# NHỮNG ĐIỂM MỚI CỦA C# 4.0 & 5.0

## C# 4.0

### Dynamic binding

Dynamic binding là quá trình giải quyết các kiểu dữ liệu, thành viên và toán tử từ compile time sang runtime.

Kiểu dynamic được khai báo với từ khóa “dynamic”:

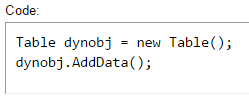
dynamic d = GetSomeObject();

d.Quack();

Một biến có kiểu dynamic sẽ thông báo cho compiler là không cần biên dịch cho đến quá trình runtime. Như vậy với ví dụ trên nếu phương thức Quack() chưa được khai báo mà lại được sử dụng thì compiler vẫn không báo lỗi nhưng trong quá trình runtime chương trình sẽ không chạy được vì không tìm thấy phương thức này.

#### Static binding và dynamic binding

Giả sử đối tượng **dynobj** có kiểu được khai báo như sau:



Trong trường hợp này, trình biên dịch sẽ tìm kiếm các phương thức AddData và sẽ ràng buộc nó tĩnh vào loại đối tượng dynobj.

###### Đầu tiên, trình biên dịch sẽ cố gắng tìm một phương thức có tên AddData mà không có tham số trong kiểu dữ liệu Table.

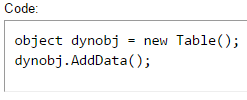
###### Sau đó, nó sẽ tìm thấy phương thức AddData trong các phương thức lấy tham số tùy chọn.

###### Nếu nó vẫn không thể tìm ra phương thức AddData, nó sẽ tìm ra phương thức trong lớp cơ sở của kiểu dữ liệu Table.

###### Nếu vẫn không tìm thấy, nó sẽ cố gắng để tìm kiếm các phương thức mở rộng kiểu Table là tham số.

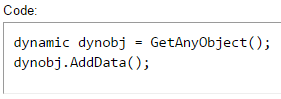
Nếu tất cả các điều kiện trên là sai, trình biên dịch sẽ tạo ra một lỗi mà nó không thể tìm ra phương thức AddData do đó các mã không thể được biên dịch. Đây là static binding.

Thực hiện thay kiểu Table sang object như code sau:



Trong trường hợp này, các giá trị được lưu trữ trong **dynobj** có thể chứa phương thức AddData, nhưng trình biên dịch không biết điều đó. Trình biên dịch chỉ có các thông tin về các loại đối tượng tức là biến trong trường hợp này. Do đó, nó sẽ cố gắng để tìm ra phương thức AddData trong object và tất nhiên là không tìm thấy nó và sẽ cho lỗi compile time.

Đến dynamic binding, tiếp tục với ví dụ trước:



Trên thực tế các dynamic object cũng không phải mô tả tương tự như các kiểu dữ liệu đối tượng đã nói ở trên. Sự khác biệt giữa một đối tượng không mô tả tĩnh và một đối tượng không mô tả động là nó cho phép biên dịch mã mà không giải quyết các kiểu dữ liệu trên loại biên dịch. Một dynamic object giải quyết các kiểu dữ liệu khi runtime chứ không phải là compile time.

Khi trình biên dịch thấy các biểu thức dynamic (có từ khóa dynamic), nó sẽ không giải quyết các kiểu dữ liệu và các thành viên và sẽ không ràng buộc bất cứ điều gì mà chỉ đơn giản gói các biểu thức này lại và sẽ chờ đến quá trình runtime mới ràng buộc.

#### Custom Binding

Custom binding là các loại ràng buộc xảy ra khi một đối tượng kế thừa interface IDynamicMetaObjectProvider. Bạn có thể tạo một đối tượng trong C# và kế thừa các interface IDUMP này để sử dụng dynamic binding. Tuy nhiên, phổ biến hơn, các đối tượng IDUMP được lấy từ các ngôn ngữ thực hiện tự động trong .NET như IronRuby và IronPython. Những ngôn ngữ sử dụng thời gian chạy ngôn ngữ năng động hoặc DLR cho các mục đích liên kết động. Các đối tượng trong các lớp này mặc nhiên kế thừa interface IDOMP; do đó dynamic binding xảy ra theo mặc định trong khi giải quyết các kiểu dữ liệu, các thành viên và phương thức. Ví dụ:

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Dynamic;

namespace CSharpTutorial

{

public class Table : DynamicObject

{

public override bool TryInvokeMember(InvokeMemberBinder binder, object[] args, out object result)

{

Console.WriteLine(binder.Name + "method was dynamically called");

result = null;

return true;

}

}

class Program

{

static void Main(string[] args)

{

dynamic dynobj = new Table();

dynobj.AddData();

dynobj.GetData();

Console.ReadLine();

}

}

}

Trước tiên là import namespace Dynamic, namespace này chứa lớp DynamicObject mà kế thừa interface IDynamicMetaObjectProvider. Chúng tôi tạo ra một lớp có tên Table và kế thừa nó từ lớp DynamicObject mà kế thừa IDMOP. Điều đó có nghĩa là tại lớp Table của chúng tôi cũng thực hiện các interface IDOMP có nghĩa là dynamic binding các đối tượng của lớp Table sẽ được tùy chỉnh ràng buộc (custom binding).

Bên trong lớp Table của chúng tôi, chúng tôi có ghi đè phương thức TryInvokeMember của lớp DynamicObject mà chúng tôi có được thừa kế. Bây giờ bất cứ khi nào, trình biên dịch tự động cố gắng để ràng buộc các đối tượng mà thực hiện IDMOP, phương thức này cũng sẽ được gọi. Các tham số ràng buộc của phương thức này thực sự có chứa các thông tin liên quan đến các phương thức đang được tự động bị ràng buộc.

Bây giờ đến các phương thức chính, ở đây chúng tôi đã khai báo đối tượng động dynobj kiểu Table. Sau đó gọi AddData và sau đó là GetData. Lớp Table không chứa hai phương thức này, nhưng bạn sẽ thấy rằng trình biên dịch vẫn sẽ biên dịch mã mà không có bất kỳ lỗi và sẽ trì hoãn các ràng buộc cho đến khi thời gian chạy. Điều này là do thực tế là chúng tôi đã thực hiện các đối tượng dynobj động. Nếu đối tượng dynobj không năng động, trình biên dịch sẽ tạo ra một lỗi về thời gian biên dịch và mã của chúng tôi sẽ không được biên dịch. Bây giờ, khi bạn gọi AddData và phương thức GetData trên đối tượng dynobj, phương thức TryInvokeMember của lớp Table sẽ được gọi hai lần và trong phương thức này, chúng tôi đã in tên của Invoker. Kết quả output như sau:



#### Language Binding

Language binding là một loại dynamic binding khác. Trong language binding, các đối tượng không kế thừa interface IDMOP. Thông thường language binding được sử dụng để tránh các vấn đề cố hữu trong tiềm ẩn .NET ràng buộc và các kịch bản nơi các kiểu dữ liệu được thiết kế không hoàn hảo. Một ví dụ điển hình của việc sử dụng language binding là với các kiểu số, vì kiểu số không chứa interface chung. Trong tình huống như vậy chúng ta có thể khai báo một phương thức động với tham số động và kiểu toán tử để thực hiện các phép tính số học trên thời gian chạy. Ví dụ:

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Dynamic;

namespace CSharpTutorial

{

class Program

{

static dynamic Average(dynamic num1, dynamic num2)

{

return ((num1 + num2) / 2);

}

static void Main(string[] args)

{

int num1 = 8;

int num2 = 4;

Console.WriteLine(Average(num1,num2));

Console.ReadLine();

}

}

}

Trong trường hợp này tất cả các loại resolution và phương thức ràng buộc sẽ bị trì hoãn cho đến thực hiện runtime. Kỹ thuật này là hữu ích, nhưng nó đi kèm với nhược điểm tiềm ẩn. Việc bảo đảm kiểm tra kiểu tĩnh là không có sẵn trong trường hợp này. Một nhược điểm của dynamic binding là nó đi kèm với hiệu suất đạt vì ngôn ngữ năng động, cơ chế thời gian chạy bộ nhớ đệm. Tuy nhiên, các lời gọi lặp đi lặp lại cùng một chức năng có thể được tối ưu hóa và số truy cập hiệu suất có thể được giảm nhẹ phần nào.

#### Bắt ngoại lệ Dynamic Binding (Dynamic Binding Exception Handling)

Trong phần trước, chúng ta sử dụng dynamic binding khi chúng ta muốn để ràng buộc các thành viên mà chúng ta biết nhưng trình biên dịch không biết, do đó không có ngoại lệ compile time xảy ra. Tuy nhiên, kịch bản có thể xảy ra khi các biến thành viên, chúng ta đang cố gắng để ràng buộc không có sẵn tại thời gian chạy là tốt. Trong trường hợp này là một ngoại lệ được ném ra, gọi là RuntimeBinderException. Hãy xem xét một ví dụ đơn giản của một kiểu số nguyên được xác định tự động. Nếu chúng ta gọi là một số phương thức mà không có sẵn trên kiểu số nguyên, trên một đối tượng tự động tạo ra mà là đề cập đến một kiểu số nguyên, RuntimeBinderException sẽ được ném ra. Khái niệm này được giải thích tốt nhất với sự giúp đỡ của một ví dụ:

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Dynamic;

namespace CSharpTutorial

{

class Program

{

static void Main(string[] args)

{

dynamic dynobj = 10;

dynobj.AddData();

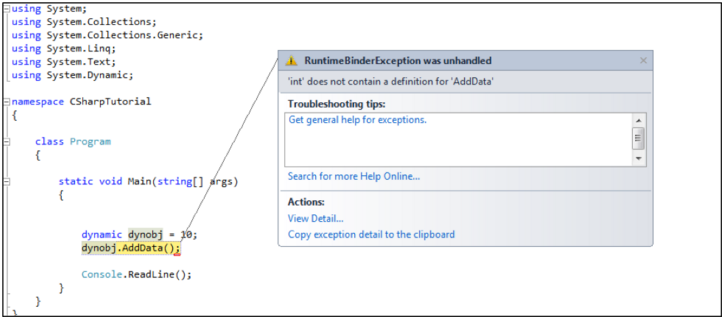
Console.ReadLine();

}

}

}

Chúng ta đã tạo ra một đối tượng dynobj động và đã được lưu trữ một biến số nguyên trong đối tượng động này. Khi chúng ta gọi phương thức AddData trên đối tượng này, trình biên dịch sẽ không tạo ra một lỗi bởi vì nó đã trì hoãn việc ràng buộc cho đến khi thời gian chạy. Khi chạy, trình biên dịch sẽ cố gắng liên kết phương thức AddObject với các đối tượng dynobj nhưng vì dynobj là đề cập đến một số nguyên là 10, và số nguyên không chứa bất kỳ phương thức AddData mà có thể được liên kết với nó trên thời gian chạy, do đó một RuntimeBinderException sẽ được ném ra. Bây giờ nếu bạn cố gắng để chạy mã trong VS 2010 trong chế độ gỡ lỗi nó sẽ có kết quả như sau.



Bạn có thể thực hiện các cơ chế xử lý ngoại lệ để xử lý các trường hợp ngoại lệ như vậy.

Kiểu dynamic và kiểu object là gần như giống hệt nhau chỉ ngoại trừ rằng loại động cho phép bạn để giải quyết các loại và các thành viên động. Nếu bạn gọi typeof trên một đối tượng động và gõ kết quả sẽ là như nhau. Khái niệm này cũng mở rộng sang mảng động và các kiểu tự xây dựng. Một sự giống nhau giữa một kiểu động và đối tượng là một kiểu động có thể trỏ đến đối tượng của bất kỳ loại giống như một tài liệu tham khảo đối tượng với ngoại lệ của kiểu con trỏ.

Kiểu động và các loại đối tượng có thể được chuyển đổi bên trong. Điều này sẽ được giải thích tốt nhất với sự giúp đỡ của ví dụ sau:

using System.Linq;

using System.Text;

using System.Dynamic;

namespace CSharpTutorial

{

class Program

{

static void Main(string[] args)

{

object obj = new System.Text.StringBuilder();

dynamic dynobj = obj;

dynobj.Append("go4Expert");

Console.WriteLine(obj);

Console.ReadLine();

}

}

}

Chúng ta đã chỉ đơn giản khai báo một obj mà có thể lưu trữ một đối tượng StringBuilder. Tiếp theo, chúng tôi đã khai báo một đối tượng dynobj kiểu động và lưu trữ các tham chiếu của obj trong đó. Sau đó gọi là phương thức Append của lớp StringBuilder để nối thêm dữ liệu vào chuỗi. Bởi vì bây giờ đối tượng động dynobj đang trỏ đến đối tượng obj; do đó thay đổi được thực hiện bởi đối tượng động sẽ được phản ánh trong thực tế đối tượng obj là tốt. Sau đó in giá trị của obj mà bây giờ là một chuỗi. Kết quả như sau:



### Tham số tùy chọn và đặt tên đối số

Thực ra đây không phải là một tính năng duy nhất mà là hai tính năng khác nhau. Bạn có thể nhận được lợi ích nhiều hơn nếu bạn làm việc với chúng. Tham số có tên là một cách để cung cấp một tham số cho phương thức sử dụng tên của các tham số tương ứng thay vì vị trí của nó trong danh sách tham số. Trong khi đó, tham số tùy chọn cho phép bạn bỏ qua các đối số để gọi thành viên.

Đầu tiên bắt đầu với các tham số tùy chọn. Giả sử bạn đang tạo ra một ứng dụng Máy tính, nơi bạn có để làm một số "Add" hoạt động bằng cách đi qua hai, ba hoặc bốn tham số của phương thức. Bạn sẽ làm gì nếu bạn đang sử dụng C# 2.0 hoặc 3.0? Bạn sẽ không có lựa chọn nào khác hơn là tạo ra nhiều phương pháp và đi qua các thông số trong này được gọi là phương thức overloading.

public int Add(int a, int b);

public int Add(int a, int b, int c);

public int Add(int a, int b, int c, int d);

Nếu bạn đang sử dụng C# 4.0, chỉ cần quên đi phương thức overload bằng cách sử dụng các tham số tùy chọn. Tham số tùy chọn là một giá trị mặc định trong phương thức. Sử dụng tham số tùy chọn như sau:

public int Add(int a, int b, int c = 0, int d = 0);

Ở đây, chúng ta đang truyền các giá trị mặc định cho các tham số "c" và "d". Do đó bạn có thể gọi phương pháp này theo yêu cầu:

Add(10, 20); *// 10 + 20 + 0 + 0*

Add(10, 20, 30); *// 10 + 20 + 30 + 0*

Add(10, 20, 30, 40); *// 10 + 20 + 30 + 40*

Trong trường hợp đầu tiên, nó sẽ bỏ qua '0' (không) ở tham số tùy chọn c và d, trong trường hợp thứ hai tham số thứ tư sẽ được xem như "0" (zero) và trong trường hợp thứ ba tất cả các thông số có sẵn với giá trị thích hợp.

Bây giờ, chúng ta hãy nghĩ rằng chúng ta có một tình huống mà chúng ta đang tạo một tài khoản của một người sử dụng với các phương thức có tên " CreateAccount" trong đó có một thông số không tùy chọn "Name" và hai tham số tùy chọn "Address" & "Age". Ở đây trong một số kịch bản, bạn muốn bỏ qua "Name" và "Age".

public void CreateAccount(string name, string address = "unknown", int age = 0);

Bạn đang nghĩ đến việc gọi phương thức với một chuỗi rỗng hoặc giá trị null cho tham số "Address "? Một tính năng mới của C# 4.0 được gọi là "Các tham số được đặt tên". Sử dụng tính năng này, bạn có thể gọi các phương pháp với tên riêng thích hợp hoặc thậm chí bạn có thể thay đổi vị trí của các tham số khi gọi phương thức.

CreateAccount("Kunal", age: 28);

CreateAccount(address: "India", name: "Kunal");

Ví dụ đầu tiên cho thấy cách gọi phương thức mà không có tham số giữa "Address" và ví dụ thứ hai chứng tỏ cách gọi phương thức tương tự sau khi thay đổi vị trí của các tham số trong danh sách tham số bằng cách sử dụng tên của tham số. Bằng cách này, bạn có thể tạo ra một phương thức duy nhất với việc đặt tên và tùy chọn tham số thay vì quá tải nó nhiều lần và sử dụng nó ở khắp mọi. Tính năng này giúp cho mã nguồn trong sáng hơn và lập trình viên tốn ít công sức hơn.

### Type variance with generic interfaces and delegates

Thuật ngữ phương sai (variance) đề cập đến khả năng sử dụng một kiểu nơi khác đã được chỉ định, và trong bối cảnh đó có 3 chú ý mà chúng ta đã từng biết:

Bất biến (invariant): Một tham số trở lại là bất biến, nếu chúng ta phải sử dụng kết hợp chính xác của tên kiểu dữ liệu giữa các kiểu thời gian chạy và kiểu được khai báo. Đối với các thông số bất biến, không phải hiệp biến và cũng không được phép là phản biến.

Hiệp biến (covariant): Một tham số là hiệp biến nếu chúng ta có thể sử dụng một loại có nguồn gốc như là một thay thế cho các loại tham số, và dụ một lớp dẫn xuất có thể được sử dụng khi một cá thể của lớp cha được mong đợi. Hiệp biến là việc chuyển đổi một loại từ cụ thể hơn để tổng quát hơn. Ví dụ, chuyển đổi một đối tượng của loại xe ô tô các loại.

Phản biến (Contravariance): là chính xác đối diện của hiệp biến, tức là nó là sự chuyển đổi của một loại từ tổng quát hơn để cụ thể hơn. Một giá trị trả về là phản biến nếu chúng ta có thể gán kiểu trả về cho một biến của một loại ít xuất phát so với tham số. Một ví dụ lớp cơ sở có thể được sử dụng khi một trường lớp đã được dự kiến.

Đúng là một tài sản của nhà khai thác mà hành động trên các loại. Đó là khái niệm để chỉ định trong và ngoài các thông số trên các loại chung chung và cho phép tập nơi đó là an toàn. Các tham số kiểu biến thể có thể được khai báo cho các giao diện và các loại đại biểu. Thông số chung trong giao diện là bất biến theo mặc định, vì vậy chúng ta cần phải xác định một cách rõ ràng cho dù chúng ta cần một tham số chung cụ thể được hiệp biến hoặc phản biến. Ví dụ dưới đây cho thấy cả hai đồng phương sai và hỗ trợ phản biến trong C# 4.0:

class Fruit { }  
  
class Apple : Fruit { }  
  
class Program  
{  
delegate T Func<out T>();  
delegate void Action<in T>(T a);  
  
static void Main(string[] args)  
  
    {  
    // Covariance  
    Func<Apple> apple = () => new Apple();  
    Func<Fruit> fruit = apple;  
    // Contravariance  
    Action<Fruit> fru1 = (fru) =>  
    { Console.WriteLine(fru); };  
    Action<Apple> app1 = fru1;  
    }  
}

Đối số hiệp biến sẽ chỉ được sử dụng ở các vị trí output: các giá trị phương thức hoàn trả, được chỉ thuộc tính và chỉ mục, và các đối số phản biến sẽ chỉ xảy ra ở các vị trí input: các thông số phương pháp, thiết lập chỉ tính hoặc chỉ mục.

Ngoài ra, tính năng phương sai chung cũng cho phép sự phân công của các loại đối tượng IEnumerable <string> để một biến với kiểu IEnumerable <object>.

### Cải thiện khả năng trao đổi COM

#### Omitting ref

Bởi vì một mô hình lập trình khác nhau, nhiều API COM chứa rất nhiều thông số tham chiếu. Những thông số này thường không có nghĩa là để biến đổi một đối số được truyền vào, nhưng chỉ đơn giản là một cách khác để truyền tham số giá trị.

Cụ thể đối với các phương thức COM, trình biên dịch cho phép khai báo các phương thức gọi đi qua các đối số theo giá trị và sẽ tự động tạo ra các biến tạm thời cần thiết để giữ các giá trị để vượt qua chúng bằng cách tham chiếu và sẽ loại bỏ các giá trị của chúng sau khi trở về cuộc gọi. Từ điểm nhìn của các lập trình viên, các đối số được truyền theo giá trị.

Phương thức dưới đây:

object fileName = "Test.docx";

object missing = Missing.Value;

document.SaveAs(ref fileName,

ref missing, ref missing, ref missing,

ref missing, ref missing, ref missing,

ref missing, ref missing, ref missing,

ref missing, ref missing, ref missing,

ref missing, ref missing, ref missing);

Viết lại phương thức trên:

document.SaveAs("Test.docx",

Missing.Value, Missing.Value, Missing.Value,

Missing.Value, Missing.Value, Missing.Value,

Missing.Value, Missing.Value, Missing.Value,

Missing.Value, Missing.Value, Missing.Value,

Missing.Value, Missing.Value, Missing.Value);

Và bởi vì tất cả các thông số được nhận giá trị Missing.Value có giá trị đó là giá trị mặc định của nó, việc kê khai của các lời gọi phương thức thậm chí có thể được giảm xuống:

document.SaveAs("Test.docx");

#### Dynamic Import

Nhiều phương thức COM chấp nhận và trả lại các loại biến thay đổi, được thể hiện trong các PIA như đối tượng. Trong hầu hết các trường hợp, một lập trình viên kêu gọi các phương thức đã biết kiểu tĩnh của một đối tượng trở thành bối cảnh của lời gọi, nhưng phải thực hiện một cách rõ ràng một hình thức trên các giá trị trở lại sử dụng các kiến thức đó. Những hình thức rất phổ biến mà họ tạo thành một mối phiền toái lớn.

Để làm dễ dàng hơn cho các lập trình viên, bây giờ có thể import các COM API trong một cách mà các biến thể được thay vì được đại diện bằng cách sử dụng kiểu dữ liệu động đó có nghĩa là COM chữ ký bây giờ có lần xuất hiện của kiểu động, thay vì đối tượng.

Điều này có nghĩa rằng các thành viên của một đối tượng trả về bây giờ có thể dễ dàng truy cập hay gán vào một biến mà không cần phải chuyển kiểu.

Thay vì dùng code này:

((Excel.Range)(excel.Cells[1, 1])).Value2 = "Hello World!";

Thì ta có thể dùng:

excel.Cells[1, 1] = "Hello World!";

Thay vì dùng code này:

Excel.Range range = (Excel.Range)(excel.Cells[1, 1]);

Thì ta có thể thay bằng:

Excel.Range range = excel.Cells[1, 1];

#### Các thuộc tính chỉ mục và mặc định

Một vài tính năng giao diện COM vẫn không có sẵn trong C#. Trên đầu danh sách là thuộc tính được lập chỉ mục và các thuộc tính mặc định. Như đã đề cập ở trên, sẽ là tốt nếu các giao diện COM được truy cập tự động, nhưng sẽ không được nhận ra nếu đó là kiểu dữ liệu tĩnh.

#### Kiểu dữ liệu tương đương và kiểu dữ liệu nhúng

Đối với việc xác định các mã tổ hợp với PrimaryInteropAssemblyAttribute, trình biên dịch sẽ tạo ra các kiểu dữ liệu tương đương (interface, struct, enumeration và delegate) và nhúng chúng trong mã assembly.

Để giảm kích thước cuối cùng của mã assembly, chỉ có các loại đã qua sử dụng và các thành viên của họ sử dụng sẽ được tạo ra và nhúng.

Mặc dù điều này làm cho sự phát triển và triển khai các ứng dụng bằng cách sử dụng các thành phần COM dễ dàng hơn bởi vì không có nhu cầu triển khai các PIAs, phát triển thành phần COM vẫn phải xây dựng các PIAs.

## C# 5.0

## Tài liệu tham khảo

Understanding Dynamic Binding in C#

<http://www.go4expert.com/articles/understanding-dynamic-binding-c-sharp-t30004/>

C# 4.0 New Features – Named and Optional Parameters

<http://www.codeproject.com/Articles/67146/C-New-Features-Named-and-Optional-Parameters>