التعامل مع العناصر Value Types and Reference Types

المؤلف: Mohammad Elsheimy

رابط الدرس: http://www.arabteam2000-forum.com/index.php?showtopic=198848

محتويات هذا الدرس:

مقدمة

يعتبر فهم كيفية التعامل مع العناصر Objects من أهم المواضيع التي يجب التطرق إليها. وبظهور بيئة الدوت نت في صيف 2001 تقريبا, ظهر مفهوم البرمجة الموجهة للكائنات بشكل واضح, وأصبحت كلمة Object من أهم الكلمات وأكثرها التي تتعامل معها يوميا.

سوف نقوم في هذا الدرس بشرح لأنواع البيانات في بيئة الدوت نت والتي تنقسم إلى نوعين, وسوف نقوم أيضا بشرح كل نوع على حدة, ثم نقوم بمقارنتهما معا. وسوف نقوم أيضا بشرح العنصر الأم لجميع العناصر الأخرى وهو System.Object وبناءا عليه سنقوم بشرح أساليب التحويل بين نوعين البيانات.

وأما الجزء الثاني من الدرس فيتكلم عن مدة حياة العناصر وكيف يتحرر العنصر من الذاكرة وما هي ذاكرة Stack و Heap وكيفية التعامل مع ذاكرة Heap من خلال.Garbage Collector

أنواع العناصر في بيئة الدوت نت

تنقسم العناصر في بيئة الدوت نت إلى نوعين:

- Value-Based Types: العناصر التي تمرر بالقيمة.
- Reference-Based Types: العناصر التي تمرر بالمرجعية.

Value-Based Types

ويدخل تحت هذا النوع جميع البيانات الرقمية (مثل Integer و Double وغيرها), وكذلك النصية (مثل String و Char) وكذلك العدادات Enumerations وأيضا.Structures

ويتم تسجيل بيانات هذا النوع في القسم الأول من الذاكرة وهو .Stack ولذلك يتم إزالة هذه البيانات من الذاكرة فور انتهاء قطعة الكود التي تستخدم هذا النوع.

مثال:

```
'VB .NET Code
    'لا ننسى جميع البيانات العددية هي
    'Value types
    Public Sub Main()
        Dim i As Integer = 0
        Console.WriteLine(i)
    End Sub
    'عند هذه النقطة يتم إزالة المتغير
    'i
    'من الذاكرة
```

```
        // C# Code
        // لا ننسى جميع البيانات العددية هي
        // Value types
        public static void Main()
        {
            int i = 0;
            Console.WriteLine(i);
        }
        // عند هذه النقطة يتم إزالة المتغير
        // i
        // من الذاكرة
```

وعند إسنادك لقيمة عنصر من نوع Value Type إلى عنصر آخر مماثل, يتم إنشاء نسخة من قيمة العنصر الأول إلى العنصر الثاني.

مثال:

```
    'VB .NET Code
        Dim i As Integer = 100
        Dim j As Integer = i
        ' بعد الإسناد التالي
        'i
        'ما زالت تساوي 100
        j = 300
```

```
        // C# Code
        int i = 100;
        int j = i;
        // بعد الإسناد التالي
        // i
        // ما زالت تساوي 100
        j = 300;
```

وهذه القاعدة تنطبق أيضا على Structures و Enumerations كما قنا سبقا. وكما في المثال:

```
'VB .NET Code
     Public Sub Main()
        Dim struct1 As New MyStruct
        struct1.x = 100
        struct1.y = 100

        Console.WriteLine("-> Assigning struct2 to struct1")
        Dim struct2 As MyStruct = struct1

        'struct1
        Console.WriteLine("struct1.x = " & struct1.x)
        Console.WriteLine("struct1.y = " & struct1.y)
        'struct2
        Console.WriteLine("struct2.x = " & struct2.x)
        Console.WriteLine("struct2.y = " & struct2.y)
```

```
        Console.WriteLine("-> Changing struct2.x to 900")
        struct2.x = 900

        Console.WriteLine("-> X again")
        Console.WriteLine("struct1.x = " & struct1.x)
        Console.WriteLine("struct2.x = " & struct2.x)
    End Sub

    Public Structure MyStruct
        Public x, y As Integer
    End Structure
```

```
    // C# Code
    public static void Main()
    {
        MyStruct struct1 = new MyStruct();
        struct1.x = 100;
        struct1.y = 100;

        Console.WriteLine("-> Assigning struct2 to struct1");
        MyStruct struct2 = struct1;

        // struct1
        Console.WriteLine("struct1.x = " + struct1.x);
        Console.WriteLine("struct1.y = " + struct1.y);
        // struct2
        Console.WriteLine("struct2.x = " + struct2.x);
        Console.WriteLine("struct2.y = " + struct2.y);

        Console.WriteLine("-> Changing struct2.x to 900");
        struct2.x = 900;

        Console.WriteLine("-> X again");
        Console.WriteLine("struct1.x = " + struct1.x);
        Console.WriteLine("struct2.x = " + struct2.x);
    }

    public struct MyStruct
    {
        public int x, y;
    }
```

كما شاهدت في المثال السابق, عند إسناد struct1 إلى struct2 تصبح عندنا نسختان من القيمة في struct1 وهذا هو معنى-Value Based Types.

Reference-Based Types

يعتبر هذا النوع ضد النوع الأول بشكل كامل حيث يدخل تحت هذا النوع جميع الـ Classes, يتم تسجيل هذا النوع في المنطقة الثانية في الذاكرة وهي .Heap ولا تستطيع تحديد متى سيتم إلغاء هذا النوع من الذاكرة حيث أن Garbage Collector (GC) هي المسؤولة عن إلغائه (سيتم شرح GC لاحقا).

ويظهر الفرق واضحا في بين النوعين في أن عند إسنادك أي عنصر من هذا النوع إلى عنصر آخر يتم إنشاء نسخة من النوع الأول

وتتغير النسخة المنشأة بتغيير النسخة الأصلية والعكس صحيح, أي أنك لا تستطيع إنشاء نسختين منفصلتين, ولكن تنشئ نسخة مرتبطة بالنسخة الأولى. تأمل هذا المثال:

```
'VB .NET Code
    Public Sub Main()
        Dim class1 As New MyCls
        class1.x = 100
        class1.y = 100

        Console.WriteLine("-> Assigning class2 to class1")
        Dim class2 As MyCls = class1

        'struct1
        Console.WriteLine("class1.x = " & class1.x)
        Console.WriteLine("class1.y = " & class1.y)
        'struct2
        Console.WriteLine("class2.x = " & class2.x)
        Console.WriteLine("class2.y = " & class2.y)

        Console.WriteLine("-> Changing class2.x to 900")
        class2.x = 900

        Console.WriteLine("-> X again")
        Console.WriteLine("class1.x = " & class1.x)
        Console.WriteLine("class2.x = " & class2.x)
    End Sub

    Public Class MyCls
        Public x, y As Integer
    End Class
```

```
    // C# Code
    public static void Main()
    {
        MyCls class1 = new MyCls();
        class1.x = 100;
        class1.y = 100;

        Console.WriteLine("-> Assigning class2 to class1");
        MyCls class2 = class1;

        // class1
        Console.WriteLine("class1.x = " + class1.x);
        Console.WriteLine("class1.y = " + class1.y);
        // class2
        Console.WriteLine("class2.x = " + class2.x);
        Console.WriteLine("class2.y = " + class2.y);

        Console.WriteLine("-> Changing class2.x to 900");
        class2.x = 900;

        Console.WriteLine("-> X again");
        Console.WriteLine("class1.x = " + class1.x);
        Console.WriteLine("class2.x = " + class2.x);
    }
    public struct MyCls
    {
        public int x, y;
    }
```

فنلاحظ أن النتيجة تكون أن class1.x = class2.x. حيث أنك عند تغييرك لـ class1 تتغير class2 والعكس صحيح فعند تغييرك لـ class2تتغير class1. فهذا هو الفرق بين البيانات المرجعية Reference Types والبيانات بالقيمة Value Types.

قاعدة : يمكن لـ Value Type أن تحتوي على Reference Type. وتنطبق جميع قواعد Reference Types على الـ Reference Typeالداخلية.

مثال على القاعدة السابقة:

```
'VB .NET Code
    Public Class MyRefType
        Public x, y As Integer
    End Class

    Public Structure MyValType
        Public i As Integer
        Public RefType As MyRefType

        Public Sub New(ByVal num As Integer)
            i = num

            RefType = New MyRefType
            RefType.x = 100
            RefType.y = 100
        End Sub
    End Structure
```

```
    // C# Code
    public class MyRefType
    {
        public int x, y;
    }

    public struct MyValType
    {
        public int i;
        public MyRefType RefType;

        public MyValType(int num)
        {
            i = num;

            RefType = new MyRefType();
            RefType.x = 100;
            RefType.y = 100;
        }
    }
```

فعند محاولتك لتغيير العنصر RefType من خلال أي كود خارجي يتم معاملته كأي Class آخر كما رأينا في المثال قبل السابق. ولا يعتبر بوجوده داخل Value Type.

مقارنة بين Value Types وReference Types

التالي هو مقارنة بين النوعين باختصار:

- أين يتم تخزين بيانات هذا النوع؟
Value-Types: في الجزء الأول من الذاكرةStack :
Reference-Types: في الجزء الآخر من الذاكرةHeap :

- كيف يتم الإسناد إلى هذا النوع؟
Value-Types: يتم إنشاء نسخة من القيم الداخلية يتم إسنادها إلى النسخة الأخرى
Reference-Types: يتم إنشاء نسخة مرجعية من النسخة الأصلية تتغير بتغيير النسخة الأصلية والعكس صحيح

- ما هي أنواع البيانات التي تدخل تحت هذا النوع؟
Value-Types: جميع البيانات الرقمية وكذلك النصية وأيضا Enumerations وStructures
Reference-Types: Classes

- ما هو العنصر الذي ينحدر/يتوارث (inherits) منه هذا النوع؟
Value-Types: هذا النوع ينحدر/يتوارث (inherits) منSystem.ValueType
Reference-Types: هذا النوع ينحدر/يتوارث (inherits) منSystem.Object

- هل يمكن أن يكون هذا النوع قاعدة Base للأنواع الأخرى؟ بمعنى آخر: هل يمكن أي نوع آخر أن ينحدر/يتوارث من هذا النوع؟
Value-Types: لا, لا يمكن ذلك
Reference-Types: نعم, يمكنك ذلك إلا في بعض الحالات كتحديد الـ Class كـ) NotInheritable في (VB .NET أو sealed في(C#)

- عند تمرير هذا النوع كمعامل Argument كيف يمرر؟
Value-Types: تمرر نسخة من قيمة هذا النوع كمدخلات
Reference-Types: يمرر العنصر نفسه وليس نسخة منه

- هل يمكننا إزالة هذا النوع من الذاكرة باستخدام) ()Object.Finalize سوف يتم شرح هذه الدالة لاحقا)؟
Value-Types: لا, حيث أن هذا النوع لا يتم تسجيله في Heap مطلقا
Reference-Types: نعم, مباشرة

- متى سوف يتم تحرير الذاكرة الخاصة بهذا النوع؟
Value-Types: عند انتهاء قطعة الكود التي تستخدم هذا النوع
Reference-Types: عند إجراء عملية) Garbage Collection سوف يتم شرح هذه العملية لاحقا(

العنصر الأمSystem.Object

يعتبر العنصر System.Object من أهم العناصر في بيئة الدوت نت حيث أن جميع العناصر الأخرى تنحدر من هذا العنصر.

قاعدة :جميع العناصر تنحدر من System.Object حتىSystem.ValueType!

ونقصد بالمصطلح تنحدر القاعدة الأولى من قواعد البرمجة الموجهة للكائنات ,(OOP) Object-Oriented Programming ألا وهي التوارث.Inheritance

ويحتوي هذا العنصر System.Object على مجموعة من الدوال والتي يمكنك استخدامها إما عن طريق العنصر نفسه (System.Object.MethodName)أو عن طريق العناصر المنحدرة من هذا العنصر (مثل:(System.String.MethodName :

- Equals:

تقوم هذه الدالة بأخذ عنصرين كمدخلات ثم تقوم بإرجاع القيمة) True أو true في (#C إذا كانا هذين العنصرين يحتويان على نفس القيمة.

- GetHashCode:

تقوم هذه الدالة بإرجاع رقم يميز هذا العنصر ويختلف باختلاف محتوى هذا العنصر.
لاحظ المثال التالي:

```
'VB .NET Code
     Dim s1 As String = "A"
     Dim s2 As String = "B"

     Console.WriteLine("s1.GetHashCode = {0}", s1.GetHashCode)
     Console.WriteLine("s2.GetHashCode = {0}", s2.GetHashCode)
```

```
     // C# Code
     string s1 = "A";
     string s2 = "B";

     Console.WriteLine("s1.GetHashCode = {0}", s1.GetHashCode());
     Console.WriteLine("s2.GetHashCode = {0}", s2.GetHashCode());
```

:GetType
تقوم هذه الدالة بإرجاع عنصر من نوع System.Type ويحتوي على وصف كامل للعنصر المستخدم.

- :ToString

تقوم هذه الدالة بإرجاع قيمة من نوع string) String في (#C تمثل العنصر المستخدم. يمكن أن تكون هذه القيمة هي الاسم بالكامل لهذا العنصر, مثل ,Namespace.ClassName :ويمكن أن تكون أي نص آخر يصف محتويات هذا العنصر.

- :MemberwiseClone

تقوم هذه الدالة بإرجاع نسخة طبق الأصل من العنصر المستخدم.

- :Finalize

وهي دالة مهمة جدا. حيث تستخدم مع العناصر من نوع .Reference Type حيث تقوم بيئة الدوت نت باستدعاء هذه الدالة قبل إزالة هذا العنصر مباشرة من الجزء الآخر من الذاكرة وهو .Heap ويمكنك عمل Overriding لهذه الدالة وإضافة الكود الخاص بك لإغلاق ملف معين مثلا أو لحذف ملفات مؤقتة مستخدم من قِبل العنصر مثلا.

التحويل من Value Type إلى Reference Type والعكس

بما أنك لديك نوعين من العناصر أحدهما بالقيمة والآخر بالمرجعية فربما تحتاج لتحويل العنصر بالقيمة ليكون بالمرجعية أو العكس.

وعملية التبديل من Value Type إلى Reference Type هذه تسمى .Boxing وأما عملية التبديل من Reference Type إلى Value Typeفتسمى.Unboxing

عمليةBoxing
وتظهر هذه العملية في المثال التالي -لاحظ الملاحظات في الكود:

```
'VB .NET Code
     'Boxing
     Dim num As Byte = 25
     Dim numObj As Object = num
```

```
     // C# Code
     // Boxing
     byte num = 25;
     object numObj = num;
```

Boxing: هي عملية تحويل العنصر من نوع Value Type إلى نوع ,Reference Type وحينئذ ينطبق عليه جميع قواعد Reference Types.

عمليةUnboxing
وتظهر هذه العملية في المثال التالي -لاحظ الملاحظات في الكود:

```
'VB .NET Code
     'Boxing
     Dim num As Byte = 25
     Dim numObj As Object = num
     'Unboxing
     Dim anotherNum As Byte = CByte(numObj)
```

```
     // C# Code
     // Boxing
     byte num = 25;
     object numObj = num;
     // Unboxing
     byte anotherNum = (byte)numObj;
```

Unboxing: هي عملية تحويل العنصر من نوع Reference Type إلى نوع Value Type, وحينئذ تنطبق عليه جميع قواعد Reference Types.

قاعدة :يجب عليك معرفة نوع محتوى العنصر Object وإلا سوف يتم حدوث الخطأ InvalidCastException أثناء وقت التشغيل إذا تم التحويل إلى عنصر آخر, وخصوصا في.#C

مثال على القاعدة السابقة:

```
'VB .NET Code
     'Boxing
     Dim num As Byte = 25
     Dim numObj As Object = num
     'Unboxing
     'VB .NET does not throw an exception
     Dim anotherNum As String = CStr(numObj)
```

```
     // C# Code
     // Boxing
     byte num = 25;
     object numObj = num;
     // Bad Unboxing
     string anotherNum = (string)numObj;
```

نلاحظ أن في المثال السابق لم يتم حدوث خطأ في كود ,VB .NET بينما حدث الخطأ InvalidCastException في كود.#C

يمكنك تغيير الوضع الأساسي لـ VB.NET كي يقوم بفحص العنصر مقدما ويقوم بإحداث الخطأ إذا تم التحويل إلى عنصر آخر عن طريق إما إضافة السطر Option Explicit On إلى بداية صفحة الكود أو عن طريق تغيير هذه الخاصية في إعدادات المشروع في خيارات Compile.

عملية Boxing التلقائيةAutomatic Boxing

أحيانا يقوم وقت التشغيل بعمل عملية Boxing تلقائيا من غير الحاجة إلى وضع أكواد خاصة لهذه العملية.
مثال:

```
'VB .NET Code
     Public Sub Main()
          'Boxing
          Dim num As Byte = 25

          'Automatic Boxing
          UseObj(num)
     End Sub

     Public Sub UseObj(ByVal obj As Object)
          Console.WriteLine(obj.GetType)
          Console.WriteLine(obj.ToString)
```

```
        ' يمكنك عمل عملية
        'Unboxing
        في نفس السطر'
        Console.WriteLine _
            (CByte(obj).GetTypeCode.ToString)
    End Sub
```

```
    // C# Code
    public static void Main()
    {
        // Boxing
        byte num = 25;

        // Automatic Boxing
        UseObj(num);
    }

    public static void UseObj(object obj)
    {
        Console.WriteLine(obj.GetType());
        Console.WriteLine(obj.ToString());
        // يمكنك عمل عملية
        // Unboxing
        // في نفس السطر
        Console.WriteLine(((byte)obj).GetTypeCode());
    }
```

نلاحظ في المثال السابق أن كود VB .NET يعمل بشكل صحيح, بينما يظهر خطأ في كود .#C حيث أن #C لا تسمح بعملية Unboxing التلقائية.

Boxingو !Unboxing ماذا يحدث وراء الستار؟

تستطيع كتابة الكود التالي كمثال لعمليتي Boxing وUnboxing:

```
'VB .NET Code
    Public Sub Main()
        'Boxing
        Dim num As Byte = 25
        Dim numObj As Object = num
        'Unboxing
        Dim anotherNum As Byte = CByte(numObj)
    End Sub
```

```
        // C# Code
        public static void Main()
        {
            // Boxing
            byte num = 25;
```

```
            object numObj = num;
            // Unboxing
            byte anotherNum = (byte)numObj;
        }
```

كيف يتعامل VS .NET مع كودCIL

كما نعلم أنك عندما تبدأ عملية Build للمشروع الخاص بك فـ VS .NET يستخدم برامج Compilers لتحويل الكود إلى صيغة أخرى وهي لغة (Intermediate Language) IL وهي التي يستطيع وقت التشغيل Runtime التعامل معها, فيقوم هذا الـ Compiler بقراءة الكود الخاص بك وترجمته إلى ملف تنفيذي Executable أو حتى مكتبة- Library حسب نوع المشروع-, وهذا الملف التنفيذي يحتوي على كود IL الخاص الذي تم ترجمته, فإذا كان مشروعك بـ VB .NET فـ VS .NET يستخدم برنامج VBC.EXE وهو برنامج Compiler الخاص بتحويل الكود الذي كتبته إلى الكود المماثل له في لغة IL, وأما C# فتستخدم برنامج CSC.EXE لنفس العملية.

ولغة Intermediate Language أو كما تختصر إلى IL أو CIL (Common IL) أو MSIL (Microsoft IL) هي لغة عامة لا تختلف باختلاف لغة مشروعك -إلا في حالات نادرة كما سنلاحظ في الأمثلة القادمة!-

بمعنى, أنه عند تحويل كود VB .NET إلى كود IL يكون الكود مماثل تقريبا لنفس كود IL المحول من C# أو MC++ أو COBOL أو غيرها من اللغات التي تستخدم.NET Framework.

والآن, لنشاهد كود IL الناتج من الكود السابق.

VB.NET CIL

هذا هو كود CIL الناتج من كود) VB.NET أحيانا يختلف الكود قليلا:(

```
.method public static void  Main() cil managed
     {
       .entrypoint
       .custom instance void [mscorlib] System.STAThreadAttribute::.ctor() = ( 01 00
00 00 )
       // Code size       18 (0x12)
       .maxstack  1
       .locals init ([0] uint8 anotherNum, [1] uint8 num, [2] object numObj)
       IL_0000:  ldc.i4.s   25
       IL_0002:  stloc.1
       IL_0003:  ldloc.1
       IL_0004:  box        [mscorlib]System.Byte
       IL_0009:  stloc.2
       IL_000a:  ldloc.2
       IL_000b:  call uint8 [Microsoft.VisualBasic]
Microsoft.VisualBasic.CompilerServices.Conversions::ToByte(object)
       IL_0010:  stloc.0
       IL_0011:  ret
     } // end of method MainClass::Main
```

لاحظ هذا السطر والذي يحتوي على أمر عمليةBoxing :

```
IL_0004:  box        [mscorlib]System.Byte
```

C# CIL

وهذا هو الكود الناتج من C# (أحيانا يختلف الكود قليلا):

```
.method public hidebysig
               static void  Main() cil managed
    {
      .entrypoint
      // Code size       18 (0x12)
      .maxstack  1
      .locals init ([0] uint8 num, [1] object numObj)
      IL_0000:  ldc.i4.s   25
      IL_0002:  stloc.0
      IL_0003:  ldloc.0
      IL_0004:  box        [mscorlib]System.Byte
      IL_0009:  stloc.1
      IL_000a:  ldloc.1
      IL_000b:  unbox.any  [mscorlib]System.Byte
      IL_0010:  pop
      IL_0011:  ret
    } // end of method MainClass::Main
```

لاحظ هذا السطر والذي يحتوي على أمر عمليةBoxing :

```
IL_0004:  box        [mscorlib]System.Byte
```

لاحظ هذا السطر والذي يحتوي على أمر عمليةUnboxing :

```
IL_000b:  unbox.any  [mscorlib]System.Byte
```

ونلاحظ تشابه الكودين إلى حد كبير, بينما يختلفان في هذه العملية في أن VB .NET يستخدم دالة لعملunboxing :

```
  IL_000b:  call uint8 [Microsoft.VisualBasic]
Microsoft.VisualBasic.CompilerServices.Conversions::ToByte(object)
```

وهذه الدالة هي الدالة ToByte الموجودة في Microsoft.VisualBasic.CompilerServices.Conversations وهي Module خاصة بالتحويلات في VB .NET وهي الـ Module موجودة في Assembly التي تحمل الاسم التالي- Microsoft.VisualBasic وهي واحدة من مميزات VB .NET عن C# حيث أن هذه الـ Assembly تحتوي على الـ Modules والـ Classes والتي كانت في VB 6 والتي تساعد مبرمج VB .NET لإنجاز العديد من المهام في أسرع وقت.-

وعلى النقيض من ذلك فـ C# ليس يستخدم الدالة ToByte فـ C# يستخدم الأمر المباشر:

```
  IL_000b:  unbox.any  [mscorlib]System.Byte
```

قمنا بالحصول على كود CIL من خلال برنامج IL Disassembler والذي يأتي مع بيئة الـ.NET. Visual Studio

يمكنك تحميل برنامج IL Disassembler من خلال الإنترنت من خلال موقع .Microsoft كما يمكنك أيضا تحميل برامج أخرى لإظهار كود CIL أو حتى الكود الأصلي VB .NET) أو (C# من خلال الإنترنت أيضا, والذي يعد برنامج XenoCode Fox Code Analyzerمن أشهر هذه البرامج وأقواها. وتسمى هذه البرامج Disassemblers أو Disassembliers وتقوم هذه البرامج على فك الـ Assemblyالتي تحددها (سواء EXE أوDLL) وتقوم بإرجاع الكود الخاص بها (CILأو VB .NET أو C# وغيرها).

مدة حياة العناصرObjects Lifetime

كما قلنا من قبل أن العناصر من نوع Value Type تخزن في الجزء الأول من الذاكرة وهو Stack ولذلك يتم إزالتها فور انتهاء قطعة الكود التي تحتويها.

مثال:

```
'VB .NET Code
     For i As Integer = 0 To 10
         Console.WriteLine(i)
     Next
     'هنا تنتهي
     'i
```

```
     // C# Code
     for (int i = 1; i <= 10; i++)
         Console.WriteLine(i);
     // هنا تنتهي
     // i
```

فبانتهاء قطعة الكود التي تحتوي العنصر (مثلاi) تنتهي حياة هذا العنصر بإنهائه وتحريره من الذاكرة.Stack

وماذا عنReference Types ؟

للذاكرة- Heap وهي التي تخزن فيها العناصر من نوع -Reference Type نظام آخر!

فلا تنتهي الحياة للعنصر بمجرد انتهاء القطعة التي تحتويه! ولكن بدلا من ذلك يتدخل وقت التشغيل Runtime لحل هذه المشكلة باستخدام واحدة من أقوى الأدوات -والمنسية غالبا من قِبل المبرمجين- وهي!!(GC (Garbage Collector

قاعدة :ليس عليك عند التعامل مع Heap سوى أن تقوم بإنشاء العنصر باستخدام الكلمة الدليلية New) new) في (#C فقط, ودع GC تتولى أمره.

فبظهور بيئة الدوت نت سهلت على كثير من المبرمجين القدماء التعامل مع ,Heap فحلت لهم مشكلة الاضطرار إلى إلغاء العنصر من الذاكرة يدويا.

ماذا يحدث عن إنشاءك لعنصر جديد باستخدام أمر الإنشاء New) new) في(#C ؟

عند إنشاءك لعنصر جديد يقوم Runtime بتنفيذ خطوات محددة لإنشاء هذا العنصر وهذه الخطوات هي كالتالي:

- حساب المساحة المطلوبة لهذا العنصر وهي العناصر الداخلية Internal Members والخارجية (وهي التي تم اكتسابها عن طريق التوارث). فيقوم CLR بحساب المساحة المطلوبة لهذا العنصر في الذاكرة.
- بعد ذلك يقوم :CLR (Common Language Runtime) وقت التشغيل) بحساب هل هناك مساحة كافية لهذا العنصر أم لا؟ فإن وجدت هذه المساحة ينتقل إلى الخطة الثالثة. فإن لم يجد مساحة كافية يقوم بإجراء عملية Garbage Collection لإزالة

العناصر التي تم انتهاء استخدامها من الذاكرة .Heap حيث أن CLR لا يقوم بإزالة العنصر الذي تم انتهاء استخدامه مطلقا من الذاكرة إلا عند إنهاءك للبرنامج بالكامل أو إذا امتلأت الذاكرة!

- يقوم بعد ذلك CLR بفحص الذاكرة Heap للتأكد من أن هناك مساحة كافية لهذا العنصر. فإن وجدت يتم إضافة هذا العنصر إلى الذاكرة وذلك بعد آخر عنصر تمت إضافته مباشرة. ثم يقوم CLR بإرجاع Reference لهذا العنصر كي يستطيع الكود الوصول إليه.

قاعدة :لا يقوم CLR بإزالة أي عنصر من نوع Reference Type من الذاكرة Heap مطلقا, إلا إذا امتلأت الذاكرة أو قام المستخدم بإنهاء برنامجك.

فعلى هذه القاعدة نأخذ قاعدة أخرى:

قاعدة :لا يمكنك مطلقا تحديد متى يقوم CLR بإنهاء العنصر من نوع Reference Type من الذاكرة أثناء وقت التشغيل, بعكس Value Typeفتستطيع تحديد متى يتم إنهاؤه.

ولا ننسى هذه القاعدة والتي تعتمد عليها القاعدتين السابقتين.

قاعدة :لا يقوم CLR بمحاولة اكتشاف المساحة المتبقية من الذاكرة إلا عند إنشاءك لعنصر جديد.

وهذا رسم تخيلي لطريقة عمل:Heap

حيث يقوم CLR كلما بإضافة عنصر جديد من بداية المكان الذي يشير إليه مؤشر المساحة, ويقوم CLR أيضا بتحريك هذا المؤشر كلما أضاف عنصر جديد.

Garbage Collection

كلما زاد حجم برنامجك كلما زاد عدد العناصر فيه التي تستخدم ,Heap وربما يجئ الوقت الذي تمتلئ فيه ذاكرة Heap بالعناصر التي تستخدمها والتي حتى لا تستخدمها.

فعند إنشاءك لعنصر جديد (وتسمى عملية ,(Instantiating an Object يقوم CLR بفحص الذاكرة للتأكد من أن هناك مساحة كافية لهذا العنصر المطلوب إضافته أم لا؟

فإن لم توجد مساحة كافية يقوم CLR بإجراء عملية Garbage Collection باستخدام الخدمة.Garbage Collector

لاحظ الفرق بين اسمي الخدمة Garbage Collector وبين العملية التي تقوم بها هذه الخدمة .Garbage Collection حاول أن لا يختلط عليك الاسمان.

إن قيل GC فقط, فاعلم بأنها Garbage Collector أي الخدمة نفسها وليست العملية.

قاعدة: إذا لم توجد مساحة كافية في الذاكرة Heap يقوم CLR تلقائيا بإجراء عملية.Garbage Collection

قاعدة: تختص GC بالتعامل مع الذاكرة .Heap ولذلك فكل العناصر المسجلة في هذه الذاكرة تكون من نوع Reference Type كما لنا سابقا.

تذكر أن أثناء شرح GC أو Heap جميع العناصر التي نتكلم عنها هي من نوع.Reference Type

تذكر أيضا أنه أثناء شرح GC إذا ذكرت "الذاكرة" فقط فهي ذاكرة.Heap

جذور البرنامجApplication Roots

ولكي يستطيع CLR معرفة هل هذا العنصر ما زال يستخدم من قِبل برنامجك أم لا, يقوم بالاعتماد على "جذور البرنامج Application Roots"أو كما تختصر إلى "الجذور."Roots

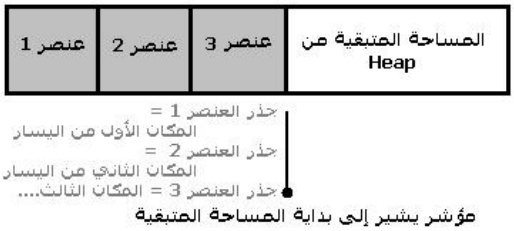
الجذر :Root هو عبارة عن مساحة مخصصة في الذاكرة Heap تحتوي على ارتباط Reference للعنصر الأصلي في الذاكرة. فإن وجد هذا الارتباط إذا فالعنصر ما زال يستخدم في برنامج. والعكس بالعكس.

ويمكنك فهم الجذر Root بطريقة أسهل على أنه المتغير Variable في برنامجك الذي يستخدم العنصر الموجود في الذاكرة ففي حالة وجود هذا المتغير إذا فالعنصر ما زال يستخدم. وإلا كما قلنا فالعكس بالعكس.

فيقوم CLR بالتأكد هل لهذا العنصر الموجود في الذاكرة جذور في برنامجك؟ فإن لم توجد له جذور يتم إضافة علامة Mark بجانبه كي يتم إنهاؤه. حتى إذا انتهى CLR من فحص جميع العناصر, يقوم CLR بإنهاء العناصر التي تم إضافة لها العلامة كي يتم إنهاؤها.

ثم يقوم CLR بعد ذلك بتنظيم الذاكرة وكذلك تنظيم الجذور لتتغير بتغيير مكان العنصر, وأيضا يقوم بوضع المؤشر على بداية المساحة المتبقية.

لاحظ الرسم التالي والذي يوضح الشكل التخيلي للذاكرة Heap قبل وبعد إزالة العنصر 2 منها.



تسمى ذاكرة Heap بـ Managed Heap وذلك بسبب أنك لا يمكنك التعامل مع الـ Heap بشكل مباشر, ولكن بدلا من ذلك تستطيع التعامل مع GC والتي تفرض عليك المعتقدات الخاصة بها. فإن Heap يتم إدارتها من خلال.GC

الخدمةGarbage Collector

GCأو Garbage Collector هي واحدة من الخدمات المهمة التي يقدمها لنا ,CLR وهي المسؤولة عن التعامل مع الذاكرة .Heap وليس للمبرمج أدنى تصرف مع هذه الذاكرة حيث أنه لا يمكنه التعامل معها إلا من خلال هذه الخدمة.

CLR: هو اختصار لـ ,Common Language Runtime أو كما نسميه وقت التنفيذ ,Runtime وهو مجموعة من الخدمات التي تحتاجها الـ Assembly الخاصة بك أثناء وقت التشغيل. ولا يمكن لأي .NET Assembly أن تعمل بدون.CLR

ومن هذا التعريف نأخذ تعريف آخر:

GC: وهي اختصار لـ ,Garbage Collector ويمكننا أن نطلق عليها "اسم مدير الذاكرة ,"Heap وهي خدمة من الخدمات المهمة التي يوفرها لنا ,CLR والتي تستخدم لإدارة الذاكرة.Heap

كما قنا سبقا, عندما يحاول CLR إضافة عنصر إلى الذاكرة Heap ولا يجد المساحة الكافية لهذا العنصر في الذاكرة, يقوم باستدعاء الخدمة ,Garbage Collector وهي المسؤولة عن تنفيذ عملية Garbage Collection وهي عملية مسح شامل للذاكرة لإزالة العناصر الغير مستخدمة.

Garbage Collection: هي العملية التي تقوم بها الخدمة ,Garbage Collector على الذاكرة. وهي عملية مسح شامل للذاكرة لإزالة العناصر الغير مستخدمة وهي التي ليس لها جذور Roots في برنامجك.

كلما زاد حجم برنامجك كلما زاد عدد العناصر المستخدمة فيه. وكلما زادت عدد العناصر زادت المساحة المستخدمة من الذاكرة!

ربما يستخدم برنامجك آلاف العناصر أثناء تشغيله.

فمما لمنا أنه ربما تكون هناك آلاف العناصر المسجلة في الذاكرة. ويظهر هنا سؤال:

هل تقوم GC أثناء عملية المسح بالتأكد من هذه العناصر جميعها من حيث استخدامها في برنامجك؟

والإجابة هي لا!!! فالتأكد من كل هذه العناصر سيستهلك الكثير والكثير من الزمن. ولذلك فـ GC وضعت قاعدة جديدة وهي Object Generation.

مجموعاتHeap

قاعدة :عند إضافتك لأي عنصر جديد في الذاكرة تقوم GC بنسبته إلى مجموعة (وهي مجموعة العناصر التي تم إضافتها مؤخرا) عن طريق رقم المجموعة وهو 0.

ففكرة Object Generation هي أن العناصر المسجلة في الذاكرة مقسمة إلى 3 أقسام وكل قسم له رقم Generation معين. وهذه الأقسام هي:

- Generation 0:
وهي مجموعة العناصر التي تم إضافتها مؤخرا ولم تحدث أي عملية مسح (Garbage Collection) منذ إضافتها.

- Generation 1:
وهي المجموعة الثانية والتي تمثل العناصر التي تمت إضافتها من فترة وحدثت عملية مسح واحدة فقط عليها.

- Generation 2:
وتمثل المجموعة الأخيرة وهي العناصر التي تمت إضافتها من فترة أطول وحدثت أكثر من عملية مسح عليها.

فلتفادي مرور الزمن أثناء إجراء عملية المسح على آلاف العناصر, تقوم GC أثناء عملية المسح (Garbage Collection) بالتالي:

- إجراء عملية المسح على العناصر التي تمت إضافتها مؤخرا والتي لم تجرى أي عملية مسح عليها قبل ذلك.
- بعد انتهاء عملية المسح تقوم GC بإزالة العناصر التي تم لم تعد تستخدم حتى الآن.
- بعد عملية الإزالة باقي العناصر التي كانت في المجموعة الأولى والتي تحمل الـ Generation 0 يتم تحويلهم إلى المجموعة الثانية وهي مجموعة العناصر التي مرت عليها عملية مسح واحدة والتي تحمل الـ Generation 1.
بعد هذه الخطوة يقوم CLR بحساب المساحة المتبقية من الذاكرة فإن لم توجد مساحة كافية لإضافة العنصر المراد إضافته ينتقل إلى الخطوة التالية. وإلا فإنه يتم إضافته.
- فإن لم توجد مساحة بعد عملية المسح على المجموعة الأولى تقوم GC بإجراء عملية بحث على المجموعة الثانية.
- بعد انتهاء عملية المسح على المجموعة الثانية تقوم GC بإزالة العناصر التي لم تعد تستخدم (طبعا باستخدام الجذور Roots).
- بعد عملية الإزالة تقوم GC بنقل العناصر من المجموعة الثانية إلى المجموعة الثالثة وهي المجموعة التي تحمل Generation 2والتي تحتوي على العناصر التي حدث لها أكثر من عملية مسح.
بعد هذه الخطوة يقوم CLR بحساب المساحة المتبقية من الذاكرة فإن لم توجد مساحة كافية لإضافة العنصر (حالة غير شائعة) المراد إضافته ينتقل إلى الخطوة التالية, وإلا فإنه يتم إضافته.
- فإن لم توجد مساحة بعد عملية المسح على المجموعة الثانية تقوم GC بإجراء عملية بحث على المجموعة الثالثة.
- بعد انتهاء عملية المسح على المجموعة الثالثة تقوم GC بإزالة العناصر التي لم تعد تستخدم.

وطبعا لأنه ليس هناك مجموعة أعلى من المجموعة الثالثة والتي تحمل الـ Generation 2, فتبقى العناصر المتبقية فيها بعد الإزالة كما هي.

فإن لم توجد مساحة متبقية بعد عملية المسح هذه (حالة نادرة جدا) يقوم CLR بإنشاء خطأ Exception وإظهاره للمستخدم.

قاعدة :لا يقوم GC أبدا بإزالة أي عنصر من الذاكرة أبدا إلا إذا لم توجد مساحة لإضافة العنصر الجديد المراد إضافته.

العنصرSystem.GC

بعض دوال وخواص العنصر System.GC والمسؤول عن الخدمةGarbage Collector :

- Collect:

هذه الدالة تقوم باستدعاء الخدمة GC لإجراء عملية المسح.(Garbage Collection)
ويمكنك إضافة رقم الـ Generation كمدخلات لتحديد المجموعة التي سوف يتم عمل المسح عليها.
وعندها يزيل GC العناصر الغير مستخدمة.

- CollectionCount:

تقوم هذه الدالة بأخذ رقم Generation كمدخلات وتقوم بإرجاع قيمة تحدد عدد المرات التي تم حدوث عملية المسح فيها على المجموعة المحددة.

- GetGeneration:

تقوم هذه الدالة بإرجاع رقم الـ Generation للعنصر المحدد.

- GetTotalMemory:

تقوم هذه الدالة بإرجاع عدد الكيلو بايتات التي تستخدم الآن من قِبل العناصر المسجلة في الذاكرة -حتى العناصر الغير مستخدمة في برنامجك ولم يتم إزالتها.-
تقوم هذه الدالة بأخذ قيمة توضح هل يتم عمل عملية مسح أولا لإزالة العناصر الغير مستخدمة قبل إجراء هذه الدالة؟

- KeepAlive:

تقوم هذه الدالة بأخذ عنصر كمدخلات وتقوم بجعل هذا العنصر موجود في الذاكرة حتى انتهاء هذه الـ Method التي تم إنشاؤه فيها حتى وإن تم انتهاء استخدامه.

- MaxGeneration:

وهي خاصية وليست دالة وهي مسؤولة عن إرجاع أعلى رقم Generation وهو الرقم 2 كما عـمنا.

- WaitForPendingFinalizers:

وهذه الدالة هي المسؤولة عن إيقاف برنامجك حتى انتهاء أي عملية تقوم بها الآن.GC
وأغلب استخدام هذه الدالة بعد دالة Collect لإيقاف عمل برنامجك حتى تنتهي عملية المسح والإزالة -إن وجدت.-

الاستعلام عنObject Generation

تأمل هذا المثال لتلاحظ نظام مجموعات GC (نقصدGC Generations):

```
'VB .NET Code
     Public Sub Main()
         Dim c1 As New HelloClass
         Dim c2 As New HelloClass

         Console.WriteLine("******Display Generations*****")
         Console.WriteLine("c1 = " & System.GC.GetGeneration(c1))
         Console.WriteLine("c2 = " & System.GC.GetGeneration(c2))

         GC.Collect()

         Console.WriteLine("******Display Generations*****")
         Console.WriteLine("c1 = " & System.GC.GetGeneration(c1))
         Console.WriteLine("c2 = " & System.GC.GetGeneration(c2))
```

```
        GC.Collect()

        Console.WriteLine("******Display Generations*****")
        Console.WriteLine("c1 = " & System.GC.GetGeneration(c1))
        Console.WriteLine("c2 = " & System.GC.GetGeneration(c2))

        GC.Collect()

        Console.WriteLine("******Display Generations*****")
        Console.WriteLine("c1 = " & System.GC.GetGeneration(c1))
        Console.WriteLine("c2 = " & System.GC.GetGeneration(c2))
    End Sub

    Class HelloClass
        Public Sub SayHello()
            Console.WriteLine("Hello World!")
        End Sub
    End Class
```

```
    // C# Code
    public static void Main()
    {
        HelloClass c1 = new HelloClass();
        HelloClass c2 = new HelloClass();

        Console.WriteLine("******Display Generations*****");
        Console.WriteLine("c1 = " + System.GC.GetGeneration(c1));
        Console.WriteLine("c2 = " + System.GC.GetGeneration(c2));

        GC.Collect();

        Console.WriteLine("******Display Generations*****");
        Console.WriteLine("c1 = " + System.GC.GetGeneration(c1));
        Console.WriteLine("c2 = " + System.GC.GetGeneration(c2));

        GC.Collect();

        Console.WriteLine("******Display Generations*****");
        Console.WriteLine("c1 = " + System.GC.GetGeneration(c1));
        Console.WriteLine("c2 = " + System.GC.GetGeneration(c2));

        GC.Collect();

        Console.WriteLine("******Display Generations*****");
        Console.WriteLine("c1 = " + System.GC.GetGeneration(c1));
        Console.WriteLine("c2 = " + System.GC.GetGeneration(c2));
    }

    class HelloClass
    {
        public void SayHello()
        {
            Console.WriteLine("Hello World!");
        }
    }
```

الخاتمة

تكلمنا في البداية عن أنواع البيانات في بيئة الدوت نت وقلنا أنها تنقسم إلى قسمين Value Types و Reference Types وكل قسم له قواعده. وقلنا أن Value Types يتم تسجيلها في الجزء الأول من الذاكرة وهو ,Stack بينما Reference Types تسجل في الجزء الآخر وهوHeap .

وتكلمنا أيضا عن Boxing و Unboxing وهما عمليتي التحويل من Value Type إلى Reference Type والعكس. وتكلمنا أيضا عن كيف تمثل هاتين العمليتين فيCIL .

وتكلمنا أيضا عن مدة حياة العناصر وكيف تنتهي العناصر من الذاكرة سواء Value Types أوReference Types .

وتكلمنا أيضا عن كيفية إدارة الذاكرة Heap باستخدام GC وهي واحدة من الخدمات العريقة فيCLR .

وفي النهاية تكلمنا عن Object Generations وكيف تحصل على الـ Generation للعنصر والتي يمكننا أن نسميها المجموعة التي ينتمي إليها العنصر في الذاكرة .