Chương 1: Ôn tập các lớp cơ sở

Tổng quan

Trong chương này ta sẽ có cái nhìn rõ hơn về các **lớp cơ sở (base classes)** và cách mà chúng tương tác với ngôn ngữ C# để hỗ trợ cho ta trong việc viết mã.Cụ thể ta sẽ xem xét các chủ đề sau:

- Chuỗi và biểu thức chính quy (regular expression)
- Nhóm đối tượng ,bao gồm các danh sách mảng, collections và từ điển

Ta cũng xem xét System.Object, lớp mà mọi thứ đều được dẫn xuất từ nó.

1.1 System.object

System.object là lớp cơ sở chung mà mọi đối tượng khác được thừa kế và ta cũng xem xét về các phương thức thành viên chính của nó. Trong chương này ta sẽ tìm hiểu các phương thức còn lại của system.object. Đầu tiên ta sẽ tìm hiểu tóm tắt của từng phương thức:

Phương thức	**Truy xuất**	**Mục đích**
string ToString()	public virtual	Trả về một chuỗi đại diện cho đối tượng
int GetHashCode()	public virtual	Trả về mã băm của đối tượng được thiết kế cho phép ta tìm kiếm một cách hiệu quả các thể hiện của đối tượng trong từ điền
bool Equals(object obj)	public virtual	So sánh đối tượng này với một đối tượng khác
bool Equals(object objA, object objB)	public static	So sánh 2 đối tượng

bool ReferenceEquals(object objA, object objB)	public static	So sánh các tham chiếu đối tượng để xem chúng có chỉ đến cùng đối tượng
Type GetType()	public	Trả về một đối tượng dẫn xuất từ System.Type mà đưa ra chi tiết kiểu dữ liệu
object MemberwiseClone()	protected	
void Finalize()	protected virtual	Hàm hủy (Destructor)

1.2 Xử lý chuỗi (System.string)

Phương thức	**Mục đích**
Compare	So sánh nội dung của 2 chuỗi
CompareOrdinal	Giống compare nhưng không kể đến ngôn ngữ bản địa hoặc văn hoá
Format	Định dạng một chuỗi chứa một giá trị khác và chỉ định cách mỗi giá trị nên được định dạng.
IndexOf	Vị trí xuất hiện đầu tiên của một chuỗi con hoặc kí tự trong chuỗi.
IndexOfAny	Vị trí xuất hiện đầu tiên của bất kì một hoặc một tập kí tự trong chuỗi.
LastIndexOf	Giống indexof, nhưng tìm lần xuất hiện cuối cùng.
LastIndexOfAny	Giống indexofAny, nhưng tìm lần xuất hiện cuối cùng.

PadLeft	Canh phải chuỗi điền chuỗi bằng cách thêm một kí tự được chỉ định lặp lại vào đầu chuỗi.
PadRigth	Canh trái chuỗi điền chuỗi bằng cách thêm một kí tự được chỉ định lặp lại vào cuối chuỗi.
Replace	Thay thế kí tự hay chuỗi con trong chuỗi với một kí tự hoặc chuỗi con khác.
Split	Chia chuỗi thành 2 mảng chuỗi con ,ngắt bởi sự xuất hiện của một kí tự nào đó.
Substring	Trả về chuỗi con bắt đầu ở một vị trí chỉ định trong chuỗi.
ToLower	Chuyển chuỗi thành chữ thường.
ToUpper	Chuyển chuỗi thành chữ in.
Trim	Bỏ khoảng trắng ở đầu và cuối chuỗi .

1.2.1 Định dạng Chuỗi

Nếu ta muốn những lớp mà ta viết thân thiện với người sử dụng , thì chúng cần để trình bày chuỗi theo bất cứ cách nào mà người sử dụng muốn dùng. Thời gian chạy .NET định nghĩa một cách chuẩn để làm: dùng một interface hoặc IFormatable

Ví dụ:

```
double d = 13.45;
int i = 45;
Console.WriteLine("The double is {0,10:E} and the int contains {1}", d, i);
```

Chuỗi định dạng tự nó bao gồm hầu hết văn bản được trình bày, nhưng bất cứ ở đâu có biến được định dạng, chỉ mục của nó trong danh sách thông số trong dấu ngoặc. Có thể là thông tin khác bên trong dấu ngoặc về việc định dạng của mục đó.

Số kí tự được giữ bởi sự trình bày của mục có thể xuất hiện, thông tin này sẽ có dấu phảy đứng trước. Một số âm chỉ định rằng mục đó được canh trái, trong khi một số dương chỉ định mục đó được canh phải. Nếu mục đó giữ nhiều kí tự hơn được yêu cầu, nó vẫn xuất hiện đầy đủ.

Một chỉ định định dạng cũng có thể xuất hiện. Điều này sẽ được đặt trước bởi dấu hai chấm và chỉ định cách ta muốn mục được định dạng. Ví dụ ta muốn định dạng số như kiểu tiền tệ hoặc trình bày theo ký hiệu khoa học?

Đặc tả	**Áp dụng đến**	**Ý nghĩa**	**Ví dụ**
C	numeric types	locale-specific monetary value	$4834.50 (USA)£4834.50 (UK)
D	integer types only	general integer	4834
E	numeric types	scientific notation	4.834E+003
F	numeric types	fixed point decimal	4384.50
G	numeric types	general number	4384.5
N	numeric types	usual locale specific format for numbers	4,384.50 (UK/USA)4 384,50 (continental Europe)
P	numeric types	Percentage notation	432,000.00%
X	integer types only	hexadecimal format	1120 (NB. If you want to display 0x1120, you'd need to write out the 0x separately)

1.3 Biểu thức chính quy (Regular Expression)

1.3.1 Giới thiệu:

Ngôn ngữ biểu thức chính quy là ngôn ngữ được thiết kế đặc biệt cho việc xử lí chuỗi. Chứa đựng 2 đặc tính:

- Một tập mã escape cho việc xác định kiểu của các kí tự. Ta quen với việc dùng kí tự * để trình bày chuỗi con bất kì trong biểu thức DOS. Biểu thức chính quy dùng nhiều chuỗi như thế để trình bày các mục như là 'bất kì một kí tự' ,'một từ ngắt ','một kí tự tuỳ chọn',...

- Một hệ thống cho việc nhóm những phần chuỗi con, và trả về kết quả trong suốt thao tác tìm. Dùng biểu thức chính quy , có thể biểu diễn những thao tác ở cấp cao và phức tạp trên chuỗi.

ví dụ :

- Xác định tất cả các từ lặp lại trong chuỗi , chuyển ' "The computer books books" thành "The computer books"

- Chuyển tất cả các từ theo title case, như là chuyển "this is a Title" thàh "This Is A Title".

- Chuyển những từ dài hơn 3 kí tự thành title case , ví dụ chuyển "this is a Title" to "This is a Title"

- Bảo đảm các câu được viết hoa

- Phân cách những phần tử của URL

Mặc dù có thể sử dụng các phương thức System.String và System.Text.StringBuilder để làm các việc trên nhưng nếu dùng biểu thức chính quy thì mã có thể được giảm xuống còn vài dòng. Ta khởi tạo một đối tượng System.Text.RegularExpressions.RegEx, truyền vào nó chuỗi được xử lí, và một biểu thức chính quy (Một chuỗi chứa đựng các lệnh trong ngôn ngữ biểu thức chính quy).

Một chuỗi biểu thức chính quy nhìn giống một chuỗi bình thường nhưng có thêm một số chuỗi hoặc kí tự khác làm cho nó có ý nghĩa đặc biệt hơn. Ví dụ chuỗi \b chỉ định việc bắt đầu hay kết thúc một từ, vì thế nếu ta muốn chỉ định tìm kí tự th bắt đầu một từ, ta có thể tìm theo biểu thức chính quy, \bth. Nếu muốn tìm tất cả sự xuất hiện của th ở cuối từ ta viết th\b. tuy nhiên, biểu thức chính quy có thể phức tạp hơn thế, ví dụ điều kiện để lưu trữ phần kí tự mà tìm thấy bởi thao tác tìm kiếm.

Một ví dụ khác giả sử như ta muốn chuyển một số diện thoại UK từ trong nước sang định dạng quốc tế. Trong UK, định dạng ví dụ như là 01233 345532 hoặc (01233 345532) mà theo quốc tế sẽ là +44 12330345532, nói cách khác số 0 đầu sẽ được thay bằng +44 và các dấu ngặc phải được bỏ.

Thao tác này không quá phức tạp, nhưng cũng hơi rắc rối nếu ta dùng lớp chuỗi để làm (nghĩa là dùng các phương thức trong lớp chuỗi). Ngôn ngữ biểu thức chính quy sẽ cho phép ta xây dựng một chuỗi ngắn mà sẽ được phiên dịch để đạt được yêu cầu trên.

Ta xem đoạn văn bản này là chuỗi input. Giả sử ta muốn tìm tất cả các lần xuất hiện của ion. ta sẽ viết như sau:

```
string Pattern = "ion";
MatchCollection Matches = Regex.Matches(Text, Pattern,
RegexOptions.IgnoreCase |
RegexOptions.ExplicitCapture);
foreach (Match NextMatch in Matches)
{
Console.WriteLine(NextMatch.Index);
}
```

Trong ví dụ này ta dùng phương thức tĩnh Matches() của lớp Regex trong namespace System.Text.RegularExpressions. Phương thức này có thông số là text, pattern, và tập cờ từ cấu trúc liệt kê RegexOptions. Trong trường hợp này ta chỉ định tìm

kiếm không phân biệt chữ hoa - thường. Và cờ ExplicitCapture, cập nhật cách mà match được thu thập.

Ta sẽ thấy tại sao hàm Matches() trả về một tham chiếu đến đối tượng MatchCollection. Match là một thuật ngữ kĩ thuật cho những kết quả của việc tìm một thể hiện của pattern trong biểu thức. Được trình bày bởi lớp System.Text.RegularExpressions.Match. Do đó ta sẽ trả về một MatchCollection chứa tất cả các match, mỗi cái được trình bày bởi một đối tượng Match. Trong đoạn mã trên, ta đơn giản lặp trên tập thu được và dùng thuộc tính index của lớp Match, mà trả về chỉ mục trong đoạn input nơi mà match được tìm thấy. Khi chạy nó sẽ tìm ra 4 match.

	Ý nghĩa	**Ví dụ**	**Examples that this will match**
^	Bắt đầu của chuỗi nhập	^B	B, nhưng chỉ nếu kí tự đầu tiên trong chuỗi
$	Kết thúc của chuỗi nhập	X$	X, nhưng chỉ nếu kí tự cuối cùng trong chuỗi
.	Bất kì kí tự nào ngoại trừ kí tự xuống dòng(\n)	i.ation	isation, ization
*	Kí tự trước có thể được lặp lại 0 hoặc nhiều lần	ra*t	rt, rat, raat, raaat, and so on
+	Kí tự trước có thể được lặp lại một hoặc nhiều lần	ra+t	rat, raat, raaat and so on, (but not rt)
?	Kí tự trước có thể được lặp lại 0 hoặc 1 lần	ra?t	rt and rat only
\s	Bất kì kí tự khoảng trắng	\sa	[space]a, \ta, \na (\t and \n có ý nghĩa giống như trong C#)
\S	Bất kì kí tự nào không phải là khoảng trắng	\SF	aF, rF, cF, but not \tf
\b	Từ biên	ion\b	any word ending in ion

\B	bất kì vị trí nào không phải là từ biên	\BX\B	bất kì kí tự X ở giữa của một từ

1.3.2 Trình bày kết quả

Trong phần này ta sẽ xét ví dụ RegularExpressionsPlayaround. Để ta thiết lập một vài biểu thức chính quy và trình bày kết quả để thấy cách mà biểu thức chính quy làm việc.

Tâm điểm là phương thức WriteMatches(), mà trình bày tất cả các match từ MatchCollection theo định dạng chi tiết hơn. Trong mỗi match, nó trình bày chỉ mục nơi mà match được tìm thấy trong chuỗi nhập, chuỗi của match bao gồm match cộng thêm 19 kí tự bao quanh nó trong chuỗi nhập - 5 kí tự đứng trước và 5 kí tự đứng sau. (Nhỏ hơn 5 kí tự nếu match xuất hiện trong 5 kí tự của phần đầu và kết thúc của đoạn nhập.) Ví dụ match trên từ messaging mà xuất hiện gần cuối của chuỗi nhập được đánh dấu sẽ trình bày "and messaging of d" (5 kí tự trước và sau match)nhưng một match trên từ cuối data sẽ trình bày "g of data. "(chỉ một kí tự sau match). Bởi vì sao đó là cuối chuỗi. Một chuỗi dài hơn để ta thấy rõ nơi biểu thức chính quy định vị match:

```
static void WriteMatches(string text, MatchCollection matches)
{
  Console.WriteLine("Original text was: \n\n" + text + "\n");
  Console.WriteLine("No. of matches: " + matches.Count);
  foreach (Match nextMatch in matches)
  {
    int Index = nextMatch.Index;
    string result = nextMatch.ToString();
    int charsBefore = (Index < 5) ? Index : 5;
    int fromEnd = text.Length - Index - result.Length;
    int charsAfter = (fromEnd < 5) ? fromEnd : 5;
    int charsToDisplay = charsBefore + charsAfter + result.Length;
    Console.WriteLine("Index: {0}, \tString: {1}, \t{2}",
      Index, result,
```

```
        text.Substring(Index - charsBefore, charsToDisplay));
    }
}
```

Phần lớn của quy trình trong phương thức này minh hoạ số kí tự đượctrình bày trong chuỗi con dài hơn mà nó có thể trình bày không quan tâm đến đầu hay cuối chuỗi. Lưu ý ta sử dụng một thuộc tính khác của đối tượng Match, Value, chứa chuỗi xác định trong Match.RegularExpressionsPlayaround chứa một số phương thức với tên như là Find1, Find2... mà biểu diễn việc tìm kiếm dựa trên ví dụ trong phần này. Ví dụ find2 tìm bất kì chuỗi chứa n vào lúc đầu của một từ :

```
static void Find2()
{
string text = @"XML has made a major impact in almost every aspect of
software development. Designed as an open, extensible, self-describing
language, it has become the standard for data and document delivery on
the web. The panoply of XML-related technologies continues to develop
at breakneck speed, to enable validation, navigation, transformation,
linking, querying, description, and messaging of data.";
string pattern = @"\bn";
MatchCollection matches = Regex.Matches(text, pattern,
RegexOptions.IgnoreCase);
WriteMatches(text, matches);
}
```

Cùng với phương thức này là một phương thức main() mà ta có thể chỉnh sửa đề chọn một trong những phương thức Find<n>( ):

```
static void Main()
{
Find1();
```

```
Console.ReadLine();
}
```

1.3.3 Matches, Groups, and Captures:

Một đặc tính hay nữa của biểu thức chính quy là ta có thể nhóm những kí tự cùng nhau. Nó làm việc theo cùng cách như lệnh hợp trong C#. Trong Pattern biểu thức chính quy, ta có thể nhóm bất kì kí tự (bao gồm metacharacters và chuỗi escape) với nhau, và kết quả xem như là một kí tự đơn. Chỉ khác là ta dùng ngoặc đơn thay cho ngoặc vuông. Chuỗi kết quả gọi là **group.**

Ví dụ pattern (an)+ sẽ định vị bất kì chuỗi an. Quatifier + áp dụng chỉ cho kí tự trước nó. Nhưng bởi vì ta đã nhóm chúng lại nên việc áp dụng sẽ cho cả an như là một kí tự thống nhất. Ví dụ ta dùng (an)+ trong chuỗi nhập "bananas came to Europe late in the annals of history", sẽ cho anan từ bananas, nếu chỉ viết an+ thì sẽ có ann từ annals, cũng như 2 chuỗi tách biệt an từ bananas. Biểu thức (an)+ sẽ bắt sự xuất hiện của an, anan, ananan,... Trong khi biểu thức an+ sẽ bắt sự xuất hiện của an, ann, annn,... ta có thể thắc mắc là (an)+ sẽ cho anan từ bananas chứ không phải là an, bởi vì theo luật thì nếu có 2 trường hợp có khả năng (ở đây là an và anan) thì mặc định match sẽ chứa khả năng dài nhất có thể.

ví dụ khác : ta có URL có định dạng sau:

<protocol>://<address>:<port>

port là tuỳ chọn. Ví dụ của URL là **http://www.wrox.com:4355**. giả sử ta muốn lấy protocol, address, port từ URL trên. Ta biết là có thể có khoảng trắng hoặc không có (nhưng không có dấu chấm). Ta có thể dùng biểu thức sau: \b(\S+)://(\S+)(?::(\S+))?\b

Đây là cách biểu thức làm việc. Đầu tiên là phần đầu và đuôi là chỗi \b bảo đảm rằng chúng ta chỉ quan tâm đến phần kí tự mà là từ nguyên vẹn. Trong đó, nhóm đầu tiên là (\S+):// sẽ lấy một hoặc nhiều kí tự mà không đếm khoảng trắng, mà theo sau bởi :// điều này sẽ lấy http:// vào phần đầu của HTTP URL. Chuỗi con (\S+) sẽ lấy phần như là

www.wrox.com của URL trên. Nhóm này cũng kết thúc khi nó gặp phần cuối của từ (\b) hoặc nó gặp dấu hai chấm (:) đánh dấu phần kế tiếp.

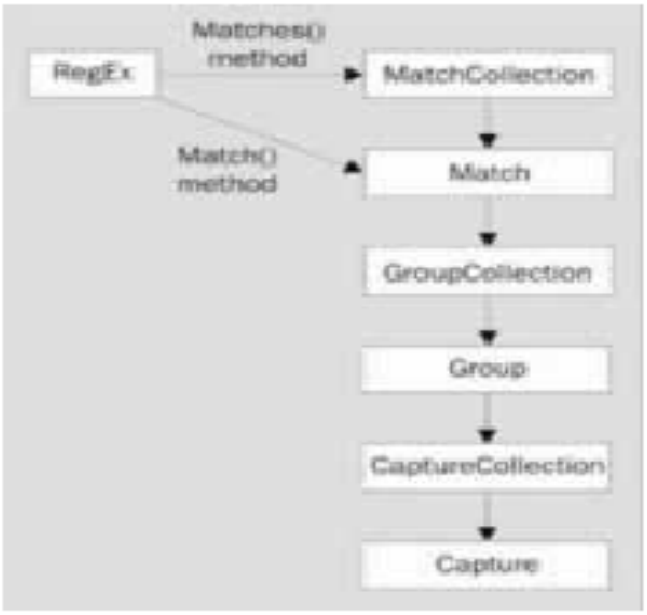
Phần kế tiếp được lấy là port. Dấu ? chỉ định nhóm này là tuỳ chọn trong match Quan trọng là số port không phải luôn được đặc tả trong URL - có lúc nó không có mặt. Ta muốn chỉ định dấu hai chấm có thể xuất hiện hoặc không, nhưng ta không muốn lưu trữ dấu hai chấm trong nhóm. Ta làm điều này bằng cách tạo 2 group lồng nhau. Cái bên trong (\S+) sẽ lấy những thứ sau dấu hai chấm (ví dụ ở đây là 4355) nhóm ngoài chứa đựng nhóm trong đứng trước dấu hai chấm mà được đứng trước chuỗi ?: , chuỗi này chỉ định rằng nhóm trong câu hỏi không được lưu (ta chỉ muốn lưu 4355; không lưu :4355) đừng nhầm lẫn bởi 2 dấu hai chấm - dấu đầu tiên là của phần chuỗi ?: nói rằng ' không lưu nhóm này' , và cái thứ hai là kí tự được tìm kiếm. Nếu ta chạy pattern này trên chuỗi :

Hey I've just found this amazing URI at http:// what was it - oh yes http://www.wrox.com

Ta sẽ lấy một match http://www.wrox.com. Trong match này có 3 nhóm được đề cập do đó có thể mỗi nhóm có thể không lấy gì, một hoặc nhiều hơn một nhóm. Mỗi match riêng này được biết đến như là **capture**. Vì thế, nhóm đầu tiên , (\s+), có một capture, http. Nhóm thứ hai cũng có một capture, www.wrox.com nhưng nhóm thứ ba không có capture, bởi vì không có số port trong URL này.

Lưu ý chuỗi chứa một nửa http://. Mặc dù điều này không phù hợp với nhóm đầu tiên của ta. Nó sẽ không được lấy ra qua tìm kiếm bởi vì biểu thức tìm kiếm đầy đủ sẽ không phủ hợp vời phần kí tự này.

Ta không phải biểu diễn bất kì ví dụ của C# mà dùng Groups và captures. Nhưng ta sẽ đề cập những lớp .NET RegularExpressions hổ trợ groups và captures, là những lớp Group và Capture. Cũng có những lớp GroupCollection và CaptureCollection, mà trình bày việc thu thập groups và captures. Lớp Match chứa phương thức Group(). Mà trả về một đối tượng GroupCollection. Lớp Group thi hành một phương thức Captures() mà trả về một CaptureCollection. Mối quan hệ giữa những đối tượng được thể hiện qua biểu đồ sau :



Việc trả về một đối tượng Group mỗi lần ta muốn nhóm một số kí tự cùng với nhau có thể không phải là những gì ta muốn làm. Có một số overhead liên quan đến việc khởi tạo đối tượng, mà bị lãng phí nếu tất cả những gì ta muốn là nhóm một vài kí tự cùng nhau như là một phần pattern. Ta có thể không cho phép điều này bằng việc bắt đầu nhóm với chuỗi kí tự ?: cho mỗi nhóm riêng, khi ta làm trong ví dụ URL, hoặc cho tất cả những nhóm bằng việc chỉ định cờ RegExOptions.ExplicitCaptures trên phương thức REgEX.Matches() như ta đã làm trong các ví dụ trước.

1.4 Nhóm các đối tượng

Chúng ta đã khảo sát một số lớp cơ sở của .NET có cấu trúc dữ liệu trong đó một số đối tượng được nhóm với nhau. Cấu trúc đơn giản mà ta đã học là mảng, đây là một thể hiện của lớp System.Array. Mảng có lợi điểm là ta có thể truy nhập từng phần tử thông qua chỉ mục. Tuy nhiên khuyết điểm của nó là ta phải khởi tạo kích thước của nó. Không thể thêm, chèn hoặc bỏ một phần tử sau đó. Và phải có một chỉ mục số để truy nhập vào một phần tử. Điều này không tiện lắm ví dụ như khi ta làm việc với một bản ghi nhân viên và muốn tìm bản ghi theo tên nhân viên.

.NET có một số cấu trúc dữ liệu khác hổ trợ cho công việc này. Ngoài ra còn có một số inteface, mà các lớp có thể khai báo chúng hổ trợ tất cả chức năng của một kiểu cụ thể cấu trúc dữ liệu. Chúng ta sẽ xem xét 3 cấu trúc sau :

- **Array lists**
- **Collection**
- **Dictionary (hay maps)**

Các lớp cấu trúc dữ liệu này nằm trong namespace System.Collection

1.4.1 Array lists

Arraylist giống như mảng, ngoại trừ nó có khả năng phát triển. Được đại diện bởi lớp System.Collection.Arraylist.

Lớp Arraylist cũng có một một vài điểm tương tự với lớp StringBuilder mà ta tìm hiểu trước đây. Như StringBuilder cấp phát đủ chỗ trống trong vùng nhớ để lưu trữ một số kí tự, và cho phép ta thao tác các kí tự trong chỗ trống đó, the Arraylist cấp đủ vùng nhớ để lưu trữ một số các tham chiếu đối tượng. Ta có thể thao tác trên những tham chiếu đối tượng này. Nếu ta thử thêm một đối tượng đến Arraylist hơn dung lượng cho phép của nó, thì nó sẽ tự động tăng dung lượng bằng cách cấp phát thêm vùng nhớ mới lớn đủ để giữ gấp hai lần số phần tử của dung lượng hiện thời.

Ta có thể khởi tạo một danh sách bằng cách chỉ định dung lượng ta muốn. Ví dụ, ta tạo ra một danh sách Vectors:

```
ArrayList vectors = new ArrayList(20);
```

Nếu ta không chỉ định kích cỡ ban đầu , mặc định sẽ là 16:

```
ArrayList vectors = new ArrayList(); // kích cỡ là 16
```

Ta có thể thêm phần tử bằng cách dùng phương thức Add():

```
vectors.Add(new Vector(2,2,2));

vectors.Add(new Vector(3,5,6));
```

Arraylist xem tất cả các phần tử của nó như là các tham chiếu đối tượng. Nghĩa là ta có thể lưu trữ bất kì đối tượng nào mà ta muốn trong một Arraylist. Nhưng khi truy nhập đến đối tượng, ta sẽ cần ép kiểu chúng trở lại kiểu dữ liệu tương đương:

```
Vector element1 = (Vector)vectors[1];
```

Ví dụ này cũng chỉ ra Arraylist định nghĩa một indexer, để ta có thể truy nhập những phần tử của nó với cấu trúc như mảng. Ta cũng có thể chèn các phần tử vào array list:

```
vectors.Insert(1, new Vector(3,2,2)); // chèn vào vị trí 1
```

Đây là phương thức nạp chồng có ích khi ta muốn chèn tất cả các phần tử trong một collection vào arraylist ta có thể bỏ một phần tử :

```
vectors.RemoveAt(1); // bỏ đối tượng ở vị trí 1
```

Ta cũng có thể cung cấp một đối tượng tham chiếu đến một phương thức khác, Remove(). Nhưng làm điều này sẽ mất nhiều thời gian hơn vì arraylist phải quét qua toàn bộ mảng để tìm đối tượng.

Lưu ý rằng việc thêm và bỏ một phần tử sẽ làm cho tất cả các phần tử theo sau phải bị thay đổi tương ứng trong bộ nhớ, thậm chí nếu cần thì có thể tái định vị toàn bộ Arraylist. Ta có thể cập nhật hoặc đọc dung lượng qua thuộc tính :

```
vectors.Capacity = 30;
```

Tuy nhiên việc thay đổi dung lương đó sẽ làm cho toàn bộ Arraylist được tái định vị đến một khối bộ nhớ mới với dung lượng được yêu cầu.

Để biết số phần tử thực sự trong arraylist ta dùng thuộc tính Count :

```
int nVectors = vectors.Count;
```

Một arraylist có thể thực sự hữu ích nếu ta cần xây dựng một mảng đối tượng mà ta không biết kích cỡ của mảng sẽ là bao nhiêu. Trong trường hợp đó, ta có thể xây dựng 'mảng' trong Arraylist, sau đó sao chép Arraylist trở lại mảng khi ta hoàn thành xong nếu

ta thực sự cần dữ liệu như là một mảng (ví dụ nếu mảng được truyền đến một phương thức xem mảng là một thông số). Mối quan hệ giữa Arraylist và Array theo một cách nào đó giống như mối quan hệ giữa StringBUilder và String không như lớp StringBuilder, không có phương thức đơn nào để làm việc chuyển đổi từ một arraylist sang array. Ta phải dùng một vòng lặp để sao chép thủ công trở lại. Tuy nhiên ta chỉ phải sao chép tham chiếu chứ không phải đối tượng:

```
// vectors is an ArrayList instance being used to store Vector instances
Vector [] vectorsArray = new Vector[vectors.Count];
for (int i=0 ; i< vectors.Count ; i++)
vectorsArray[i] = (Vector)vectors [i];
```

1.4.2 Collections

Ý tưởng của Collection là nó trình bày một tập các đối tượng mà ta có thể truy xuất bằng việc bước qua từng phần tử. Cụ thể là một tập đối tượng mà ta có thể truy nhập sử dụng vòng lặp foreach. nói cách khác, khi viết một thứ gì đó như:

```
foreach (string nextMessage in messageSet)
{
DoSomething(nextMessage);
}
```

Ta xem biến messageSet là một collection. Khả năng để dùng vòng lặp foreach là mục đích chính của collection.

Tiếp theo ta tìm hiểu chi tiết collection là gì và thi hành một collection riêng bằng việc chuyển ví dụ Vector mà ta đã phát triển .

1.4.2.1 Collection là gì ?

Một đối tượng là một collection nếu nó có thể cung cấp một tham chiếu đến một đối tượng có liên quan, được biết đến như là enumarator, mà có thể duyệt qua từng mục trong collection. đặc biệt hơn, một collection phải thi hành một interface.

System.Collections.IEnumerable. IEnumerable định nghĩa chỉ một phương thức như sau:

```
interface IEnumerable
{
IEnumerator GetEnumerator();
}
```

Mục đích của GetEnumarator() là để trả về đối tượng enumarator. Khi ta tập họp những đoạn mã trên đối tượng enumarator được mong đợi để thi hành một interface, System.Collections.IEnumerator. Ngoài ra còn có một interface khác Icollection, được dẫn xuất từ IEnumerable. Những collection phức tạp hơn sẽ thi hành interface này. Bên cạnh GetEnumerator(), nó thi hành một thuộc tính trả về trực tiếp số phần tử trong collection. Nó cũng có đặc tính hổ trợ việc sao chép collection đến một mảng và có thể cung cấp thông tin đặc tả nếu đó là một luồng an toàn.tuy nhiên trong phần này ta chỉ xem xét interface IEnumerable.

IEnumarator có cấu trúc sau:

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

IEnumarator làm việc như sau: đối tượng thực thi nên được kết hợp với một collection cụ thể. Khi đối tượng này được khởi động lần đầu tiên, nó chưa trỏ đến bất kì một phần tử nào trong collection, và ta phải gọi MoveNext(), mà sẽ di chuyển enumarator

để nó chuyển đến phần tử đầu tiên trong collection. Ta có thể nhận phần tử này với thuộc tính Current. Current trả về một tham chiếu đối tượng, vì thế ta sẽ ép kiểu nó về kiểu đối tượng mà ta muốn tìm trong Collection. Ta có thể làm bất cứ điều gì ta muốn với đối tượng đó sau đó di chuyển đến mục tiếp theo trong collection bằng cách gọi MoveNext() lần nữa. Ta lập lại cho đến khi hết mục trong collection- khi current trả về null. Nếu muốn ta có thể quay trở về vị trí đầu trong collection bằng cách gọi Reset(). Lưu ý rằng Reset() thực sự trả về trước khi bắt đầu collection, vì thế nếu muốn di chuyển đến phần tử đầu tiên ta phải gọi MoveNext() collection là một kiểu cơ bản của nhóm đối tượng. Bởi vì nó không cho phép ta thêm hoặc bỏ mục trong nhóm. Tất cả ta có thể làm là nhận các mục theo một thứ tự được quyết định bởi collection. Và kiểm tra chúng, thậm chí ta không thể thay thế hoặc cập nhật mục vì thuộc tính current là chỉ đọc. Hầu như cách dùng thường nhất của collection là cho ta sự thuận tiện trong cú pháp của lặp foreach.

Mảng cũng là một collection, nhưng lệnh foreach làm việc tốt hơn mảng.

Ta có thể xem vòng lặp foreach trong C# là cú pháp ngắn trong việc viết:

```
{
  IEnumerator enumerator = MessageSet.GetEnumerator();
  string nextMessage;
  enumerator.MoveNext();
  while ( (nextMessage = enumerator.Current) != null)
  {
    DoSomething(nextMessage);   // NB. We only have read access
                                // toNextMessage
    enumerator.MoveNext();
  }
}
```

Một khía cạnh quan trọng của collection là bộ đếm được trả về như là một đối tượng riêng biệt.lý do là để cho phép khả năng có nhiều hơn một bộ đếm có thể áp dụng đồng thời trong cùng collection.

1.4.2.2 Thêm collection hỗ trợ cấu trúc Vector

Trong lần cuối cùng ta nói về Vector, một thể hiện của Vector chứa đựng 3 phần: x,y,z. Nó có thể được xem một thể hiện Vector là một mảng, để ta có thể truy nhập vào phần x bằng cách viết someVector[0], phần y bằng cách viết someVecor[1] và z là someVector[2].

Bây giờ ta sẽ mở rộng cấu trúc vector, dự án VectorAsCollection mà cũng có thể quét qua các phần của một vector bằng cách viết :

```
foreach (double component in someVector)
Console.WriteLine("Component is " + component);
```

Nhiệm vụ đầu tiên của ta là biểu thị vector như là một collection bằng việc cho nó thực thi interface IEnumerable, ta bắt đầu bằng việc cập nhật khai báo của cấu trúc vector:

```
struct Vector : IFormattable, IEnumerable
{
public double x, y, z;
Bây giờ ta thi hành interface IEnumerable :
public IEnumerator GetEnumerator()
{
return new VectorEnumerator(this);
}
```

Việc thi hành GetEnumerator() hầu như là đơn giản, nhưng nó tuỳ thuộc trên sự tồn tại của một lớp mới, VectorEnumerator, mà ta cần định nghĩa. Vì VectorEnumerator không phải là một lớp mà bất kì đoạn mã bên ngoài có thể thấy trực tiếp, ta khai báo nó là lớp private bên trong cấu trúc Vector. Việc định nghĩa nó như sau:

```
private class VectorEnumerator : IEnumerator
{
```

```
Vector theVector;      // Vector object that this enumerato refers to
int location;   // which element of theVector the enumerator is
                // currently referring to

public VectorEnumerator(Vector theVector)
{
  this.theVector = theVector;
  location = -1;
}

public bool MoveNext()
{
  ++location;
  return (location > 2) ? false : true;
}

public object Current
{
  get
  {
    if (location < 0 || location > 2)
      throw new InvalidOperationException(
        "The enumerator is either before the first element or " +
        "after the last element of the Vector");
    return theVector[(uint)location];
  }
}

public void Reset()
{
  location = -1;
```

```
  }
}
```

Khi được yêu cầu như một bộ đếm, VectorEnumerator thi hành interface IEnumerator. Nó cũng chứa hai trường thành viên, theVector, một tham chiếu đến Vector (collection) mà bộ đếm kết hợp, location, một số nguyên mà chỉ định nơi trong collection mà bộ đếm tham chiếu đến.

Cách làm việc là xem location như là chỉ mục và thi hành enumerator để truy nhập Vector như mảng.khi truy nhập vector như mảng giá trị chỉ mục là 0,1,2 – ta mở rộng bằng cách dùng -1 như là giá trị chỉ định bộ đếm trước khi bắt đầu collection,và 3 để chỉ nó đến cuối của collection. Vì vậy, việc khởi tạo của trường nay là -1 trong hàm dựng VectorEnumerator :

```
public VectorEnumerator(Vector theVector)
{
this.theVector = theVector;
location = -1;
}
```

Lưu ý rằng hàm dựng cũng lấy một tham chiếu đến thể hiện của Vector mà chúng ta định đếm - điều này được cung cấp trong phương thức Vector.GetEnumerator :

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

1.4.3 Dictionaries

Từ điển trình bày một cấu trúc dữ liệu rất phức tạp mà cho phép ta truy nhập vào các phần tử dựa trên một khoá nào đó, mà có thể là kiểu dữ liệu bất kì.ta hay gọi là bảng ánh xạ hay bảng băm.Từ điển được dùng khi ta muốn lưu trữ dữ liệu như mảng nhưng muốn dùng một kiểu dữ liệu nào đó thay cho kiểu dữ liệu số làm chỉ mục.nó cũng cho

phép ta thêm hoặc bỏ các mục , hơi giống danh sách mảng tuy nhiên nó không phải dịch chuyển các mục phía sau trong bộ nhớ.

Ta minh họa việc dùng từ điển trong ví dụ sau:MortimerPhonesEmployees. Trong ví dụ này công ty điện thoại có vài phần mềm xử lí chi tiết nhân viên. Ta cần một cấu trúc dữ liệu chứa dữ liệu của nhân viên. Ta giả sử rằng mỗi nhân viên trong công ty được xác định bởi ID nhân viên, là tập kí tự như B342… và được lưu trữ thành đối tượng EmployyeeID. Chi tiết của nhân viên được lưu trữ thành đối tượng EmployeeData, ví dụ chỉ chứa ID, tên, lương của nhân viên.

Giả sử ta có EmployeeID:

```
EmployeeID id = new EmployeeID("W435");
```

Và ta có một biến gọi là employees, mà ta có thể xem như một mảng đối tượng EmployeeData. Thực sự đó không phải là mảng - đó là từ điển và bởi vì nó là từ điển nên ta có thể lấy chi tiết của một nhân viên thông qua ID được khai báo trên:

```
EmployeeData theEmployee = employees[id];

// lưu ý rằng ID không phải kiểu số- đó là một thể hiện của EmployeeID
```

Ta có thể dùng kiểu dữ liệu bất kì làm chỉ mục, lúc này ta gọi là khoá chứ không phải là chỉ mục nữa. Khi ta cung cấp một khoá truy nhập vào một phần tử (như ID trên), xử lí trên giá trị của khoá và trả về một số nguyên tuỳ thuộc vào khoá, và được dùng để truy nhập vào 'mảng' để lấy dữ liệu.

1.4.3.1 Từ điển trong .NET

Trong .NET , từ điển cơ bản được trình bày qua lớp Hasthable, mà cách làm việc cũng giống như từ điển thực. Nghĩa là một bảng băm có thể lưu trữ bất kì cấu trúc dữ liệu nào ta muốn.

Ta có thể tự định nghĩa một lớp từ điển riêng cụ thể hơn. Microsoft cung cấp một lớp cơ sở trừu tượng, DictionaryBase, cung cấp những chức năng cơ bản của từ điển mà ta có thể dẫn xuất đến lớp mà ta muốn tạo. Nếu khoá là chuỗi ta có thể dùng lớp

System.Collections.Specialized.StringDictionary thay cho Hasthable. khi tạo một Hasthable ta có thể chỉ định kích thước khởi tạo:

```
Hasthable employees = new Hasthable(53);
```

Ở đây ta chọn số 53 bởi vì thuật toán bên trong được dùng cho từ điển làm việc hiệu quả hơn nếu kích thước của nó là một số nguyên tố.

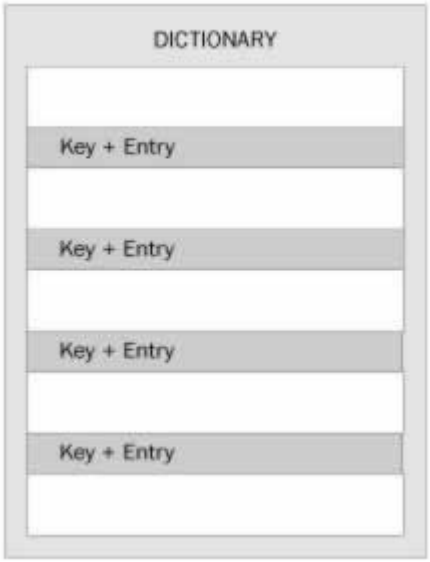
Thêm đối tượng vào từ điển ta dùng phương thức Add(), có 2 thông số kiểu object: thông số đầu là khoá, thứ hai là một tham chiếu đến dữ liệu. ví dụ:

```
EmployeeID id;
EmployeeData data;

// khởi tạo id và dữ liệu.
// giả sử employees là một thể hiện của bảng băm
//mà chứa đựng các tham chiếu EmployeeData
employees.Add(id, data);
để nhận dữ liệu ta cung cấp khoá cho nó:
EmployeeData data = employees[id];
để bỏ một mục ta cung cấp khoá và gọi :
employees.Remove(id);
Để đếm số mục trong từ điển ta dùng thuộc tính Count:
int nEmployees = employees.Count;
```

Việc lưu trữ trong từ điển không theo phải theo kiểu từ trên xuống, nghĩa là ta không thể tìm thấy một khối lớn dữ liệu ở phần đầu của cấu trúc và một khối rỗng ở phần cuối. biểu đồ sau minh hoạ cho việc lưu trữ trong từ điển, các phần không đánh dấu là

rỗng:



1.4.3.2 Cách từ điển làm việc:

Hasthable (hay bất kì lớp từ điển nào khác) sử dụng vài thuật toán để thực hiện việc đặt mỗi đối tượng dựa trên khoá. Có 2 giai đoạn, và phần mã cho từng giai đoạn phải được cung cấp bởi lớp khoá.

Một phần của thuật toán thực thi bởi lớp khoá gọi là băm (vì vậy có thuật ngữ bảng băm) và lớp Hasthable tìm một nơi cụ thể cho thuật toán băm. nó nhìn vào phương thức Gethashcode() trong đối tượng của ta, mà thừa kế từ System.Object() nếu ta nạp chồng GetHashCode().

Cách nó làm việc là Gethashcode() trả về một số nguyên. Bằng cách nào đó nó dùng giá trị của khoá để sinh ra một số nguyên. Hasthable sẽ lấy số nguyên này và làm các việc xử lí khác trên nó mà liên quan đến việc tính toán toán học phức tạp, và trả về chỉ mục của mục được lưu trữ tương ứng với khóa trong từ điển. Ta không đi sâu vào thuật toán này nhưng ta sẽ tìm hiểu tại sao nó liên quan đến số nguyên tố và tại sao dung lượng bảng băm nên là số nguyên tố.

Có một số yêu cầu nghiêm ngặt khi ta nạp chồng GetHashCode(). Những yêu cầu này nghe có vẻ trừu tượng nhưng qua ví dụ MortimerPhonesEmployees ta sẽ thấy rằng không quá khó để viết lớp khoá thỏa mãn những đòi hỏi sau:

- Phải nhanh (bởi vì việc đặt và lấy các mục trong một từ điển được coi là nhanh)
- Phải được đồng nhất - nếu ta cho 2 khoá cùng giá trị thì chúng phải cho cùng giá trị trong băm.
- Cho những giá trị khả dĩ trong khoảng giá trị của một số kiểu int.

Lí do của điều kiện cuối là: điều gì sẽ xảy ra nếu ta lấy 2 mục trong từ điển mà khi băm cả hai đều cho cùng một chỉ mục?

Nếu điều này xảy ra, lớp từ điển sẽ phải bắt đầu tìm kiếm vị trí trống có giá trị gần nhất để lưu trữ mục thứ hai.

Xung đột giữa các khóa cũng gia tăng khi từ điển đầy, vì thế cách tốt nhất là bảo đảm dung lượng lớn hơn số phần tử thực sự trong nó. Vì lí do này mà Hasthable tự định vị lại kích cỡ của nó để tăng dung lượng trước khi nó đầy. Tỷ lệ của bảng mà đầy gọi là load. Ta có thể thiết lập giá trị lớn nhất mà ta muốn load đến trước khi Hasthable tái định vị theo hàm dựng Hasthable khác :

```
// dung lượng =50, Max Load = 0.5
Hasthable employees = new Hasthable(50, 0.5);
```

Max load càng nhỏ bảng băm làm việc càng hiệu quả nhưng càng cần nhiều vùng nhớ. Khi bảng băm tái định vị để tăng dung lượng, luôn chọn một số nguyên tố làm dung lượng mới.

Một điểm quan trọng khác là thuật toán băm phải đồng nhất. Nếu 2 đối tượng chứa những gì ta coi như là dữ liệu trùng, thì chúng phải cho cùng một giá trị băm, và điều này dẫn đến một giới hạn quan trọng trên cách nạp chồng phương thức Equals() và Gethashcode() của System.Object. Cách mà Hasthable quyết định 2 khoá a và b là bằng nhau là nó gọi a.equals(b). Nghĩa là ta phải chắc rằng điều sau luôn đúng:

Nếu a.equals(b) là đúng thì a.gethashcode() và b.gethashcode() phải luôn trả về cùng mã băm. Nếu ta cố ý nạp chồng những phương thức này để những câu lệnh trên không đúng thì bảng băm sẽ không làm việc bình thường. Ví dụ như ta đặt một đối tượng vào bảng băm nhưng không nhận lại được nó hay nhận lại được nhưng không đúng mục.

Trong system.object điều kiện này đúng, vì Equals() đơn giản so sánh 2 tham chiếu và gethashcode() thực sự trả về một băm dựa trên địa chỉ của đối tượng. Nghĩa là bảng băm dựa trên một khoá mà không nạp chồng những phương thức này sẽ làm việc đúng. Tuy nhiên, vấn đề với cách làm này là những khóa coi là bằng chỉ nếu chúng là cùng đối tượng. Nghĩa là khi đặt một đối tượng vào từ điển ta phải nối tham chiếu đến khóa. Ta không thể khởi tạo một khóa khác sau đó mà có cùng giá trị, vì cùng giá trị được định nghĩa theo nghĩa là cùng một thực thể. nghĩa là nếu ta không nạp chồng bản object của Equals() và Gethashcode(), lớp của ta sẽ không thuận lợi để dùng trong bảng băm. Tốt hơn nếu thi hành gethashcode() sinh ra một băm dựa trên giá trị của khoá hơn là địa chỉ của nó trong bộ nhớ. Do đó ta sẽ cần nạp chồng gethashcode() va equals() trong bất kì lớp nào mà ta muốn nó được sử dụng như khoá System.String có những phương thức nạp chồng tương đương, Equals() được nạp chồng để cung cấp giá trị so sánh, và gethashcode() được nạp chồng để trả về một băm dựa trên giá trị của chuỗi. Vì lí do này thuận lợi để dùng chuỗi như là khoá trong từ điển.

1.4.3.3 Ví dụ MortimerPhonesEmployees

Đây là chương trình thiết lập từ điển nhân viên. Chương trình khởi tạo từ điển, thêm vài nhân viên và sau đó mời người dùng gõ vào Id nhân viên. Mỗi khi gõ, chương trình dùng ID để trỏ vào tử điển và nhận chi tiết nhân ivên. quy trình lặp lại cho đến khi người dùng gõ X:

MortimerPhonesEmployees

Enter employee ID (format:A999, X to exit)> B001

Employee: B001: Mortimer £100,000.00

Enter employee ID (format:A999, X to exit)> W234

Employee: W234: Arabel Jones £10,000.00

Enter employee ID (format:A999, X to exit)> X

Các lớp của chương trình :

```
class EmployeeID
  {
    private readonly char prefix;
    private readonly int number;
    public EmployeeID(string id)
    {
      prefix = (id.ToUpper())[0];
      number = int.Parse(id.Substring(1,3));
     }

    public override string ToString()
    {
      return prefix.ToString() + string.Format("{0,3:000}", number);
    }

    public override int GetHashCode()
    {
      return ToString().GetHashCode();
    }

    public override bool Equals(object obj)
    {
      EmployeeID rhs = obj as EmployeeID;
      if (rhs == null)
        return false;
      if (prefix == rhs.prefix && number == rhs.number)
        return true;
      return false;
    }
  }
```

Phần định nghĩa đầu tiên của lớp lưu trữ ID.bao gồm một kí tự chữ đứng đầu theo sau là 3 kí tự số. Ta dùng kiểu char để lưu chữ đầu và int để lưu phần sau. Hàm dựng nhận một chuỗi và ngắt nó thành những trường này.phương thức Tostring() trả về ID là chuỗi:

```
return prefix.ToString() + string.Format("{0,3:000}", number);
```

Phần đặc tả định dạng (0,3:000) để phần int chứa số được điền thêm số 0 ví dụ ta sẽ có B001 không phải B1ta đến 2 phương thức nạp chồng trong từ điển :

Đầu tiên là Equals() để so sánh giá trị của những thể hiện EmployeeID :

```
public override bool Equals(object obj)
{
EmployeeID rhs = obj as EmployeeID;
if (rhs == null)
return false;
if (prefix == rhs.prefix && number == rhs.number)
return true;
return false;
}
}
```

Đầu tiên ta kiểm tra xem đối tượng trong thông số có phải là một thể hiện của EmployeeID không bằng cách thử ép kiểu nó thành đối tượng EmployeeID. Sau đó ta chỉ việc so sánh những trường giá trị của nó có chứa cùng giá trị như đối tượng này không. Tiếp theo là Gethashcode():

```
public override int GetHashCode()
{
string str = this.ToString();
return str.GetHashCode();
}
```

Phần trên ta đã xem xét các yêu cầu giới hạn mà mã băm được tính phải thỏa mãn. Tất nhiên có những cách để nghĩ ra những thuật toán băm hiệu quả và đơn giản. Tói chung, lấy một trường, nhân nó với một số nguyên tố lớn, và công những kết quả lại với nhau là một cách tốt. Nhưng ta không phải làm những điều đó vì MIcrosoft đã làm toàn bộ trong lớp String, vì thế ta có thể lợi dụng lớp này để tạo ra số dựa trên nội dung của chuỗi. Nó sẽ thoã mãn tất cả những yêu cầu của mã băm.

Chỉ có một khuyết điểm khi dùng phương thức này là có vài việc thi hành đã mất kết hợp với việc chuyển đổi lớp EmployeeID thành chuỗi trong phần đầu tiên. Nếu không muốn điều này ta sẽ cần thiết kế mã băm riêng thiết kế thuật toán băm là một chủ đề phức tạp mà ta không không thể đi sâu trong cuốn sách này. Tuy nhiên ta sẽ đưa ra một cách đơn giản cho vấn đế này, mà chỉ việc nhân số dựa trên những trường thành phần của lớp với số nguyên tố khác (nhân bởi một số nguyên tố khác giúp ta ngăn ngừa sự kết hợp giá trị khác nhau của các trường từ việc cho cùng mã băm):

```
public override int GetHashCode() // alternative implementation
{
return (int)prefix*13 + (int)number*53;
}
```

Ví dụ này sẽ làm việc nhanh hơn Tostring() mà ta dùng ở trên. Tuy nhiên khuyết điểm là mã băm sinh ra bởi các employeeID khác nhau thì không trải rộng trên vùng số kiểu int.

Tiếp theo ta xem lớp chứa dữ liệu nhân viên:

```
class EmployeeData
  {
    private string name;
    private decimal salary;
    private EmployeeID id;
```

```
    public EmployeeData(EmployeeID id, string name, decimal salary)
    {
      this.id = id;
      this.name = name;
      this.salary = salary;
    }

    public override string ToString()
    {
      StringBuilder sb = new StringBuilder(id.ToString(), 100);
      sb.Append(": ");
      sb.Append(string.Format("{0,-20}", name));
      sb.Append(" ");
      sb.Append(string.Format("{0:C}", salary));
      return sb.ToString();
    }
  }
```

Ta dùng đối tượng StringBuilder để sinh ra chuỗi đại diện cho đối tượng Employeedata. Cuối cùng ta viết đoạn mã kiểm tra lớp TestHarness:

```
class TestHarness
{

  Hashtable employees = new Hashtable(31);

  public void Run()
  {
    EmployeeID idMortimer = new EmployeeID("B001");
    EmployeeData mortimer = new EmployeeData(idMortimer, "Mortimer",
```

```
                                100000.00M);
EmployeeID idArabel = new EmployeeID("W234");
EmployeeData arabel= new EmployeeData(idArabel, "Arabel Jones",
                                10000.00M);
employees.Add(idMortimer, mortimer);
employees.Add(idArabel, arabel);

while (true)
{
  try
  {
    Console.Write("Enter employee ID (format:A999, X to exit)> ");
    string userInput = Console.ReadLine();
    userInput = userInput.ToUpper();
    if (userInput == "X")
      return;
    EmployeeID id = new EmployeeID(userInput);
    DisplayData(id);
  }
  catch (Exception e)
  {
    Console.WriteLine("Exception occurred. Did you use the correct
                format for the employee ID?");
    Console.WriteLine(e.Message);
    Console.WriteLine();
  }

  Console.WriteLine();
}
```

```
  }

  private void DisplayData(EmployeeID id)
  {
    object empobj = employees[id];
    if (empobj != null)
    {
      EmployeeData employee = (EmployeeData)empobj;
      Console.WriteLine("Employee: " + employee.ToString());
    }
    else
      Console.WriteLine("Employee not found: ID = " + id);
  }
}
```

Đầu tiên ta thiết lập dung lượng của từ điển là số nguyên tố, 31, phần chính của lớp này là phương thức run(). Đầu tiên là thêm vài nhân viên vào từ điển mortimer và arabel và thêm chi tiết của họ vào:

```
employees.Add(idMortimer, mortimer);
employees.Add(idArabel, arabel);
```

Tiếp theo ta bước vào vòng lặp để yêu cầu người dùng nhập vào EmployeeID. Có khối try bên trong vòng lặp, bẫy những lỗi khi người dùng không gõ đúng định dạng cuả EmployeeID:

```
string userInput = Console.ReadLine();
userInput = userInput.ToUpper();
if (userInput == "X")
return;
EmployeeID id = new EmployeeID(userInput);
```

Nếu hàm dựng EployeeID đúng, ta trình bày kết hợp nhân viên bằng cách gọi DisplayData(). Đây là phương thức mà ta muốn truy nhập vào từ điển với cú pháp mảng. nhận dữ liệu nhân viên với ID là việc đầu tiên trong phương thức này:

```
private void DisplayData(EmployeeID id)
{
object empobj = employees[id];
```

Nếu không có nhân viên với ID tên, thì employees[id] trả về Null, mà ta sẽ đưa ra thông báo lỗi nếu ta tìm thấy. Nếu không ta ép kiểu tham chiếu empobj thành EmployeeData. Khi ta có tham chiếu EmployeeID, ta trình bày dữ liệu của nó bằng phương thức EmployeeData.ToString():

```
EmployeeData employee = (EmployeeData)empobj;
Console.WriteLine("Employee: " + employee.ToString());
Ta có phần cuối của mã - phương thức main() kích hoạt ví dụ trên . khởi tạo đối tượng
TestHarness và chạy nó:
static void Main()
{
TestHarness harness = new TestHarness();
harness.Run();
}
```