حضر نفسك من هنا للانتقال الى سي شارب 2

هاني الأتاسي
2005/10/04

هذه الميزة قد تتيح تسهيلات كثيرة للمبرمجين عند تعاملهم مع ال Delegates . في السي# استخدام الdelegate قد يكون غريبا قليلا عن استخدام التوابع في سي++ . ففي السي++ إذا أردت تمرير مؤشر التابع إلى تابع ما تقوم فقط بكتابة اسم التابع . أما في السي# يجب عليك أن تنشئ كائن جديد delegate يحتوي التابع الذي تريد تمريره .

في الاصدار الجديد من سي# تمت تحسين هذه الطريقة في تمرير التوابع ، والآن بدلا من أن تنشئ delegate خاص يمكنك فقط كتابة اسم التابع تماما كما في سي++ :

كود

```
// previous version of C#
button1.Click += new EventHandler(button1_Click);

// new version of C#
button1.Click += button1_Click
```

والمثال التالي يوضح كيف يمكن تمرير اسم التابع فقط لتابع آخر بدلا من انشاء delegate خاص:

كود

```
class Program {
    static void Main() {
 ThreadPool.QueueUserWorkItem(MyCallbackFunction, 10);
    }

    static void MyCallbackFunction(object o) {
 Console.WriteLine(o);
    }
}
```

بالاضافة إلى كل السابق فتم اضافة تسهيلات جديدة في سي# الاصدار الثاني وهي تشابه ميزات موجودة في JavaScript ولكن أفضل بكثير. حيث يمكنك أن تكتب التابع Callback Function مباشرة أثناء تمريره بدلا من تعريف تابع منفصل .

كود

```
  class Program
   {
       static void Main(string[] args)
       {
         System.Threading.Timer timer = new System.Threading.Timer(
               delegate(Object obj) { MessageBox.Show("Hello :)"); }, 10, 1000, 1000);

           Console.ReadLine();
       }
   }
```

لاحظ كيف قمنا بكتابة الكود الذي يظهر الرسالة oHell مباشرة أثناء انشائنا للTimer . الكود السابق سوف يظهر الرسالة Hello كل ثانية حتى يتم انهاء البرنامج.

الآن إذا حاولنا رؤية الassembly الناتجة لمعرفة ماذا فعل المترجم للكود الذي كتبناه لنجد أن المترجم قد أنشئ تابع جديد وقام باضافة كود جديد داخل التابع nMai . استخدت أداة 'Lutz Roeder NET Reflector.s من http://www.aisto.com/roeder/dotnet/

كود

```
class Program {

 private static void Main(string[] args)
 {
     if (Program.<>9__CachedAnonymousMethodDelegate1 == null)
     {
           Program.<>9__CachedAnonymousMethodDelegate1 = new
TimerCallback(Program.<Main>b__0);
     }
     Timer timer1 = new Timer(Program.<>9__CachedAnonymousMethodDelegate1, 10, 0x3e8, 0x3e8);
     Console.ReadLine();
```

```
}

private static void <Main>b__0(object obj)
{
    MessageBox.Show("Hello :)");
}

private static TimerCallback <>9__CachedAnonymousMethodDelegate1;

}
```

لاحظ أنه تم تعريف متحول من نوع TimerCallback وهو static واسمه <>9__CachedAnonymousMethodDelegate1 . إذا لاحظنا داخل Main فإن هذا المتحول سوف يحتوي على الكائن الذي يحوي مؤشر إلى التابع الجديد الذي انتجه المترجم . هذا التابع اسمه <Main>b__0 وهو يحتوي الكود الذي كتبناه أصلا .

بالمختصر فإن العملية ليست معقدة كثيرا حتى الآن ، فالمترجم يقوم وراء الكوليس بعمل كل هذه الأمور بشكل آلي . لاحظ أنه لا توجد اي مشكلة إذا حاولنا من داخل الـanonymous method الوصول إلى متحولات معرفة داخل الكلاس لأن التابع الجديد الذي أنشئه المترجم هو أساسا داخل الكلاس. انظر المثال:

كود

```
class Program
 {
     static int counter = 0;

     static void Main(string[] args)
     {
         Timer timer = new Timer(
             delegate(Object obj) { Console.WriteLine(counter++); }, 10, 1000, 1000);

         Console.ReadLine();
     }
 }
```

تنفيذ البرنامج السباق سوف يظهر الأرقام 0 ، 1 ، 2 ، 3 ، 4 ، ....

أحب أن أذكر أيضا أنه ليس من الداعي ذكر البارامترات عند كتابتك لل anonymous method إذا لم تكن تستخدمهم من الداخل . على سبيل المثال يمكن كتابة الكود السابق كالتالي:

كود

```
class Program
 {
     static int counter = 0;

     static void Main(string[] args)
     {
         Timer timer = new Timer(
             delegate { Console.WriteLine(counter++); }, 10, 1000, 1000);

         Console.ReadLine();
     }
 }
```

الآن ، ماذا لو فكرنا قليلا وحاولنا الوصول إلى متحولات معرفة ضمن التابع الأساسي من داخل الـanonymous method .. كما رئينا سابقا فإن التابع الذي ينشئه المترجم هو تابع منفصل عن التابع الأساسي ، فكيف يمكن الوصول إلى المتحولات المعرفة داخل التابع الأساسي!؟ .. انظر المثال التالي:

كود

```
class Program
 {
     static void StartTimer()
     {
         int counter = 0;
         Timer timer1 = new Timer(
             delegate { Console.WriteLine("Timer1 = {0}", counter++); }, 10, 1000, 1000);
     }

     static void Main(string[] args)
     {
         StartTimer();
         Console.ReadLine();
     }
 }
```

نتيجة تنفيذ البرنامج السابق هي:

0

1

2

وهكذا ...

لاحظ أنه قمنا باستخدام المتحول الداخلي counter من ضمن ال anonymous method . وبعد انشائنا للمؤقت فإننا نخرج من التابع مباشرة ، ونرجع إلى التابع Main . ولكن ال counter هي عبارة عن متحول تم انشائه في ال stack ويتم حذفه مباشرة عند الخروج من التابع StartTimer . ولكن كيف عمل البرنامج بصورة صحيحة؟! وكأن المتحول counter مازال على قيد الحياة .

لو نظرنا إلى الكود المنتج من المثال السابق داخل الأداة .NET Reflecotr ، لوجدنا أن المترجم قال بعمل اضافي هنا، انظر إلى الكود الجديد الذي أنتجه المترجم:

كود

```
class Program
{
 private static void Main(string[] args) {
     Program.StartTimer();
     Console.ReadLine();
 }

 private static void StartTimer() {
     Program.<>c__DisplayClass1 class1 = new Program.<>c__DisplayClass1();
     class1.counter = 0;
     Timer timer1 = new Timer(new TimerCallback(class1.<StartTimer>b__0), 10, 0x3e8, 0x3e8);
 }

 private sealed class <>c__DisplayClass1 {
     // Methods
     public <>c__DisplayClass1() {}
     public void <StartTimer>b__0(object) {
         Console.WriteLine("Timer1 = {0}", this.counter++);
     }

     // Fields
     public int counter;
 }
}
```

المترجم قام باضافة كلاس جديد يحتوي على الanonymous method الذي قمنا بكتابته بالاضافة إلى متحول counter . لاحظ داخل التابع StartTimer فإنه تم انشاء هذا الكلاس الجديد واسناد قيمة للمتحول counter ومن ثم استخدام التابع عندم انشاء المؤقت .. أي بالمختصر فإن المترجم قام بعمل كلاس جديد لتغليف المتحولات التي تم تعريفها داخل التابع الأساسي.

جميل جدا بأن تتيح سي# تسهيلات عديدة في استخدام ال Anonymous Method . ولكن كنصيحة مني يفضل أن لا يزيد تابع ال anonymous method عن ثلاث أسطر!! لإن كثرة استخدامها وخصوصا إذا كان التابع كبير قد يجعل من مراجعة وتنقيح البرنامج صعبا جدا .

Generics

القسم الأول:

تعتبر ال Generics من أشهر الخصائص التي تمت اضافتها في الاصدار الثاني من سي#. وسوف أحاول هنا شرحها بشكل مختصر وسريع.

تمكننا هذه الخاصية من أستخدام أنواع معطيات عامة غير معروفة عند انشائنا للكود، ومن ثم بعد ذلك يمكننا تخصيص هذا الكود ليعمل مع أي معطيات كانت. مثلا ArrayList في الاصدار الأول تقبل اضافة اي كائن من نوع Object ، وبما أن جميع كائنات الدوت نيت تندرج من Object بشكل مباشر أو آخر فإنه يمكن اضافة اي كائن إلى الArrayList حيث يمكن اضافة ارقام ، تواريخ ، اسماء ، إلخ .. طبعا عند استخراجك لهذه الكائنات يجب أن تسند الكائن إلى النوع الصحيح قبل استخدامه ، ولا تنسى أنه عند اضافتك أي قيمة بسيطة struct type فيجب على الـruntime ان تقوم بعملية تغليف لهذه القيمة gBoxin قبل استخدامها مما ينتج عنه بعض البطئ.

لمحة أولى على كود مكتوب باستخدام Generics :

كود

```
 class MyList<T>
  {
      private T[] entries;
      private int count;

      public void Add(T elemment)
      { ... }

      public T this[int index]
      {
```

```
        get { ... }
        set { ... }
    }
}

class Program
{
    public static void Main()
    {
        List<string> StringList = new List<string>();
        StringList.Add("Hani");
        StringList.Add("Monir");
        StringList.Add(343);        // Compiler Error, cannot cast 343 to string.

        ArrayList OldList = new ArrayList();
        OldList.Add("Name");
        OldList.Add(523);
        OldList.Add(DataTime.Now);
    }
}
```

فوائد استخدام ال Generics :

1- الكفائة في اعادة استخدام الخوارزميات العامة . حيث يقوم مطور ما بكتابة خوارزمية عامة تعمل على أي نوع من أنواع المعطيات ومبرمج آخر يقوم باستخدامها مثلا على نوع معطيات DateTime.

2- مستخدم الخوارزمية يستطيع استخدامها بدون الحصول على الكود المصدري وهذا مختلف عن سي++ templates وجافا.

3- Type Safety: عند استخدام كود ال Generics يكون التعامل مع نوع معطيات واحد وليس مع نوع معطيات عام Object.

4- سرعة أداء البرنامج : عند استخدام أنواع معطيات بسيطة value type ، فلن يحصل اي من ال boxing عند استخدامك للgenerics لأنك تتعامل مباشرة مع النوع البسيط وليس مع Object

ماذا يمكن أن يكون نوع عام Generics:

أي من الأنواع التالية يمكن أن تكون عامة ذات بارامتر واحد أو أكثر: Classes, structs, interfaces, delegates, methods . بعض الأمثلة على استخدام nericsGe

كود

```
class ListNode<T> {
    public T _info;
    public ListNode<T> _next;
}

struct ComplextType<TFirst, TSecond> {
    TFirst _first;
    TSecond _second;
}

interface IComparable<T> {
    int CompareTo(T other);
}

void Swap<T>(ref T val1, ref T val2) {
    T temp = val1;
    val1 = val2; val2 = temp;
}
```

الأنواع البسيطة Generics في منصة دوت نيت:

كل الcollections في دوت نيت تم اعادة بنائها لتستفيد من الGenerics . تم وضع الأنواع الجديدة تحت Generics.ionsCollect.System وأيضا تم تغير اسمائها ولكنها تؤدي نفس الغرض:

كود

```
ArrayList    <==>  List<T>
Hashtable    <==>  Dictionary<TKey, TValue>
SortedList    <==>  StoredDictionary<TKey, TValue>
Stack  <==>  Stack<T>
Queue  <==>  Queue<T>
(n/a)  <==>  LinkedList<T>
```

ايضا تم اضافة Interfaces جديدة تدعم ال generics معظم هذه الInterfaces تشابه نفسها الموجودة حاليا:

كود

```
IList<T>
IDictionary<TKey, TValue>
ICollection<T>
IEnumuration<T>
IEnumerable<T>
IComparer<T>
IComparable<T>
```

نلاحظ أن البارامتر للGeneric كلها تبدأ بالحرف T ، هذا الأمر فقط متفق عليه بأنه يفضل بدأ هذا النوع من البارامتر بالحرف الكبير T . هذا الأمر مشابه لبدأ الInterface بالحرف الكبير I

أيضا تم اضافة العديد من التوابع ال static إلى الArray class .حيث يمكنك استخدام هذه التوابع مثلا من أجل ترتيب مصفوفة من أي نوع وهكذا . هذه التوابع هي:

AsReadOnly, BinarySearch, ConvertAll, Exists, Find, FindAll, FindIndex, FindLast, FindLastIndex, ForEach, IndexOf, LastIndexOf, Sort, TrueForAll

طبعا يفضل استخدام هذه التوابع خصوصا على الأنواع البسيطة struct types من أجل سرعتها.

انتاج الكود في الذاكرة للأنواع العامة Generics :

في السي++ عندما نستخدم ال Templates يقوم المترجم بانتاج كود جديد كليا لكل نوع نقوم باستخدامه مع ال Template . هذا قد يجعل حجم الملف التنفيذي كبير إذا أصرفنا في استخدام ال Templates . طبعا هذا الأمر قد تحسن جزئيا في دوت نيت .

آلية تنفيذ كود الدوت نيت تقوم بعملية انتاج كود جديد للأنواع Generics في الذاكرة عندما نستخدمها من أجل كل نوع بسيط مختلف. مثلا إذا كان لدينا <int>Stack و <byte>Stack فكل نوع سوف يتم توليده في قسم مختلف في الذاكرة . أما بالنسبة للأنواع من نوع reference فالدوت نيت سوف تقوم بتوليد كود واحد لها في الذاكرة وتستخدم نفس هذا القسم من الذاكرة لأي نوع من نوع reference .

هذه الطريقة منطقية ، فالدوت نيت يمكن أن تشارك الكود إذا كان النوع من نوع reference لأن النوع reference في الذاكرة عبارة عن مؤشر فقط (بالعادة 32 بت).

أما الأنواع البسيطة فقد تكون byte أو int(32 بت) و 32UInt أو struct أو غيرها . ولو فكرت بها أكثر لتجد أنه يوجد تعليمات آلة مختلفة من أجل كل نوع بسيط ، فعملية جمع عددين من نوع 32Int تختلف عن عملية جمع عددين من نوع 32UInt .

ويمكن اثبات هذا الموضوع بكتابة برنامج بسيط :

كود

```
class Test<T> {
     public void Print() {
         Console.WriteLine("Inside {0}", typeof(T));
         Debugger.Break();   // Watch the EIP CPU register here
     }
 }

 class Program
 {

     public static void Main()
     {
         Test<Int32> Int32Test = new Test<Int32>();
         Int32Test.Print();
         Test<UInt32> UInt32Test = new Test<UInt32>();
         UInt32Test.Print();
         Test<string> StringTest = new Test<string>();
         StringTest.Print();
         Test<object> ObjectTest = new Test<object>();
         ObjectTest.Print();
     }
 }
```

قم بمراقبة قيمة EIP ففي الحالة الأولى والثانية سوف يختلف : وهذا يعني أنا الكود تم انشائه في مكانين منفصلين في الذاكرة ، أما الحالة الثالثة والرابعة فسوف تجد أن EIP نفسها : وهذا يعني أن الكود متشارك عليه في الذاكرة .

القسم الثاني Constrains :

المترجم في حالة ال Generics يتأكد أن الكود المكتوب سوف يعمل في أي نوع من أنواع المعطيات المتاحة ، على سبيل المثال فإن التابع Swap يعمل فلى جميع أنواع المعطيات المتوفرة في ال CLR :

كود

```
void Swap<T>(ref T val1, ref T val2) {
     T temp = val1;
     val1 = val2; val2 = temp;
```

التابع Swap يعمل مع جميع أنواع المعطيات البسيطة struct والreference ، ويمكن استدعاء ToString.1val() لأن جميع الأنوع هي من النوع Object . طبعا لو حاولنا استدعاء التابع Read.1val() لأعطانا المترجم خطأ في الترجمة . لكن ماذا لو من بنائنا للخوارزمية فعلا أردنا فقط استخدام الأنواع من نوع Stream ، وبالتالي يمكننا استدعاء التابع Read .

يمكننا تحديد الأنواع التي يمكن استخدامها في ال Generics بما يسمى Constraints . في هذه الحالة التابع سوف يكون له حرية أكثر في استدعاء التوابع التي يكون مصمم لها . مثال :

كود

```
static T Min<T>(T arg1, T arg2) where T : IComparable {
    if (arg1.CompareTo(arg2) < 0) return arg1;
    return arg2;
}
```

في المثال السابق قمنا بتخصيص أن الأنواع التي يمكن استخدامها مع التابع Min يجب أن تكون من النوع IComparable ، وبهذه الحالة يمكن استدعاء 1arg . CompareTo.

القسم الثالث Generic Delegates

يمكننا استخدام Generics في تعريف ال Delegates . وهذه الميزة تساعد جدا فقد تسهل علينا في استخدام نفس ال Delegate مع عدة أنواع من المعطيات . وأكبر مثال على هذا هو ال EventHandler delegate . ففي السابق يجب علينا أن نعرف delegate خاص عندما ننشئ event جديد إذا كان يحتاج معلومات غير متوفرة في النوع EventArgs. الآن ليس من الداعي تعريف delegate جديد من أجل كل حالة:

كود

```
public delegate void EventHandler<T>(object sender, T e)
    where T : EventArgs;

public class MyForm {
    public event EventHandler<CursorPosEventArgs> CursorClick;
}
public class CursorPosEventArgs : EventArgs
{    public int X, Y;}
```

ملخص:

ال Generics تعتبر من اكبر الاضافات في الاصدار الجديد ، وإذا استخدمت بالشكل الصحيح سوف تفيد البرنامج في الأداء وفي اعادة استخدام الكود . ففي السابق كنت أستغرق أكثر من نصف ساعة من أجل كتابة strong typed collection أما مع ال generics فيمكنك أن تصل إلى نفس النتيجة بدون أي وقت يذكر .

بشكل عام يمكنك استخدام الGenerics في هذه الحالات:

1- عند استخدامك لل ectionsColl فهي سريعة جدا فيها

2- الخوارزميات التي يمكن أن نطبق عليها أكثر من نوع من المعطيات كخوارزميات الترتيب والبحث

3- أنواع المعطيات التي يمكن استخدام أكثر من نوع لمحتوياتها

4- العديد من الكود الحالي الذي كتبته ويستخدم Object يمكنك اعادة كتابته للاستفادة من Generics والتخصيص لنوع معطيات معين.

Nullable Types

[معتز شمس]

شكرا أخي هاني على هذه المبادرات ... أنا أيضا كنت أحضر لكتابة الاضافات الجديده منذ زمن بعيد ولكن كنت أتكاسل كثيرا خصوصا بأن المنتدى لا يتفاعل بالشكل الذي يشجع.

لذلك ... بعد اذنك سأقوم بكتابة بعض الأشياء.

فأنا أود التحدث عن الـ Nullable Types أولا ::

يمكننا أن نقوم في سي شارب 2 بكتابة كود بهذا الشكل ::

كود

```
If ( myInteger == null )
{
    //
}
```

ولكن كل شئ لكنه قوانينه .. فالمتغير myInteger الذي قمت باختباره اذا كان null أم لا في الكود السابق لم أعلن عنه بالطريقه العاديه ولكن بطريقه جديده في سي شارب 2 ... قمت بالاعلان هكذا ::

كود

```
Int? myInteger;
```

علامة الاستفهام الموجوده في الكود ليست خطأ مني ولكنها هى العلامه التي تستخدم عندما نريد الاعلان عن Value Type بأنه يمكن أن يسواي = null.

أو يمكن الاعلان عنه هكذا ::

كود

```
Nullable<int> myInteger;
```

الأمر يشبه قليلا بشئ ما في قواعد البيانات .. فعندما نحدد الـ Date Type لاحدى الأعمده في جدول في قاعدة البيانات ... فاننا نقوم بتحديد مثلا نوعه int وبعد ذلك نخبر الجدول بأنه من الممكن أن لا تحتوي على شئ = Allow Nulls.

يمكننا التعامل مع هذه الأنواع من خلال الـ Properties مثل Value التي ترجع لنا القيمه التي يحتويها وأيضا من خلال HasValue التي ترجع لنا Boolean لتفيد لنا ما اذا كان المتغير يحتوي فعلا على قيمه أو لا يحتوي على شئ (null).

مثال ::

كود

```
int? myInteger = null;

If(myInteger == null)
{
    //
}
myInteger  = 4;

int myInteger2 = myInteger.Value;

أو

int myInteger2 = (int)myInteger;

if (myInteger.HasValue)
{
    //True
}
```

ملحوظه :: الـ Value هى ReadOnly لذلك لا يمكنك كتابة ذلك ::

كود

```
int? myInteger = 4;
myInteger.Value = 56;
```

طبعا تلك الأمثله السابقه لا تعبر عن الغرض الذي تم من أجله اضافه الـ Nullable Types في سي شارب 2.

قرأت في الـ MSDN ووجدتهم هناك يقولون أنها من اجل التعامل مع قواعد البيانات ::

كود

```
avariable of nullable type can represent all the values of its underlying type, plus an
additional null value. This is particularly useful when dealing with databases and other data
types containing elements that may not be assigned a value, or which may not exist
```

وفعلا هذا الأمر جيد جدا كما توقعت ... فلن ينبغي علينا في سي شارب 2 التأكد من البيانات التي ترجع من قواعد البيانات من أنها null !

وهذا أنا علمته منذ زمن ولكني لم أجرب التعامل مع قواعد البيانات والاستفاده من الـ Nullable Types معها قبل ذلك اطلاقا , ولكن وأنا أكتب هذا الموضوع كنت قد قررت أن أضيف مثالا لها ولكني فشلت فشل كبير ,

انظر مثلا الى هذا الجدول الذي قمت بانشائه في SQL Server باسم ATMembers ::

Column Name	Data Type	Length	Allow Nulls
FName	char	10	
LName	char	10	
age	int	4	✓

وقمت باضافة بعض البيانات ::

FName	LName	age
Moutaz	Shams	221
Muhammed	A Aleem	<NULL>
Hussam	Mulhim	467
Anwar	ica	45
Bash	Mohandes	<NULL>

وبما أن الـ MSDN تقول أنه يمكننا استخدامها مع قواعد البيانات .. فبالتأكيد أنهم يقصدون كتابة الكود التالي باطمئنان في سي شارب 2 بدون اختبار اذا كانت القيمه تساوي DBNull أم لا ::

كود

```
SqlCommand myCom = new SqlCommand("Select * from ATMembers", myCon);
SqlDataReader myReader = myCom.ExecuteReader();

int? theAge = null;

while (myReader.Read())
{
theAge = myReader.GetInt32(2);
//SomeMethod(theAge.Value,........
//.......
}
```

ولكن للأسف يظهر لي NullValueException مباشرة عندما يجد أن قيمة احد الـ السن = null !!! مع ان المفروض أن المتغير theAge هو Nullable أصلا !!

معنى هذا وكأنه لم يتم اضافة شيئا جديدا فمازال يجب علينا اختبار اذا كان الـ age هو DBNull أم لا أولا كما نفعل في السي شارب الحاليه , أيضا قمت بتجربة استخدام Methods أخرى موجوده في الـDataReader وبعض المحاولات الأخرى ولكن بنفس النتيجه.

بعد ذلك قررت أن أجرب الاضافه الى الجدول فربما تكون الـ Nullable Types مدعومه هناك ولكن أيضا فشلت ::

قمت بعمل SP اسمها emberAddM للاضافه ::

كود

```
CREATE Proc AddMember @fname char(10),@lname char(10),@age int
As
insert ATMembers values(@fname,@lname,@age)
```

وبعد ذلك جربت الكود ::

كود

```
SqlCommand myCom = new SqlCommand("AddMember", myCon);
myCom.CommandType = CommandType.StoredProcedure;

string firstName = "Hany";
string lastName = "Atasy";
Nullable<int> age = null;
myCom.Parameters.AddWithValue("@fname", firstName);
myCom.Parameters.AddWithValue("@lname", lastName);
myCom.Parameters.AddWithValue("@age", age );
int rowsAffected = myCom.ExecuteNonQuery();
```

أيضا الكود لا يعمل ... طبعا هذا شئ غريب لا أفهمه حتى الآن .. ولكن من الواضح أن جميع الكلاسات المسؤوله عن التعامل مع قواعد البيانات لا تدعم الـ Nullable Types حتى الآن بالرغم من الـ MSDN يؤكدون فيه على أنها ميزه قويه عند التعامل مع قواعد البيانات ... فياريت لو يوجد من يعلم عن هذا الأمر .. يخبرنا به.

Iterators

كلنا نعرف IEnumerable و IEnumerator من أجل انشاء class يمكن معهما من استخدام foreach . بالنسبة للاصدار الجديد من سي# فالـ Iterators هي تسهيل من أجل انشاء class يدعم IEnumerable&IEnumerator بشكل تلقائي. وبشكل عام فإن ال Iterator هي أي تابع يسمح لك باستخدام foreach من غير أن تكتب كود منفصل لل IEnumerable .

دعنا نلقي نظرة على طريقة تحويل المترجم لعبارة foreach :

كود

```
// Original code:
foreach (int val in col) {
    OutputValue(val);
}

// It will be converted to:
IEnumerator e = col.GetEnumerator();
try {
  int val;
  while (e.MoveNext()) {
     val = (int)e.Current;
     OutputValue(val);
  }
}
finally {
  IDisposable d = e as IDisposable;
```

```
  if (d != null) d.Dispose();
}
```

إذا أردنا أن ننشئ كلاس يدعم eachfor في سي# الاصدار الأول فيجب أن نكتب كود للكلاس من أجل دعم الواجهة IEnumerable ، فيها فقط تابع واحد وهو GetEnumerator يقوم بارجع كائن من نوع IEnumerator . يجب أن ننشئ كلاس آخر يدعم الواجهة IEnumerator ونكتب كود من أجل كل من التوابع MoveNext و Current و esetR . العملية تحتاج إلى كتابة الكثير من الكود . مع الـ Iterators العملية أسهل ، فكل الذي عليك هو أن أن تكتب تابع يكون عبارة عن Iterator في الكلاس وداخله تكتب كود من أجل ارجاع القيم في المصفوفة أو ال collection .

لقد تم اضافة كلمة محجوزة جديدة في سي# الاصدار الثاني وهي yield ، ويتم استخدامها كالتالي:

كود

```
yield return expression;
yield break;
```

هذه الكلمة تستخدم داخل أي تابع يرجع نوع IEnumerable او IEnumerator ، وعندما يراها المترجم يعتبر المترجم أن التابع عبارة عن iterator . و yield تستخدمم من أجل ارجاع أي قيمة تريد من الcollection. مثلا أنظر الكود التالي:

كود

```
// We are using the generic version of IEnumerable
class MyNumCollection : IEnumerable<int>
{
   public IEnumerator<int> GetEnumerator() {
      yield return 0;   // first value is 0
      yield return 1;   // second value is 1
      yield return 2;   // third value is 2
      yield return 3;   // forth value is 3
   }

   // This is in place for backword compatibility with non-generic IEnumerable
   IEnumerator IEnumerable.GetEnumerator() {
      return (this as IEnumerable<int>).GetEnumerator();
   }
}

class Program
{
   static void Main(string[] args)
   {
      MyNumCollection col = new MyNumCollection();
      foreach (int val in col)
         Console.WriteLine(val);
   }
}
```

لاحظ كيف استخدمنا yield من داخل GetEnumerator ، نخبر المترجم هنا أننا نريد ارجاع القيمة 0 اولا ثم 1 ثم 2 ثم 3 .. وعند تنفيذ حلقة foreach فالنتيجة سوف نطبع القيمة 0 و 1 و2 و3 على الشاشة . المترجم وراء الكواليس يقوم بانشاء كلاس يدعم الواجهة IEnumerator . يمكن رؤية هذا الكلاس عن طريق NET . Reflector . وأنصح برؤيته لمعرفة كيف يتعامل المترجم مع ال Iterators .

طبعا يمكن كتابة أي نوع كود داخل ال GetEnumerator وتستخدم yield في أي مكان ، وعند التنفيذ المترجم يحتفظ بالحالة الأخيرة لyeild . مثلا أنظر الكود التالي:

كود

```
// Generate 10 random numbers
class RandomGenerator : IEnumerable<int>
{
   public IEnumerator<int> GetEnumerator() {
      Random rnd = new Random();
      for (int i = 0; i < 10; i++)
         yield return rnd.Next(1, 1001);
   }

   IEnumerator IEnumerable.GetEnumerator() {
      return (this as IEnumerable<int>).GetEnumerator();
   }
}

class Program
{
   static void Main(string[] args)
```

```
    {
       RandomGenerator col = new RandomGenerator();
       foreach (int val in col)
          Console.WriteLine(val);
    }
 }
```

الأمر الجميل في الكود السابق هو استخدام yield داخل حلقة .

المثال التالي يبين استخدام IEnumerable Iterator .. وهي عبارة عن تابع يرجع أرقام فيبوناشي والآخر يرجع الأرقام الأولية بين رقمين.

كود

```
class Program
 {
    // Calculate Fibonacci for (n)
    static IEnumerable<int> Fibonacci(int n) {
       if (n < 0)
          throw new ArgumentOutOfRangeException("The value n must be >= 0");
       if (n >= 0)
          yield return 1;
       if (n >= 1)
          yield return 1;
       int prev1 = 1, prev2 = 1;
       for (int i = 2; i <= n; i++) {
          int result = checked(prev1 + prev2);
          yield return result;
          prev1 = prev2;
          prev2 = result;
       }
    }

    // Returns the prime numbers between (from) and (to)
    static IEnumerable<int> Primes(int from, int to) {
       if (from < 0 || to < 0)
          throw new ArgumentOutOfRangeException("from and to must be greater than 0");
       if (from > to)
          throw new ArgumentException("to must be larger than from or equal");
       if (from <= 1) from = 2;  // 0, 1 are not primes
       for (; from <= to; from++)
       {
          if (from % 2 == 0) continue; // even numbers are not primes
          bool isPrime = true;
          for (int i = 3; i <= (int)Math.Sqrt(from); i++) {
             if (from % i == 0) {
                isPrime = false;
                break;
             }
          }
          if (isPrime) yield return from;
       }
    }

    static void Main(string[] args)
    {
       Console.WriteLine("Fibonacci");
       foreach (int val in Fibonacci(5))
          Console.WriteLine(val);

       Console.WriteLine("Primes");
       foreach (int prime in Primes(1, 15))
          Console.WriteLine(prime);
    }
 }
```

يمكنك استخدام مفهوم التعاودية على مجموعة في ال Iterator . أنظر المثال التالي:

كود

```
class Program
 {
    static IEnumerable<string> SubDirs(string root) {
       yield return root;

       foreach (string subdir in Directory.GetDirectories(root))
          foreach (string s in SubDirs(subdir))
             yield return s;
```

```
    }

    static void Main(string[] args)
    {
       foreach (string dir in SubDirs(@"c:\windows"))
          Console.WriteLine(dir);
    }
 }
```

نقاط هامة يجب أن نعرفها عند استخدام Iterators

- معظم الأنواع في مكتبة الدوت نيت عندما ترجع كائن في GetEnumerator فإنها تعيد struct بدلا من class . وبالتالي فإن كائن الIEnumerator سوف ينشئ على المكدس وهذا اسرع من انشاء الكائن في ال heap . لكن لسوء الحظ فإن ال Iterators ترجع class بدلا من struct .

- كتابة foreach سهل ورائع عند تعاملك مع collections التي يمكن أن تغير حجمها أثناء عملية سرد المحتويات أو التي يكون من الصعب أو مكلف إيجاد عدد المحتويات مسبقا. الثمن المدفوع عند استخدامك foreach هو أنه سوف يتم انشاء كائن في الذاكرة heap (في معظم الأحيان) وسوف يتم استدعاء تابعين هما MoveNext و Current من أجل كل عنصر.

الجميل في الأمر أنه تم اضافة توابع جديدة في معظم الأنواع في دوت نيت . وهذه التوابع تغنيك عن استخدام Iterators . مثلا Array و List في دوت نيت اضافت التوابع التالية:

كود

```
public void ForEach(Action<T> action)
public List<T> FindAll(Predicate<T> match)
public void Sort(Comparision<T> comparision)
public List<U> ConvertAll<U>(Convert<T, U> converter)
```

التوابع السابقة تستخدم delegates تم تعريفها جديدا في دوت نيت وهي:

كود

```
// مة معينة على الكائنمن اجل تنفيذ مه
public delegate void Action<T>(T obj);

// إذا كان يتبع شرط معين أم لا من أجل فحص الكائن الممرر وتحديد
public delegate Boolean Predicate<T>(T obj);

// من أجل مقارنة قيمتين وتحديد العلاقة بينهما
public delegate Int32 Comparision<T>(T x, T y);

// آخر من أجل تحويل القيمة من نوع إلى
public delegate U Converter<T, U>(T from)
```

والأمثلة التالية توضح كيف يمكن استخدام التوابع السابقة وأيضا لاحظ استخدام anonymous methods فيها:

كود

```
 class Program
 {
    static void Main(string[] args)
    {
       List<string> names = new List<string>(
          new string[] { "Nizar", "Hani", "Monir", "Badr" });

       // Sort into: Badr, Hani, Monir, Nizar
       names.Sort(delegate(string a, string b) {
          return string.Compare(a, b);
       });

       // Output the sorted list
       names.ForEach(delegate(string name) {
          Console.WriteLine(name);
       });

       // Finds only the names that has the letter 'a'
       names = names.FindAll(delegate(string name) {
          return name.IndexOf('a') >= 0;
       });

       // Convert all the names into upper case
       names = names.ConvertAll<string>(delegate(string name) {
          return name.ToUpper();
       });

       // Output BADR, HANI, NIZAR
       names.ForEach(delegate(string name) {
```

```
            Console.WriteLine(name);
        });
    }
```

Static Classes and more -Partial Types

تعريف نوع بشكل جزئي Partial Types

هذه الخاصية الجديدة التي تمت اضافتها في سي# 2 ، تمكنك من تعريف class او struct او interface في عدة ملفات مصدرية . حيث جزء من التعريف في ملف وجزء في ملف آخر .

هذه الميزة مهمة في عدة نواحي:

1- إذا كان التعريف سوف يعمل عليه مطورين فكل مطوريطور جزء من النوع

2- إذا أردت اضافة كود على أنواع تم انشائها بشكل آلي عن طريق visual studio ، فمثلا يمكنك الاضافة على strong typed dataset class و form class .

طبعا هذه الخاصية هي اضافة للمترجم والمترجم أثناء الترجمة يقوم بدمج الأنواع الجزئية في ملف واحد . أما بنية الدوت النيت فلا تعرف أي شئ عن الأنواع الجزئية .

تمت اضافة كلمة محجوزة جديدة للغة partial يتم اضافتها على كل نوع يمكن أن يكون جزئي .. إذا أردنا تعريف الكلاس Student بشكل جزئي في ملفين فكل ملف يجب أن يحتوي على التعريف التالي :

كود

```
File1.cs
======
public partial class Student
{
     int Id {}
}

File2.cs
======
public partial class Student
{
     string Name {}
}
```

Property Accessor Accessibility

هذه الميزة جديدة في سي# 2 .. وهي موجودة بالأساس في NET.VB .. حيث الآن يمكنك تغيير صلاحيات الوصول إلى كل من put accessor &get في ال property . ففي معظم الأحيان نريد أن نجعل get من صلاحيات public و set من صلاحيات protected .

مثال على هذا:

كود

```
public class MyType {
    private string _name;
    public string Name {
 get { return _name; }
 protected set { _name = value; }
    }
}
```

كلاس من نوع static .

بعض الكلاسات التي ننشئها تحتوي فقط على توابع تابعة للكلاس نفسه وليس للكائن أي أقصد توابع من نوع static . فالآن في سي# 2 ، يمكننا أن نعلم هذا الكلاس على أنه static إذا كنا متأكدين أن جميع التوابع في هذا الكلاس من نوع static .. وبالتالي إذا سقطت سهوا الكلمة static في كلاس معين فإن المترجم سوف ينبهنا عن خطأ

Friend Assemblies :

هذه الخاصية التي تم اضافتها جديدا تمكن assembly معينة من الوصول إلى التعاريف الداخلية internal في assembly اخرى . وهي ببساطة أن نضيف attribute في ال assembly الأساسية ومن خلالها نعرف ال assembly الصديقة بأنها يمكن أن تصل إلى التعاريف الـinternals .

مثال:

كود

```
1stAssembly
==========

using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("2ndAssembly, PublicKeyToken=21231231ababab")]
Internal class InternalType { … }
```

```
2ndAssembly
==========

class Program
{
    static void Main()
    {
     InternalType t = new InternalType();
    }
}
```

#pragma Warning

يمكنك من خلالها تعطيل وتفعيل رسائل ال warning التي يظهرها المترجم . ويمكنك تحديد رقم الwarning من أجل هذا .

كود

```
#pragma warning disable 332  // disable warning 332
#pragma warning enable 332  // enable warning 332
#pragma warning enable  // enable all warnings
#pragma warning disable  // disable all warnings
```

الرابط المباشر

http://www.arabteam2000-forum.com/index.php?showtopic=77397