity*2);                              }

                MarPoint[] temp = new MarPoint[_capacity];
                //kopiuj głowę
                for (int i = 0; i < index; i++)
                {
                    temp[i] = _elements[i];
                }
```

```
                //wklej element
                temp[index] = element;

                //kopiuj ogon
                for (int i = index; i < _size; i++)
                {
                     temp[i+1] = _elements[i];
                }

                //podmień tablicę
                _elements = temp;
                //wskaźnik zajętości
                _size++;
           }
     }
}

public void Remove()
{
     RemoveAt(_size-1);
}

public void RemoveAt(int index)
{
     if (_isReadOnly)
          throw new NotSupportedException("Operation not supported!");
     else if (index > _capacity || index > _size - 1)
     {
          throw new IndexOutOfRangeException("Out of range!");
     }
     else
     {
          //usuń
          _elements[index] = null;
          //przytnij
          if (index != _size - 1)
          {
               MarPoint[] temp = new MarPoint[_capacity];
               //kopiuj głowę
               for (int i = 0; i < index; i++)
               {
                    temp[i] = _elements[i];
               }
```

```
                //kopiuj ogon
                for (int i = index+1; i < _size; i++)
            {
                    temp[i-1] = _elements[i];
                }

                //podmień
                _elements = temp;
            }
            //wskaźnik zajętości
            _size--;
        }
}
```

Musimy dodać jeszcze metodę pomocniczą:

```
public void ExtendCollection(int newCapacity)
{
     if (newCapacity > _capacity)
     {
          _capacity = newCapacity;
          MarPoint[] temp = new MarPoint[_capacity];

          //kopiuj tablicę
          for (int i=0; i < _size; i++)
          {
               temp[i] = _elements[i];
          }

     //podmień
          _elements = temp;
     }
}
```

Jak możemy zauważyć operacje wstawiające elementy do kolekcji, badają, czy kolekcja jest już pełna. Jeżeli jest, a własność **IsFixedSize** ustawiona jest na **false** to kolekcja powiększa się dwukrotnie. Pozwala to na oszczędność pamięci. Jest to jednak operacja czasochłonna. Skoro jednak już ją zaimplementowaliśmy możemy zmniejszyć liczbę alokowanego miejsca na punkty w konstruktorze bezargumentowym. Powiedzmy, że z 1000 na 10.

Interfejs **IList** zawiera również trzy własności. Wszystkie zostały już zaimplementowane podczas tworzenia indeksatora. Działają w trochę odmienny sposób niż mówi to dokumentacja interfejsu, aby zapewnić naszej kolekcji większą funkcjonalność.

Interfejs **IList** mamy więc „zaimplementowany”.

Zakończenie

Nasza kolekcja jest już funkcjonalna. Oczywiście przydałoby się kilka innych operacji jak na przykład:

```
public MarPoint[] ToArray()
{
     return _elements;
}
```

Podana kolekcja nie jest wolna od wad. Po pierwsze nie została zaimplementowana możliwość współpracy wielu enumeratorów. Kolekcja mogłaby zwiększać swój rozmiar nie tylko za pomocą metody **Add()**, ale także za pomocą odwołania do indeksu spoza bieżącego rozmiaru (o ile kolekcja nie ma ustawionej wartości **IsFixedSize** na **true**). Implementacja interfejsu IEnumerator nie jest do końca zgodna z jego specyfikacją (porównajcie implementację **Current** ze specyfikacją). Niemniej wystarcza to już do efektywnej pracy, a o ile przyszłość na to pozwoli w innym artykule zaprezentuję dalsze możliwości rozwoju tej kolekcji (liczę także na uwagi czytelników).

Tymczasem jednak czas powiedzieć, jak możemy tę przykładową klasę zastosować do własnych potrzeb. Wystarczy, że stworzymy własną klasę – odpowiednik **MarPoint**. Po czym wymienimy wszystkie wystąpienia **MarPoint** na własną klasę. Może się zdarzyć, że niektóre operacje przebiegać mają w inny sposób, jednak w większości przypadków pozostały kod zostawimy bez zmian. Bardziej zaawansowanym programistom polecam zapoznanie się z C# generics, które pojawią się w Visual Studio .NET 2004 (kodowa nazwa „Whidbey”), a właściwie to w nowym Visual C# 2.0. C# generics pozwolą na implementację, znanych z C++ klas szablonowych, co pozwoli na łatwiejsze wykorzystanie przedstawionej tutaj kolekcji.

Zobacz też: http://msdn.microsoft.com/vstudio/productinfo/roadmap.aspx

A wracając do tematu. Musimy też pamiętać, że dodatkowy narzut kodu powoduje iż duże kolekcje działają wolniej niż bezpośrednie operacje na tablicach. Chociaż jeszcze nie miałem przypadku, w którym kolekcja krytycznie wpływała na czas realizacji algorytmu, to musimy zdawać sobie sprawę, że wybór, czy stosować kolekcję, czy tablicę zależy jedynie od naszych potrzeb w kontekście konkretnego problemu.

Życzę owocnego kolekcjonowania (.

MARIUSZ MATREJEK

P.S. W badaniu sposobu funkcjonowania kolekcji pomoże niewielka aplikacja, której kod dołączony jest do artykułu.

Drukuj Zamknij