

Mục lục

Mục lục	1
Mở đầu.....	4
Chương 1: Giới thiệu C#.....	6
1.1 Khung lập trình .NET là gì?	6
1.2 Các thành phần bên trong khung lập trình .Net.....	6
1.3 Viết ứng dụng sử dụng khung lập trình .Net	7
1.4 Assembly	8
1.5 Mã quản lý	8
1.6 Bộ thu gom rác (Garbage Collection)	9
1.7 Các bước tạo ứng dụng .Net	9
1.8 Ngôn ngữ lập trình C#.....	9
1.9 Cài đặt Visual C# 2010 Express	10
Chương 2: Căn bản C#	13
2.1 Viết chương trình C# đầu tiên	14
2.2 Biến.....	16
2.3. Hằng.....	17
2.4 Kiểu dữ liệu cơ bản.....	17
2.5 Xuất nhập qua Console.....	21
2.6 Biểu thức.....	23
2.7 Chuyển đổi kiểu.....	25
2.8 Kiểu liệt kê.....	27
2.9 Mảng	29
2.10 Không gian tên (Namespace)	29
2.11 Câu lệnh using	30
2.12 Phương thức Main()	31
2.13 Sử dụng chú thích.....	31

2.14 Chỉ dẫn tiên xử lý trong C#	31
Chương 3: Cấu trúc điều khiển.....	33
3.1 Lệnh rẽ nhánh if.....	33
3.2 Lệnh switch.....	34
3.3 Lệnh lặp	35
3.4 Hàm	40
3.5 Ứng dụng minh họa	43
3.6 Sử dụng công cụ dò lỗi của Visual Studio.NET.....	50
3.7 Bài tập.....	52
Chương 4: Đối tượng và kiểu	53
4.1 Lớp.....	53
4.2 Thành viên của lớp	56
4.3 Cấu trúc (Struct)	73
4.4 Quá tải toán tử	81
4.5 Ứng dụng minh họa	84
4.5 Bài tập.....	87
Chương 5: Sự kế thừa	89
5.1 Các kiểu kế thừa	91
5.2 Thực thi sự kế thừa.....	93
5.3 Đa hình (polymorphism)	98
5.4 Giao diện.....	103
5.5 Ứng dụng minh họa	108
Chương 6: Sự ủy nhiệm, sự kiện và quản lý lỗi.....	114
6.1 Sự ủy nhiệm (delegate).....	114
6.2 Sự kiện (Event).....	120
6.3 Quản lý lỗi và ngoại lệ.....	126

Chapter 7: Quản lý bộ nhớ và con trỏ.....	132
7.1 Quản lý bộ nhớ	132
7.2 Giải phóng tài nguyên	134
7.3 Mã không an toàn	138
Chương 8: Chuỗi, biểu thức quy tắc và tập hợp	143
8.1 System.String.....	143
8.2 Biểu thức quy tắc	146
8.3 Tập hợp đối tượng	148
8.4 Thuộc tính (attribute) tùy chọn.....	159
8.5 Reflection.....	163
8.6 Thao tác tập tin qua luồng	167
Hướng dẫn phần thực hành.....	170
Phụ lục	177
Tài liệu tham khảo.....	193

Mở đầu

Lập trình hướng đối tượng (OOP-Object Oriented Programming) đóng một vai trò quan trọng trong việc xây dựng và phát triển các chương trình hay ứng dụng trên nhiều nền tảng và môi trường khác nhau như windows, linux, web.... Đặc biệt, hiện tại các ngôn ngữ lập trình cấp cao thế hệ thứ 4 (như Java hay C#...) hầu như được xây dựng là những ngôn ngữ thuần đối tượng nhằm hỗ trợ những nguyên lý căn bản cũng như các tính năng nâng cao dựa trên hướng đối tượng giúp cho việc xây dựng và phát triển ứng dụng trên hướng đối tượng dễ dàng và nhanh chóng hơn. Do đó việc tiếp cận và nắm vững các nguyên lý lập trình hướng đối tượng rất quan trọng đối với sinh viên cho việc sử dụng và ứng dụng nó cho các môn học liên quan đến lập trình và các môn học chuyên ngành ở các học kì tiếp theo.

Mục tiêu của môn học:

- Ôn tập lại các vấn đề về kĩ thuật lập trình, cách thức phát triển ứng dụng đơn giản trên C#.
- Cung cấp cho sinh viên tiếp cận và sử dụng ngôn ngữ lập trình C#.
- Cung cấp cho sinh viên kiến thức về lập trình hướng đối tượng trên ngôn ngữ lập trình C# bao gồm tính chất đóng gói, kế thừa, đa hình, giao tiếp...
- Cung cấp các kiến thức về xử lý và thao tác dữ liệu trên tập tin văn bản và nhị phân, XML.
- Cung cấp các kiến thức về sử dụng các cấu trúc dữ liệu sẵn có trên .Net trong quá trình phát triển ứng dụng như Stack, Queue, ArrayList, HashTable.
- Giới thiệu việc xây dựng và phát triển ứng dụng trên môi trường .Net. Cung cấp cho sinh viên tiếp cận và làm quen với môi trường phát triển ứng dụng dựa trên Visual Studio.

- Thông qua các chương trình và bài tập ở cuối chương giúp sinh viên có thể vận dụng được các kiến thức đã học áp dụng vào ứng dụng cụ thể.

Trong quá trình biên soạn giáo trình, chúng tôi có tham khảo và trích dẫn một số thông tin từ cuốn sách “Visual C Sharp Step by Step” của nhà xuất bản Microsoft Press và trên trang web của Microsoft.

Trong quá trình biên soạn vẫn còn thiếu sót về mặt nội dung, mong các bạn đồng nghiệp và sinh viên đóng góp ý kiến và phản hồi về thongt@dlu.edu.vn.

Chương 1: Giới thiệu C#

Mục đích của chương:

- Khung lập trình .NET và các thành phần của nó.
- Cách thức làm việc của các ứng dụng dựa trên khung lập trình .NET.
- Giới thiệu C# và mối liên hệ của C# với khung lập trình .NET.
- So sánh C# với ngôn ngữ lập trình C và một số các ngôn ngữ lập trình khác.
- Sử dụng môi trường phát triển để xây dựng ứng dụng .NET dùng ngôn ngữ C#.

1.1 Khung lập trình .NET là gì?

Khung lập trình .Net hiện tại đang sử dụng phiên bản 4.0 là một nền tảng tân tiến cho việc phát triển các ứng dụng được tạo bởi Microsoft. Mặc dù Microsoft đưa ra khung lập trình .NET chạy trên hệ điều hành Windows nhưng nó có các phiên bản khác nhau có thể làm việc trên các hệ thống khác nhau. Ví dụ như Mono là một phiên bản mã nguồn mở của khung lập trình .NET có thể chạy trên hệ điều hành như Linux hay Mac OS; hay chúng ta có thể sử dụng “Microsoft .NET Compact Framework” chạy trên các dòng thiết bị hỗ trợ số các nhân (PDA- Personal Digital Assistant) hay các dòng điện thoại thông minh. Khung lập trình .NET không hạn chế kiểu ứng dụng phát triển. Nó cho phép phát triển ứng dụng kiểu ứng dụng Windows, ứng dụng Web, ứng dụng Web Services... Ngoài ra, khung lập trình .Net được thiết kế để có thể sử dụng bất cứ ngôn ngữ nào bao gồm C#, C++, Visual Basic hay các ngôn ngữ cũ hơn như COBOL.

1.2 Các thành phần bên trong khung lập trình .Net

Khung lập trình .Net gồm tập hợp các thư viện lập trình sẵn có trong hệ điều hành Windows hay có thể được cài thêm từ bên ngoài. Nó cung cấp những giải pháp thiết yếu cho những yêu cầu thông thường của các chương trình điện toán như lập trình giao diện người dùng, truy cập dữ liệu, kết nối cơ sở dữ liệu, ứng dụng web, các giải thuật số học và giao tiếp mạng... Ngoài ra, khung lập trình .NET quản lý việc thực thi

các chương trình .NET do đó người dùng cần phải cài .NET để có thể chạy các chương trình .NET.

Các phần trong thư viện khung lập trình .Net định nghĩa một số kiểu dữ liệu cơ bản. Một kiểu dữ liệu là một biểu diễn của dữ liệu và các kiểu này có thể được sử dụng bởi các ngôn ngữ dùng khung lập trình .Net. Các kiểu này được gọi là Common Type System (CTS).

Khung lập trình .Net cũng bao gồm thư viện .NET Common Language Runtime (CLR) dùng để quản lý việc thực thi của tất cả các ứng dụng được phát triển sử dụng thư viện .Net.

1.3 Viết ứng dụng sử dụng khung lập trình .Net

Viết một ứng dụng dùng khung lập trình .Net nghĩa là viết mã chương trình bằng bất cứ ngôn ngữ lập trình nào (được hỗ trợ bởi khung lập trình .Net) sử dụng thư viện mã .Net. Trong giáo trình này chúng ta có thể sử dụng Visual Studio 2010 hay sử dụng Visual Studio Express 2010 (VSE) cho việc phát triển ứng dụng. VSE là một công cụ miễn phí có thể đáp ứng đầy đủ các tính năng cho việc học và áp dụng các nguyên lý trong môn học lập trình hướng đối tượng.

Để các đoạn mã C# được thực thi, nó phải được chuyển thành một ngôn ngữ để hệ điều hành mà ứng dụng sẽ chạy trên đó phải hiểu được gọi là mã tự nhiên (native code). Ngôn ngữ sau khi được biên dịch bởi trình biên dịch là mã biên dịch. Trong khung lập trình .Net, điều này được thực hiện qua hai giai đoạn:

1. Khi chúng ta biên dịch mã có sử dụng thư viện khung lập trình .Net, chúng ta không tạo ra mã tự nhiên của một hệ điều hành cụ thể nào. Thay vào đó mã của chúng ta sẽ được biên dịch thành mã CIL (Common Intermediate Language). Mã này không chỉ ra bất cứ hệ điều hành nào mà chương trình sẽ chạy hay chỉ ra nó được viết bởi C#. Ví dụ, nếu chúng ta dùng ngôn ngữ Visual Basic thì nó cũng được biên dịch thành ngôn ngữ này ở giai đoạn đầu tiên. Bước biên dịch

này cũng được thực hiện bởi Visual Studio khi chúng ta phát triển ứng dụng dùng ngôn ngữ C#.

2. Rõ ràng, có nhiều công việc cần thiết phải được thực hiện để thực thi một ứng dụng. Thực thi chương trình là nhiệm vụ của trình biên dịch JIT (Just-In-Time). Trình biên dịch này sẽ biên dịch CIL thành mã tự nhiên phù hợp với một hệ điều hành hay một kiến trúc máy đích nào đó. Lúc này, hệ điều hành mới có thể chạy ứng dụng.

1.4 Assembly

Khi chúng ta biên dịch một ứng dụng, mã CIL được tạo ra sẽ được lưu trữ trong một assembly. Một assembly có thể chứa các tập tin của ứng dụng thực thi (có phần mở rộng là exe) hay tập tin thư viện (có phần mở rộng là dll-dynamic link library).

Một điểm quan trọng trong các assembly là chúng chứa các siêu dữ liệu (metadata) dùng để mô tả các kiểu và phương thức được định nghĩa tương ứng trong mã. Một assembly cũng chứa siêu dữ liệu dùng để mô tả chính assembly đó. Siêu dữ liệu chứa trong một vùng được gọi là tập tin mô tả (manifest), nó cho phép kiểm tra phiên bản và tình trạng của assembly.

Với việc assembly chứa siêu dữ liệu, nó cho phép chương trình, ứng dụng hay các assembly khác có thể gọi mã trong một assembly mà không cần tham chiếu đến Registry, hoặc một dữ liệu nguồn khác.

1.5 Mã quản lý

Vai trò của CLR không kết thúc sau khi chúng ta biên dịch mã thành CIL và trình biên dịch JIT biên dịch thành mã tự nhiên. Mã được viết dùng khung lập trình .Net được quản lý lúc nó thực thi. Điều này nghĩa CLR còn quản lý các nhiệm vụ khác như: quản lý bộ nhớ, quản lý bảo mật, quản lý dò lỗi... Ngược lại các ứng dụng chạy không chịu sự giám sát của CLR được gọi là mã không quản lý. Ví dụ ngôn ngữ C++ có thể được dùng để viết các ứng dụng dạng này, nó có thể truy cập các hàm cấp thấp của hệ điều hành.

1.6 Bộ thu gom rác (Garbage Collection)

Một trong những đặc trưng quan trọng nhất của mã quản lý là khái niệm thu gom rác. Đây là một kỹ thuật của .Net nhằm đảm bảo bộ nhớ được giải phóng hoàn toàn khi ứng dụng không cần dùng nó nữa. Trước đây, nhiệm vụ này phải được thực hiện bởi các lập trình viên và với một lỗi nhỏ khi cấp phát bộ nhớ trong mã có thể gây mất bộ nhớ với kích thước lớn và có thể dẫn đến hỏng chương trình hay hệ thống.

Bộ thu gom rác sẽ theo dõi bộ nhớ và xóa bất cứ những thứ gì không cần thiết nhằm giải phóng bộ nhớ. Tuy nhiên, .Net cũng cho phép các lập trình viên chủ động trong việc quản lý bộ nhớ này bằng cách hủy các tài nguyên theo nhu cầu của mình.

1.7 Các bước tạo ứng dụng .Net

- Mã ứng dụng được viết dùng các ngôn ngữ tương thích với .Net như C#.
- Mã ứng dụng này được biên dịch thành CIL được lưu trữ trong Assembly.
- Khi thực thi, nó phải được biên dịch thành mã tự nhiên dùng trình biên dịch JIT.

1.8 Ngôn ngữ lập trình C#

Microsoft Visual C# là một ngôn ngữ mạnh mẽ nhưng đơn giản, chủ yếu hướng đến các nhà phát triển xây dựng ứng dụng trên nền tảng .NET của Microsoft. C# kế thừa những đặc trưng tốt nhất của ngôn ngữ C++ và Microsoft Visual Basic, và loại bỏ đi một số đặc trưng không thống nhất và lạc hậu với mục tiêu tạo ra một ngôn ngữ rõ ràng và logic hơn. Sức mạnh của C# còn được bổ sung thêm một số đặc trưng mới quan trọng bao gồm Generic, cơ chế lập và phương thức ẩn tên... Môi trường phát triển cung cấp bởi Visual Studio làm cho những đặc trưng này trở nên dễ sử dụng và nâng cao năng suất cho các nhà phát triển ứng dụng.

C# có thể tạo các ứng dụng dòng lệnh (console) cũng như các ứng dụng thuần văn bản chạy trên DOS hay Window. Tất nhiên, chúng ta có thể dùng C# để tạo các ứng dụng dùng cho các công nghệ tương thích với .NET. Các ứng dụng có thể viết trên C# như:

- ✓ Ứng dụng Web.

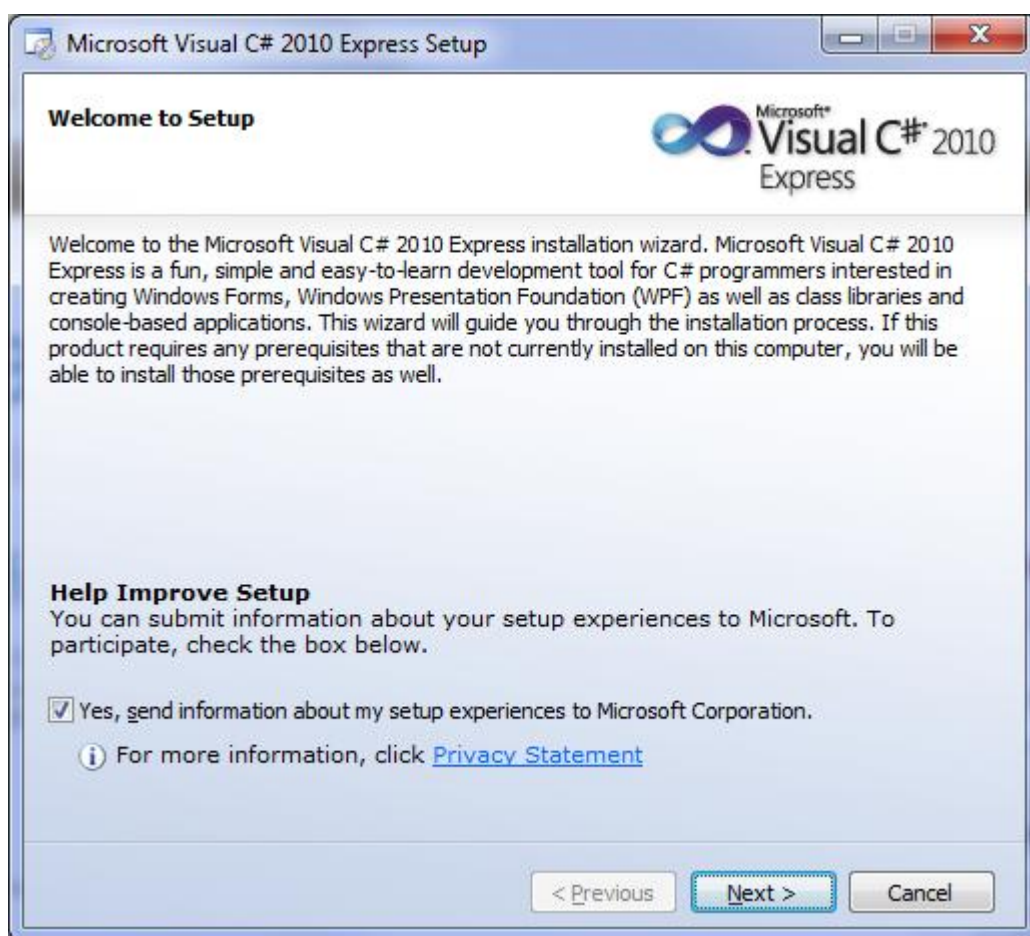
- ✓ Ứng dụng Windows.
- ✓ Web Services.

1.9 Cài đặt Visual C# 2010 Express

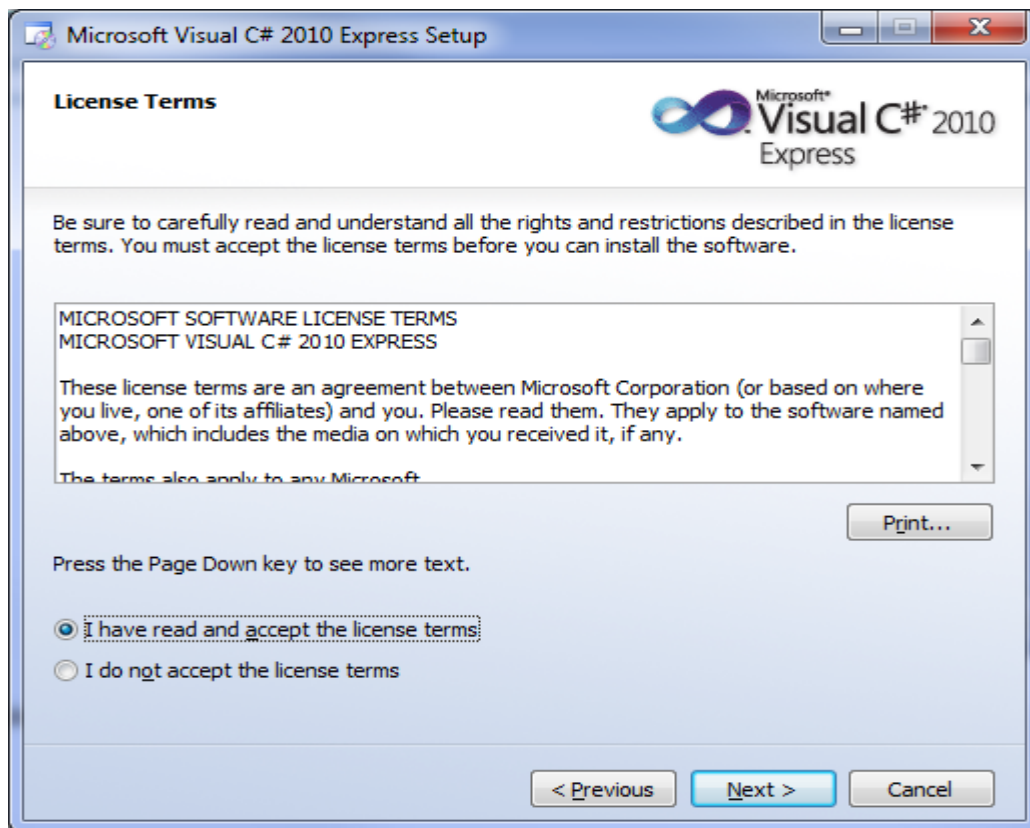
Để cài đặt Visual C# 2010 Express trực tiếp từ internet, chúng ta có thể download miễn phí từ Web Site của Microsoft:

<http://www.microsoft.com/express/Downloads/#2010-Visual-CS>.

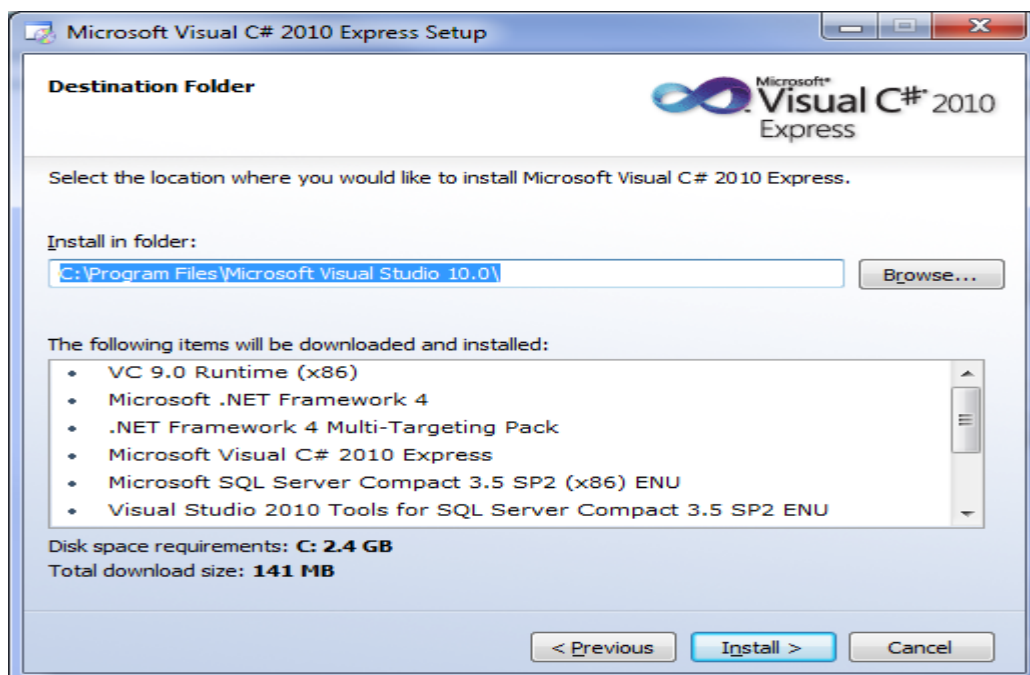
Bước 1: Chạy chương trình Setup



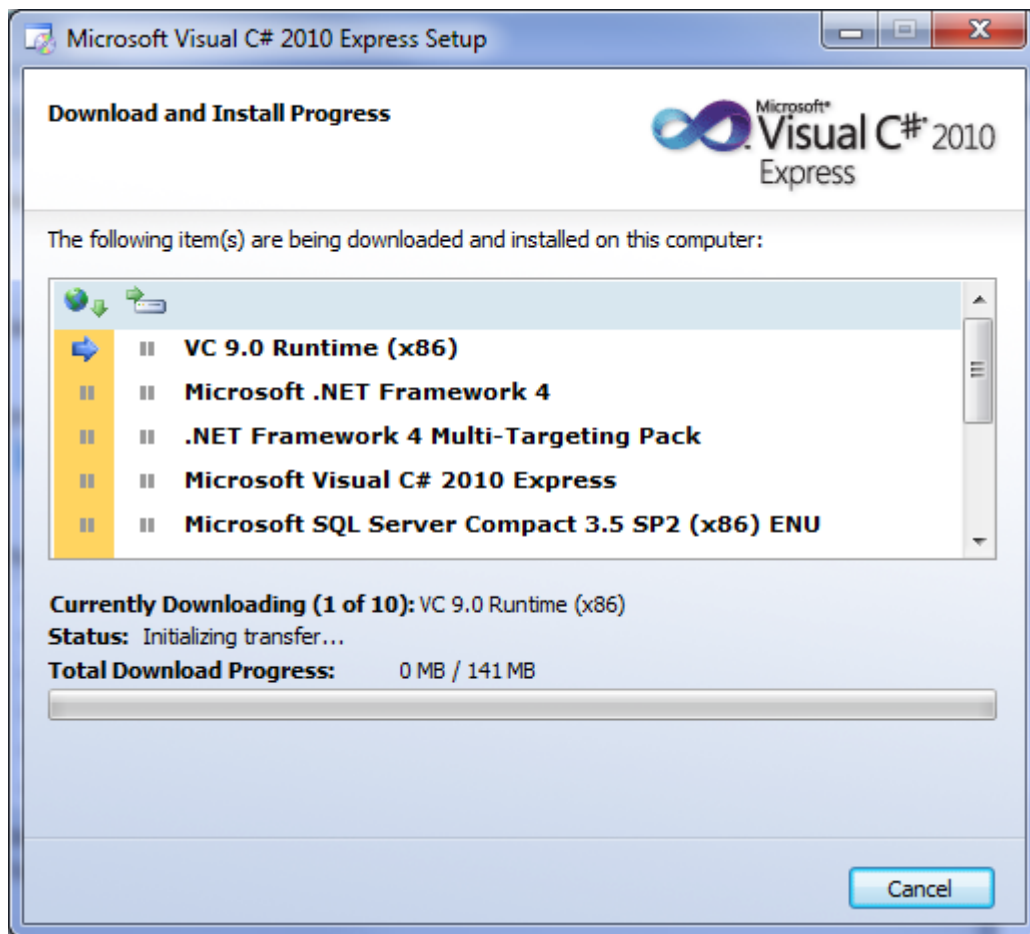
Bước 2: Đồng ý các điều kiện về bản quyền



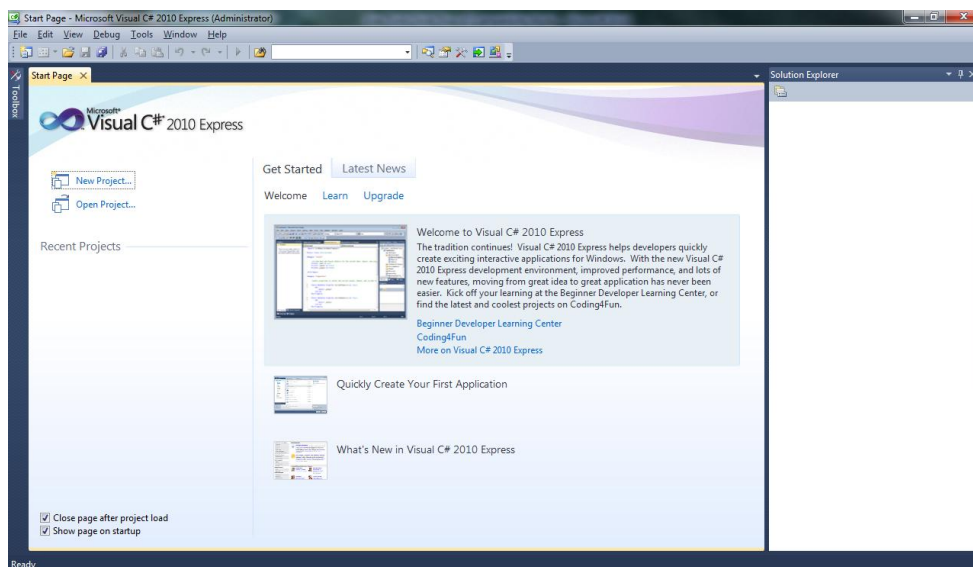
Bước 3: Chọn đường dẫn cài đặt và kiểm tra dung lượng đĩa cứng cần thiết để cài đặt



Bước 4: Tiến hành cài đặt



Bước 5: Chạy chương trình sau khi cài đặt thành công



Chương 2: Căn bản C#

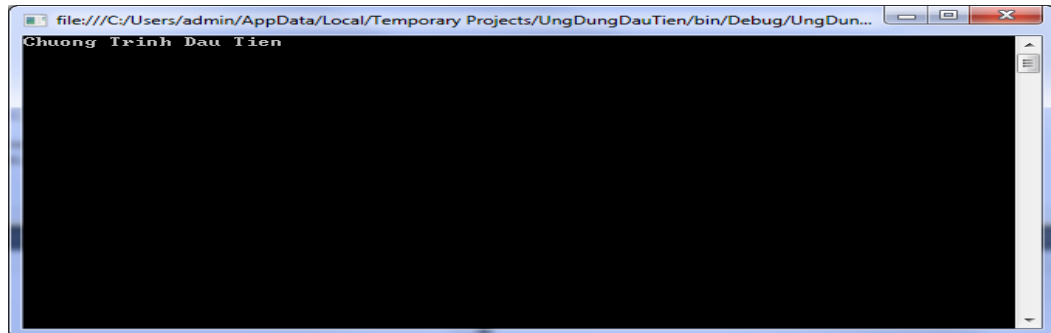
Mục đích của chương:

- Khai báo biến.
- Khởi tạo và phạm vi hoạt động của biến.
- Các kiểu dữ liệu cơ bản.
- Kiểu mảng.
- Toán tử.
- An toàn kiểu và cách để chuyển kiểu dữ liệu.
- Kiểu liệt kê (enum).
- Sử dụng không gian tên (namespace).
- Hàm Main().
- Xuất nhập chuẩn dòng System.Console.
- Sử dụng chú thích trong C#.
- Các định danh và từ khoá trong C#.

Trong môn học này, với mục đích chính là giúp cho chúng ta hiểu và áp dụng được các đặc trưng và nguyên lý của lập trình hướng đối tượng nên tất cả dự án trong giáo trình sử dụng kiểu ứng dụng Console. Ứng dụng này không sử dụng môi trường đồ họa do đó chúng ta không cần phải lo lắng về việc sử dụng các thành phần đồ họa như thế nào. Thay vào đó chúng ta chạy ứng dụng trong một cửa sổ dấu nhắc lệnh và tương tác qua cửa sổ này rất đơn giản. Việc phát triển các ứng dụng khác sẽ liên quan ở các môn học khác. Ngoài ra trong môn học này một số kiến thức căn bản về lập trình đã được trình bày chi tiết trong giáo trình lập trình cấu trúc nên chương này chỉ nhắc lại các vấn đề căn bản mấu chốt như kiểu dữ liệu, biến, cấu trúc điều khiển...

2.1 Viết chương trình C# đầu tiên

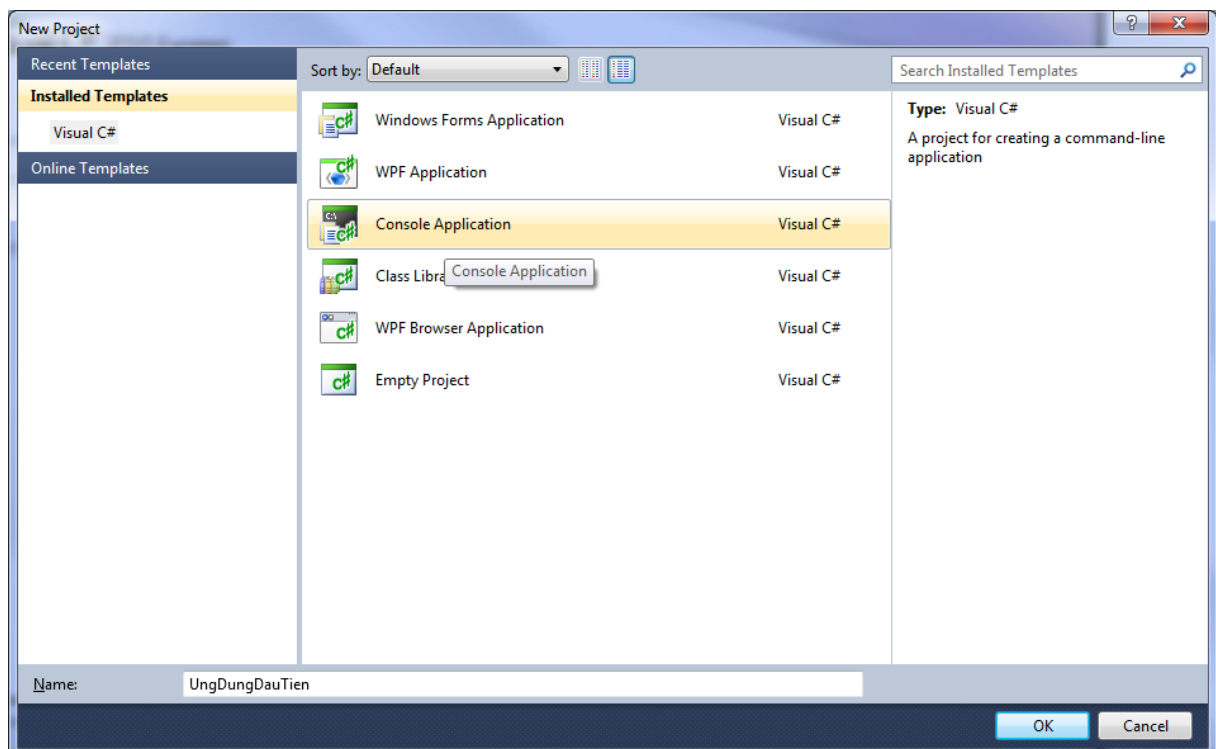
Đầu tiên chúng ta viết một chương trình đơn giản sử dụng ngôn ngữ C# nhằm xuất dòng thông báo “Ung dung dau tien” ra màn hình với kết quả như sau:



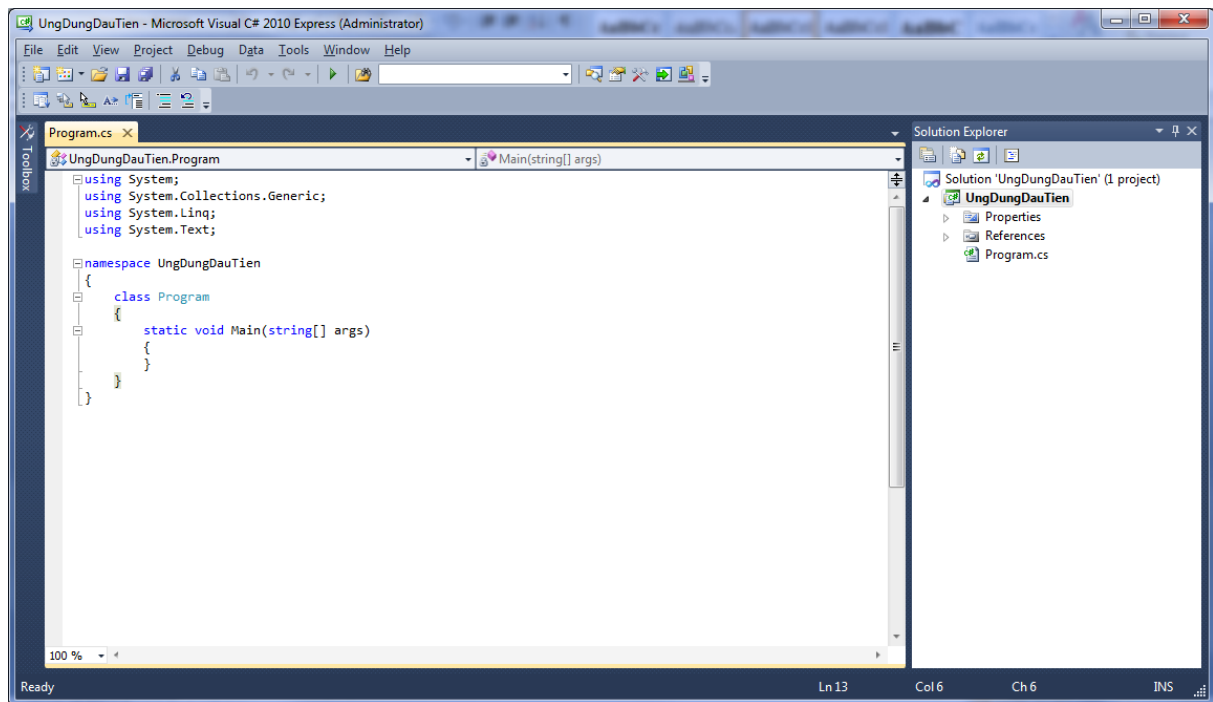
Để thực hiện được chương trình trên chúng ta thực hiện các bước như sau:

Bước 1: Chạy chương trình Microsoft Visual C# 2010 Express.

Bước 2: Tạo ứng dụng kiểu “Console Application” và đặt tên cho ứng dụng là “UngDungDauTien”



Bước 3: Kết quả sau khi thực hiện tạo dự án



Trong hình trên là môi trường phát triển ứng dụng Visual C# Express. Chúng ta thấy phần cửa sổ bên trái đang hiển thị nội dung mã của tập tin Program.cs. Tập tin này được tạo mặc định bởi VCE (Visual C# Express). Phần bên phải là cửa sổ Solution Explorer chứa “Solution”. Mỗi “Solution” có thể chứa một hay nhiều dự án (project). Trong ví dụ của chúng ta Solution “UngDungDauTien” chứa một dự án tên “UngDungDauTien”. Mỗi dự án có thể chứa nhiều tập tin .cs (tập tin chứa mã của chương trình).

Bước 4: Nhập nội dung của tập tin Program.cs như sau:

Ví dụ 2.1: Ứng dụng đầu tiên

```
1: using System;
2: using System.Collections.Generic;
3: using System.Linq;
4: using System.Text;
5:
6: namespace UngDungDauTien
7: {
8:     class Program
9:     {
10:         static void Main(string[] args)
```



```
11:      {
12:          System.Console.WriteLine("Chương Trình Đầu Tiên");
13:          System.Console.ReadLine();
14:      }
15:  }
16: }
```

Bước 6: Để chạy chương trình trên và xem kết quả chúng ta vào thực đơn Debug, chọn Start Debugging (nhấn F5).

Ngoài ra để chạy chương trình ở chế độ không debug, chúng ta có thể nhấn Ctrl + F5, khi chạy chương trình ở chế độ này cửa sổ lệnh sẽ chờ người dùng nhập vào một phím bất kì để đóng cửa sổ.

2.2 Biến

Trong chương trình một biến dùng để lưu trữ giá trị của một kiểu dữ liệu nào đó.

Cú pháp C# sau đây để khai báo một biến:

```
[bỏ_từ] kiểu_dữ_liệu định_danh;
```

- `bỏ_từ` là một trong những từ khoá: `public`, `private`, `protected`... những `bỏ_từ` này sẽ được giải thích ở các chương sau.
- `kiểu_dữ_liệu` là các kiểu dữ liệu cơ bản như số nguyên (`int`), số thực (`float`) hay các kiểu dữ liệu do người dùng định nghĩa...
- `định_danh` là tên biến.

Chú ý khi thông tin được đặt trong dấu `[]` nghĩa là thông tin này không cần bắt buộc phải khai báo.

Ví dụ dưới đây một biến tên `i` kiểu nguyên và có thể được truy cập bởi bất cứ hàm nào.

```
public int i;
```

Ta có thể gán cho biến một giá trị bằng toán tử `"="`.

```
i = 10;
```

Ta cũng có thể khai báo biến và khởi tạo giá trị cho biến như sau:

```
int i = 10;
```

Chúng ta có thể khai báo nhiều biến có cùng kiểu dữ liệu như sau:

```
1: int x = 10, y = 20;
```



```
2: int x = 10;
3: bool y = true; // khai báo trên đúng
4: int x = 10, bool = true // khai báo trên có lỗi
```

2.3. Hằng

Một hằng là một biến nhưng giá trị không thể thay đổi trong suốt thời gian thực thi chương trình. Đôi lúc ta cũng cần có những giá trị bao giờ cũng bất biến.

```
const int a = 100; // giá trị này không thể bị thay đổi
const float PI = 3.1414; // giá trị này không thể bị thay đổi
```

Hằng có những đặc điểm sau:

- Hằng bắt buộc phải được gán giá trị lúc khai báo. Một khi đã được khởi gán thì không thể thay đổi giá trị của nó.
- Trị của hằng có thể được tính toán vào lúc biên dịch. Do đó không thể gán một hằng từ một giá trị của một biến. Nếu muốn làm thế thì phải sử dụng bổ từ “read-only field”.
- Hằng bao giờ cũng static, tuy nhiên ta không thể đưa từ khoá static vào khi khai báo hằng.

Có ba thuận lợi khi sử dụng hằng:

- Hằng làm cho chương trình đọc dễ dàng hơn, bằng cách thay thế những con số vô cảm bởi những tên mang đầy ý nghĩa hơn.
- Hằng làm cho chương trình dễ chỉnh sửa hơn.
- Hằng làm cho việc tránh lỗi dễ dàng hơn, nếu chúng ta gán một trị khác cho một hằng đâu đó trong chương trình sau khi chúng ta đã gán giá trị cho hằng, thì trình biên dịch sẽ thông báo lỗi.

2.4 Kiểu dữ liệu cơ bản

C# là một ngôn ngữ được kiểm soát chặt chẽ về mặt kiểu dữ liệu, C# chia kiểu dữ liệu thành hai loại: **kiểu giá trị** (value type) và **kiểu tham chiếu** (reference type). Nghĩa là trong một chương trình C# dữ liệu được lưu trữ một trong hai chỗ khác nhau tùy theo đặc thù của kiểu dữ liệu.

- Chỗ thứ nhất là **stack**: đây là một vùng bộ nhớ dành để lưu trữ dữ liệu với chiều dài cố định, chẳng hạn một số nguyên sẽ chiếm dụng 4 bytes trong stack. Mỗi chương trình khi thực thi đều được cấp phát riêng một stack mà các chương trình khác không được phép truy cập tới. Khi một hàm được gọi thực thi thì tất cả các biến cục bộ của hàm được đưa vào trong stack và sau khi gọi hàm hoàn thành thì những biến cục bộ của hàm đều bị đẩy ra khỏi stack.
- Chỗ thứ hai là **heap**: một vùng bộ nhớ dùng để lưu trữ dữ liệu có dung lượng thay đổi như kiểu chuỗi chẳng hạn, hoặc dữ liệu có thời gian sống dài hơn phương thức của một đối tượng. Chẳng hạn, khi tạo thể hiện của một lớp, đối tượng được lưu trữ trên heap, và nó không bị tổng ra khi hàm hoàn thành giống như stack, mà ở nguyên tại chỗ và có thể trao cho các phương thức khác thông qua một tham chiếu. Trên C#, heap này được gọi là **managed heap**, và bộ dọn rác (garbage collector) sẽ lo thu hồi những vùng nhớ trong heap không còn sử dụng (không còn được tham chiếu đến).

a) Kiểu giá trị định nghĩa trước

Kiểu số nguyên(integer):

C# hỗ trợ 8 kiểu dữ liệu số nguyên sau:

Tên	Kiểu	Mô tả	Miền giá trị
sbyte	System.SByte	Số nguyên có dấu 8-bit	-128:127 ($-2^7:2^7-1$)
short	System.Int16	Số nguyên có dấu 16-bit	-32,768:32,767 ($-2^{15}:2^{15}-1$)
int	System.Int32	Số nguyên có dấu 32-bit	-2,147,483,648:2,147,483,647 ($-2^{31}:2^{31}-1$)
long	System.Int64	Số nguyên có dấu 64-bit	-9,223,372,036,854,775,808:9,223,372,036,854,775,807 ($-2^{63}:2^{63}-1$)

byte	System.Byte	Số nguyên không dấu 8-bit	0:255 ($0:2^8-1$)
ushort	System.UInt16	Số nguyên không dấu 16-bit	0:65, 35 ($0:2^{16}-1$)
uint	System.UInt32	Số nguyên không dấu 32-bit	0:4,294,967,295 ($0:2^{32}-1$)
ulong	System.UInt64	Số nguyên không dấu 64-bit	0:18,446,744,073,709,551,615 ($0:2^{64}-1$)

Ví dụ khai báo các biến kiểu nguyên:

```
long x = 0x12ab; // ghi theo dạng thập lục phân
uint ui = 1234U;
long l = 1234L;
ulong ul = 1234UL;
```

Kiểu dữ liệu số thực dấu chấm di động (Floating Point Types):

Tên	Kiểu	Mô tả	Miền giá trị
Float	System.Single	32-bit	$\pm 1.5 \times 10^{-45}$ đến $\pm 3.4 \times 10^{38}$
Double	System.Double	64-bit	$\pm 5.0 \times 10^{-324}$ đến $\pm 1.7 \times 10^{308}$

Ví dụ khai báo một biến kiểu thực:

```
float f = 12.3F;
```

Kiểu dữ liệu số thập phân (Decimal Type):

Tên	Kiểu	Mô tả	Miền giá trị
decimal	System.Decimal	128-bit	$\pm 1.0 \times 10^{-28}$ đến $\pm 7.9 \times 10^{28}$

Ví dụ khai báo biến kiểu thập phân:

```
decimal d = 12.30M; //có thể viết decimal d = 12.30m;
```

Kiểu Boolean:

Tên	Kiểu	Giá trị
Bool	System.Boolean	true hoặc false

Kiểu kí tự:

Tên	Kiểu	Giá trị
char	System.Char	Dùng unicode 16 bit

b) Kiểu tham chiếu định nghĩa trước:

C# hỗ trợ hai kiểu tham chiếu được định nghĩa trước:

Tên	Kiểu	Mô tả
object	System.Object	Kiểu cha của tất cả các kiểu trong CLR
string	System.String	Chuỗi kí tự unicode

Các ký tự escape thông dụng:

Thứ tự	Kí tự
\'	Nháy đơn
\"	Nháy kép
\\	Dấu xuyệt
\0	Null
\a	Cảnh báo
\b	Phím lui
\f	Form feed
\n	Xuống hàng

Thứ tự	Kí tự
\r	Xuống hàng
\t	Tab
\v	Tab dọc

2.5 Xuất nhập qua Console

a) Xuất dữ liệu

Ứng dụng dòng lệnh là ứng dụng không có giao diện người dùng đồ họa. Việc xuất nhập phải thông qua dòng lệnh chuẩn. Phương thức `Console.WriteLine()` trong ví dụ 1 viết chuỗi “Chương Trình Dau Tien” lên màn hình. Màn hình được quản lý bởi một đối tượng tên `Console`. Đối tượng này có một phương thức `WriteLine()` nhận một chuỗi và xuất chúng ra thiết bị xuất chuẩn (màn hình).

Hai phương thức dùng để xuất chuỗi ký tự:

- `Console.Write()` - Viết một chuỗi (giá trị) ra cửa sổ.
- `Console.WriteLine()` - tương tự như trên nhưng con trỏ sẽ tự động xuống hàng khi kết thúc lệnh.

Ví dụ 2.2: Xuất dữ liệu kiểu chuỗi

```

1: using System;
2: namespace UngDungDauTien
3: {
4:     class Program
5:     {
6:         static void Main(string[] args)
7:         {
8:             //Khai báo biến s kiểu string và khởi gán giá trị
9:             string s = "Lap trinh huong doi tuong";
10:            Console.WriteLine(s);
11:            string h = " Khoa CNTT DHDL";
12:            Console.WriteLine(s + h);
13:            Console.WriteLine("{0}{1} ", s, h);
14:            Console.WriteLine("S = {0}, H= {1} ", s, h);
15:        }
16:    }

```

Trong ví dụ 2.2 chúng ta thấy có nhiều cách khác nhau khi thực hiện phương thức `Console.WriteLine`, sử dụng phương thức này chúng ta có thể xuất giá trị của các biến ra màn hình. Sau khi thực hiện phương thức `Console.WriteLine`, dấu nhắc lệnh sẽ xuống hàng. Ta có thể thực hiện xuống dòng bằng cách xuất kí tự xuống dòng sau:

```
Console.Write(s + h + "\n");
```

Chúng ta có thể dùng phương thức `Console.WriteLine` để xuất giá trị của biến bất kì .

Ví dụ 2.3: Xuất dữ liệu kiểu số

```
1: using System;
2: namespace UngDungDauTien
3: {
4:     class Program
5:     {
6:         static void Main(string[] args)
7:         {
8:             int a = 10, b=20;
9:             Console.WriteLine(a);
10:            Console.WriteLine(b);
11:            Console.WriteLine("Gia tri cua a la " + a + " b la " + b);
12:            Console.WriteLine("Gia tri cua a la {0} b la {1} ", a, b);
13:        }
14:    }
15: }
```

Kết quả chương trình là:

```
10
20
Gia tri cua a la 10 b la 20
Gia tri cua a la 10 b la 20
```

b) Nhập dữ liệu

Để đọc một ký tự văn bản từ cửa sổ Console, chúng ta dùng phương thức:

- `Console.Read()`: giá trị trả về sẽ là kiểu `int` hoặc kiểu `string`.

Ví dụ sau sẽ cho giá trị nhập kiểu `int` và giá trị xuất ra kiểu chuỗi

```
int x = Console.Read();
Console.WriteLine((char)x);
```

Ví dụ sau nhập vào một chuỗi và gán cho một biến

```
string s = Console.ReadLine();
Console.WriteLine(s);
```

2.6 Biểu thức

Trong phần trước chúng ta đã biết cách khai báo và khởi tạo giá trị cho các biến, khi lập trình chúng ta cần thao tác trên các biến này. C# chứa một số các toán tử cho việc thao tác này. Có 3 loại toán tử:

- Một ngôi: dùng một toán hạng
- Hai ngôi: dùng hai toán hạng
- Ba ngôi: dùng ba toán hạng

a) Toán tử toán học

Toán tử	Loại	Biểu thức ví dụ	Kết quả
+	Hai ngôi	var1 = var2 + var3;	Giá trị var1 được gán bằng tổng của var2 và var3
-	Hai ngôi	var1 = var2 - var3;	Giá trị var1 được gán bằng var2 trừ var3
*	Hai ngôi	var1 = var2 * var3;	Giá trị var1 được gán bằng tích của var2 và var3
/	Hai ngôi	var1 = var2 / var3;	Giá trị var1 được gán bằng thương của var2 và var3
+	Một ngôi	var1 =+ var2;	var1 được gán giá trị của var2
-	Một ngôi	var1 =- var2;	var1 được gán giá trị của var2 nhân -1
++	Một ngôi	var1 = ++var2;	var1 được gán bằng giá trị var2 + 1. var2 tăng 1
--	Một ngôi	var1 = --var2;	var1 được gán bằng giá trị var2 - 1. var2 giảm 1

++	Một ngôi	var1 = var2++;	var1 được gán bằng giá trị var2. var2 tăng 1
--	Một ngôi	var1 = var2--;	var1 được gán bằng giá trị var2. var2 giảm 1

Ví dụ 4: Ví dụ về nhập xuất dữ liệu và sử dụng toán tử

```

1: using System;
2: namespace UngDungDauTien
3: {
4:     class Program
5:     {
6:         static void Main(string[] args)
7:         {
8:             double n1, n2;
9:             string t;
10:            Console.WriteLine("Nhap vao ten ");
11:            t = Console.ReadLine();
12:            Console.WriteLine("Nhap so thu 1 ");
13:            n1 = double.Parse(Console.ReadLine());
14:            Console.WriteLine("Nhap so thu 2 ");
15:            n2 = double.Parse(Console.ReadLine());
16:            Console.WriteLine("Tong cua {0} va {1} la {2} ", n1, n2, n1 + n2);
17:            Console.WriteLine("Hieu cua {0} va {1} la {2} ", n1, n2, n1 - n2);
18:            Console.WriteLine("Nhan cua {0} va {1} la {2} ", n1, n2, n1 * n2);
19:            Console.WriteLine("Chia cua {0} va {1} la {2} ", n1, n2, n1 / n2);
20:        }
21:    }
22: }

```

Kết quả khi chạy chương trình

Nhap vao ten

OOP

Nhap so thu 1

10.5

Nhap so thu 2

2.5

Tong cua 10.5 va 2.5 la 13

Hieu cua 10.5 va 2.5 la 8

Nhan cua 10.5 va 2.5 la 26.25

Chia cua 10.5 va 2.5 la 4.2

Trong dòng 13, 15 chúng ta dùng phương thức `double.Parse` để chuyển đổi kiểu dữ liệu từ string sang kiểu double. Chúng ta cũng có thể dùng phương thức `Convert.ToDouble(string s)` để chuyển đổi kiểu dữ liệu string sang kiểu double.

b) Toán tử gán

Toán tử	Loại	Biểu thức ví dụ	Kết quả
=	Hai ngôi	var1 = var2;	Giá trị var1 được gán bằng giá trị của var2.
+=	Hai ngôi	var1 += var2;	Giá trị var1 được gán bằng giá trị của var2 cộng var1.
-=	Hai ngôi	var1 -= var2;	Giá trị var1 được gán bằng giá trị của var2 trừ var1.
*=	Hai ngôi	var1 *= var2;	Giá trị var1 được gán bằng giá trị của var2 nhân var1.
/=	Hai ngôi	var1 /= var2;	Giá trị var1 được gán bằng giá trị của var2 chia var1.

2.7 Chuyển đổi kiểu

Khi thực hiện một lệnh gán trong chương trình `var1=var2;` thì hai vế của lệnh gán phải có cùng kiểu dữ liệu. Ngược lại có thể gây ra lỗi.

Ví dụ 2.5: Chuyển đổi kiểu

```

1: using System;
2: namespace ChuyenDoiKieuKhongTuongMinh
3: {
4:     class Program
5:     {
6:         static void Main(string[] args)
7:         {
8:             int a = int.MaxValue; //lay gia tri int lon nhat
9:             long b;
10:            b = a;
11:            Console.WriteLine("Gia tri cua b la " + b);
12:        }
13:    }
14: }
```

Kết quả của chương trình là:

Gia tri cua b la 2147483647

Trong câu lệnh 10 chúng ta thấy hai vế của lệnh gán không cùng kiểu dữ liệu (b kiểu long (64 bit), a kiểu int (32 bit)) nhưng chương trình vẫn được phép thực hiện. Vì sao?

a) Chuyển đổi kiểu không tường minh

Trong ví dụ 2.5 dòng 10, trình biên dịch tự động chuyển kiểu từ int sang long. Vì kích thước của kiểu int (32 bit) nhỏ hơn kiểu long (64 bit) nên luôn đảm bảo phép gán trong chương trình luôn thực hiện đúng. Đây chính là cách ép kiểu không tường minh được thực hiện ngầm định bởi hệ thống.

c) Chuyển đổi kiểu không tường minh

Ví dụ 2.6: Chuyển đổi kiểu tường minh

```
1: using System;
2: namespace ChuyenDoiKieuTuongMinh
3: {
4:     class Program
5:     {
6:         static void Main(string[] args)
7:         {
8:             long a = long.MaxValue; //lay gia tri long lon nhat
9:             int b;
10:            b = a;
11:            Console.WriteLine("Gia tri cua b la " + b);
12:        }
13:    }
14: }
```

Khi thực hiện chương trình trên, trình biên dịch sẽ thông báo lỗi dòng 10 vì chương trình không thể thực hiện ép một kiểu dữ liệu có kích thước lớn sang kiểu có kích thước nhỏ vì có thể gây ra hiện tượng tràn số dẫn đến kết quả sai. Nếu muốn ép kiểu chúng ta phải khai báo tường minh như trong ví dụ 2.7 nhưng kết quả thực hiện không như mong muốn.

Ví dụ 2.7: Kết quả sai do hiện tượng tràn số

```
1: using System;
2: namespace UngDungDauTien
3: {
4:     class Program
5:     {
6:         static void Main(string[] args)
7:         {
8:             long a = long.MaxValue; //lay gia tri long lon nhat
9:             int b;
10:            b = (int)a;
11:            Console.WriteLine("Gia tri cua a la " + a);
12:            Console.WriteLine("Gia tri cua b la " + b);
13:        }
14:    }
15: }
```

Kết quả thực hiện chương trình

Gia tri cua a la 9223372036854775807

Gia tri cua b la -1

Chúng ta thấy kết quả chương trình xuất giá trị b là -1 (giá trị sai) do tràn số. Do đó khi thực hiện chuyển đổi kiểu tương minh chúng ta cần chú ý đến vấn đề tràn số gây ra các hiệu ứng phụ ảnh hưởng đến kết quả chương trình.

2.8 Kiểu liệt kê

Kiểu này bổ sung những tính năng mới thuận tiện hơn kiểu hằng. Kiểu liệt kê là một kiểu giá trị phân biệt bao gồm một tập các tên hằng. Ví dụ chúng ta tạo hai hằng liên quan nhau:

```
const int NhietDoDongLanh = 32; // độ Fahrenheit
const int NhietDoSoi = 212;
```

Chúng ta có thể bổ sung một số hằng khác vào trong danh sách như:

```
const int NhietDoCoTheBoi= 72; // độ Fahrenheit
```

Quá trình này thực hiện rất cồng kềnh. Do đó, chúng ta có thể dùng kiểu liệt kê để giải quyết vấn đề:

```
enum NhietDo
{
```

```
NhietDoDongLanh = 32;
NhietDoSoi = 212;
NhietDoCoTheBoi= 72;
}
```

Mỗi kiểu liệt kê phải có một kiểu bên dưới, chúng có thể là (int, short, long. ..) ngoại trừ kiểu char, mặc định là kiểu int.

Ví dụ sau minh họa dùng kiểu enum để định nghĩa một thực đơn cho chương trình:

```
enum Menu
{
    Thoat=5,
    VeTamGiac,
    VeTamGiacLatNguoc,
    VeTamGiacGiuaManHinh,
    VeTamGiacRong
};
```

Khi chúng ta không gán giá trị cụ thể cho các biến trong enum, thì giá trị của nó sẽ bằng giá trị của biến trước cộng 1. Giá trị của biến đầu tiên mặc định bằng không.

Ví dụ 2.8: Khai báo và sử dụng kiểu enum

```
1:  class Program
2:  {
3:      enum Mau
4:      {
5:          Den,
6:          Trang,
7:          Do
8:      }
9:      static void Main(string[] args)
10:     {
11:         Mau m = Mau.Trang;
12:         Console.WriteLine("Gia tri mau {0} la {1}", Mau.Den,
13:             (int)Mau.Den);
14:         Console.WriteLine("Gia tri mau {0} la {1}", m, (int)m);
15:         Console.WriteLine("Gia tri mau {0} la {1}", Mau.Do,
16:             (int)Mau.Do);
17:     }
18: }
```

```
Kết quả thực hiện chương trình
Gia tri mau Den la 0
Gia tri mau Trang la 1
Gia tri mau Do la 2
```

2.9 Mảng

Mảng là một cấu trúc dữ liệu cấu tạo bởi một số biến được gọi là những phần tử mảng. Tất cả các phần tử này đều thuộc một kiểu dữ liệu. Chúng ta có thể truy xuất phần tử thông qua chỉ số. Chỉ số bắt đầu bằng 0.

Có nhiều loại mảng (array): mảng một chiều, mảng nhiều chiều...

Cú pháp:

```
Kiểu_dữ_liệu[ ] tên_mảng;
```

Ví dụ:

```
int[] myIntegers; // mảng kiểu số nguyên
```

```
string[] myString ; // mảng kiểu chuỗi
```

Chúng ta khai báo mảng có chiều dài xác định với từ khoá **new** như sau:

```
int[] integers = new int[32];
```

```
integers[0] = 35; // phần tử đầu tiên có giá trị 35
```

```
integers[31] = 432; // phần tử 32 có giá trị 432
```

```
//chúng ta cũng có thể khai báo như sau:
```

```
int[] integers;
```

```
integers = new int[32];
```

```
//khai báo mảng và khởi gán giá trị cho các phần tử trong mảng
```

```
string[] myArray = { "phần tử 1", " phần tử 2", " phần tử 3" };
```

2.10 Không gian tên (Namespace)

Namespace cung cấp cho ta cách tổ chức quan hệ giữa các lớp và các kiểu khác nhau.

Đây là kỹ thuật cho phép .NET tránh việc các tên lớp, tên biến, tên hàm... đụng độ nhau vì trùng tên giữa các lớp.

Để khai báo không gian tên chúng ta sử dụng từ khóa namespace. Ví dụ:

```
1: namespace LapTrinhOOP
2: {
3:     using System;
4:     public class Tester
5:     {
6:         public static int Main( )
7:         {
8:             Console.WriteLine("Su dung namespace");
9:             return 0;
10:        }
11:    }
12: }
```

Ví dụ sau khai báo không gian tên lồng nhau:

```
1: namespace KiThuatLapTrinh
2: {
3:     namespace LapTrinhOOP
4:     {
5:         using System;
6:         public class Tester
7:         {
8:             public static int Main()
9:             {
10:                 Console.WriteLine("Su dung namespace");
11:                 return 0;
12:             }
13:         }
14:     }
15: }
```

2.11 Câu lệnh using

Từ khoá using giúp chúng ta giảm thiểu việc phải gõ những không gian tên trước các hàm hoặc thuộc tính. Ví dụ sau sử dụng Console.WriteLine thay vì phải gõ đầy đủ đường dẫn System.Console.WriteLine.

```
1: using System;
2: class Test
3: {
4:     public static int Main()
5:     {
6:         Console.WriteLine("Khong gian ten ");
7:         return 0;
8:     }
9: }
```

2.12 Phương thức Main()

Khi một ứng dụng dòng lệnh hoặc ứng dụng Windows được biên dịch, theo mặc định trình biên dịch nhìn vào phương thức Main() như là điểm bắt đầu của chương trình. Nếu có nhiều hơn một phương thức Main(), trình biên dịch sẽ trả về thông báo lỗi. Do đó, mọi chương trình C# phải chứa một phương thức Main().

2.13 Sử dụng chú thích

Như chúng ta đã lưu ý lúc đầu, C# sử dụng kiểu truyền thống của C cho việc chú thích. Dùng // cho hàng đơn và /*...*/ cho một khối lệnh. Các đoạn mã lệnh trong chương trình C# cũng có thể chứa những dòng chú thích. Ví dụ:

```
// Chú thích trên 1 dòng
/* Chú thích
   trên 2 dòng */
Console.WriteLine(/* Kiểm tra chú thích! */ "Biên dịch bình thường");
GoiPhuongThuc(Rong, /*Cao*/ 100);
string s = "/* Đây là chuỗi không phải chú thích*/";
```

2.14 Chỉ dẫn tiền xử lý trong C#

Từ định danh là tên dùng cho việc đặt tên biến cũng như để định nghĩa kiểu sử dụng như các lớp, cấu trúc, và các thành phần của kiểu này. C# có một số quy tắc để định rõ các từ định danh như sau:

- Chúng phải bắt đầu bằng ký tự không bị gạch dưới.
- Chúng ta không được sử dụng từ khoá làm từ định danh.

Trong C# có sẵn một số từ khoá (keyword):

abstract	do	implicit	params	switch
as	double	in	private	this
base	else	int	protected	throw
bool	enum	interface	public	true

break	event	internal	readonly	try
byte	explicit	is	ref	typeof
case	extern	lock	return	uint
catch	false	long	sbyte	ulong
char	finally	namespace	sealed	unchecked
checked	fixed	new	short	unsafe
class	float	null	sizeof	ushort
const	for	object	stackalloc	using
continue	foreach	operator	static	virtual
decimal	goto	out	string	volatile
default	if	override	struct	void
delegate				while

Chương 3: Cấu trúc điều khiển

Mục đích của chương:

- Cách sử dụng các lệnh rẽ nhánh.
- Cách sử dụng các lệnh lặp.
- Sử dụng hàm trong chương trình.
- Các thực hiện dò lỗi trong chương trình.
- Xây dựng chương trình hoàn chỉnh thực hiện các chức năng trên mảng một chiều.

3.1 Lệnh rẽ nhánh if

Lệnh if là một lệnh rất linh hoạt và hữu ích khi thực hiện các lệnh có tính chất quyết định trong chương trình. Cú pháp của lệnh như sau:

Cú pháp:

```
if (biểu_thức)
    Lệnh 1
else
    Lệnh 2
hay
if (biểu_thức)
{
    Lệnh 1
}
else
{
    Lệnh 2
}
```

Sử dụng lệnh if đơn giản như sau, nếu biểu_thức trả về giá trị đúng (true) thì dòng lệnh 1 (hay khối lệnh) sẽ được thực hiện. Ngược lại, lệnh 2 sẽ được thực hiện. Chúng ta có thể thực hiện các khối lệnh if lồng nhau.

Ví dụ 3.1: Sử dụng lệnh rẽ nhánh if

```

1: using System;
2: class Values
3: {
4:     static void Main( )
5:     {
6:         int valueOne = 10;
7:         int valueTwo = 20;
8:         if ( valueOne > valueTwo )
9:         {
10:            Console.WriteLine("Giá trị một: {0} lớn hơn giá trị hai: {1}",
                               valueOne, valueTwo);
11:        }
12:        else
13:        {
14:            Console.WriteLine("Giá trị hai: {0} lớn hơn giá trị một: {1}",
                               valueTwo, valueOne);
15:        }
16:        valueOne = 30; //gán lại giá trị mới
17:        if ( valueOne > valueTwo )
18:        {
19:            valueTwo = valueOne++;
20:            Console.WriteLine("\nGán giá trị một cho giá trị hai, ");
21:            Console.WriteLine("và tăng giá trị một lên hai. \n");
22:            Console.WriteLine("Giá trị một: {0}, Giá trị hai: {1}",
                               valueOne, valueTwo);
23:        }
24:        else
25:        {
26:            valueOne = valueTwo;
27:            Console.WriteLine("Gán hai giá trị bằng nhau và bằng giá trị
                               hai. ");
28:            Console.WriteLine("Giá trị một: {0} giá trị hai: {1}",
                               valueOne, valueTwo);
29:        }
30:    }
31: }
32: }

```

3.2 Lệnh switch

Lệnh switch thì tương tự lệnh if, việc thực thi các lệnh dựa trên giá trị của biến `biến_kiểm_tra`. Tuy nhiên, lệnh switch cho phép chúng ta kiểm tra nhiều giá trị của biến `biến_kiểm_tra`. Khi giá trị của biến kiểm tra khớp với một `giá_trị_so_sánh` nào đó thì lệnh (khối lệnh) trong trường hợp đó sẽ được thực hiện. Nếu không có giá trị nào khớp thì các lệnh trong phần khai báo default sẽ được thực hiện.

Cú pháp:

```

switch (biến_kiểm_tra)
{
    case giá_trị_so_sánh_1:

```

```
        lệnh_thực_hiện;
        break;
    case giá_trị_so_sánh_2:
        lệnh_thực_hiện;
        break;
    [default: lệnh]
}
```

Ví dụ 3.2: Ví dụ minh họa sử dụng lệnh switch:

```
1:  class Values
2:  {
3:      static void Main( )
4:      {
5:          int chonThucDon;
6:          chonThucDon = 2;
7:          switch (chonThucDon)
8:          {
9:              case 1:
10:                 System.Console.WriteLine(" Chọn món 1");
11:                 break;
12:              case 2:
13:                 System.Console.WriteLine(" Chọn món 1");
14:                 break;
15:              default:
16:                 System.Console.WriteLine(" Phải chọn món có trong thực
                                     đơn 1");
17:                 break;
18:          }
19:      }
20: }
```

3.3 Lệnh lặp

a) Lệnh goto

C# cho phép chúng ta đặt nhãn cho các dòng lệnh và nhảy trực tiếp đến các lệnh này dùng lệnh goto. Thuận lợi của lệnh này là cách đơn giản để kiểm soát lệnh nào được thực thi. Tuy nhiên trong một số trường hợp có thể gây khó hiểu khi đọc mã. Lệnh goto được sử dụng như sau:

```
goto <tên_nhãn>;
```

Tên nhãn được định nghĩa như sau:

```
<tên_nhãn>:
```

Ví dụ 3.3: Ví dụ minh họa sử dụng lệnh goto

```
1: using System;
2: public class Tester
3: {
4:     public static int Main( )
5:     {
6:         int i = 0;
7:         repeat: // gán nhãn cho lệnh goto
8:             Console.WriteLine("i: {0}", i);
9:             i++;
10:            if (i < 10)
11:                goto repeat;
12:            return 0;
13:    }
14: }
```

Kết quả thực hiện của chương trình trên sẽ xuất giá trị i từ 0 đến 9 ra màn hình.

Cấu trúc lặp chỉ ra việc thực thi các lệnh được lặp lại nhiều lần. Kỹ thuật này rất thuận tiện vì chúng ta có thể lặp lại các thao tác nào đó nhiều lần (hàng ngàn, thậm chí hàng triệu) mà không cần viết lại mã mỗi lần. Ví dụ, để tính số tiền của một tài khoản nào đó trong ngân hàng sau 10 năm, giả sử lãi suất giống nhau giữa các năm và không có tiền nạp thêm vào tài khoản. Chúng ta có thể viết đoạn chương trình tính toán như sau:

```
1: double so_tien = 1000;
2: double lai_suat = 1.05; // lãi suất 5% trên 1 năm
3: so_tien *= lai_suat;
4: so_tien *= lai_suat;
5: so_tien *= lai_suat;
6: so_tien *= lai_suat;
7: so_tien *= lai_suat;
8: so_tien *= lai_suat;
9: so_tien *= lai_suat;
10: so_tien *= lai_suat;
11: so_tien *= lai_suat;
12: so_tien *= lai_suat;
```

Chúng ta thấy các đoạn mã trong chương trình từ dòng 3 đến dòng 12 dường như lãng phí vì trùng lặp nhiều lần. Điều gì sẽ xảy ra khi chúng ta cần thay đổi số năm cần tính là 4 hay 40 năm? Với việc sử dụng cấu trúc lặp chúng ta có thể dễ dàng thực hiện các yêu cầu trên mà không cần thay đổi mã của chương trình.

b) Lệnh lặp do

Mã lệnh của chúng ta trong vòng lặp được thực hiện. Sau đó giá trị của biểu thức biểu_thức sẽ được kiểm tra. Nếu nó trả về giá trị true, mã lệnh trong vòng lặp được tiếp tục thực hiện; ngược lại, sẽ thoát ra khỏi vòng lặp.

Cú pháp:

do

{

 <các lệnh sẽ bị Lặp>

} **while** (<biểu_thức>);

Sử dụng vòng lặp do cho ví dụ trên, ta có thể viết ngắn gọn đoạn chương trình như sau:

```
1: double so_tien = 1000;
2: double lai_suat = 1.05; // lãi suất 5% trên 1 năm
3: int nam = 1;
4: do
5: {
6:     so_tien *= lai_suat;
7:     nam++;
8: }
9: while (nam <= 10);
```

c) Lệnh lặp while

Lệnh lặp while tương tự như lệnh lặp do nhưng có một khác biệt quan trọng là giá trị của biểu_thức được kiểm tra trước khi thực hiện các lệnh trong khối lệnh lặp. Nếu giá trị ban đầu của biểu_thức là false thì các lệnh trong khối lệnh lặp không bao giờ được thực hiện.

Cú pháp:

while (<biểu_thức>)

{

 <các Lệnh sẽ bị Lặp>

}

Ví dụ dùng lệnh while

```
1: double so_tien = 1000;
2: double lai_suat = 1.05; // lãi suất 5% trên 1 năm
3: int i = 1;
4: while (i <= 10)
5: {
6:     so_tien *= lai_suat;
7:     i++;
8: }
```

d) Lệnh for

Lệnh for thường thực hiện lặp theo một số lần nào đó và phải quản lý số lần lặp theo một biến đếm nào đó. Để định nghĩa lệnh lặp for chúng ta cần các thông tin sau:

- Giá trị khởi tạo cho biến đếm (biến quản lý số lần lặp).
- Điều kiện thực hiện vòng lặp (thường liên quan đến biến đếm).
- Thao tác thực hiện trên giá trị của biến đếm khi kết thúc mỗi chu kì vòng lặp

Cú pháp:

```
for (khởi_tạo; biểu_thức; thao_tác)
{
    Lệnh;
}
```

Lệnh for trên tương đương với lệnh lặp while như sau:

```
khởi_tạo;
```

```
while (biểu_thức)
```

```
{
    lệnh;
    thao_tác;
}
```

Ví dụ dùng lệnh for:

```
1: double so_tien = 1000;
2: double lai_suat = 1.05; // lãi suất 5% trên 1 năm
```

```
3:  for(int i = 1; i <= 10; i++)
4:  {
5:      so_tien *= lai_suat;
6:  }
```

e) Lệnh continue và break

Thỉnh thoảng chúng ta muốn kiểm soát việc xử lý các mã trong vòng lặp chúng ta có thể sử dụng các lệnh sau đây:

- Lệnh continue: cho phép quay lại vòng lặp mà không cần thực hiện các lệnh còn lại trong vòng lặp
- Lệnh break: cho phép thoát ra khỏi vòng lặp ngay lập tức;
- Lệnh goto: cho phép nhảy đến một nhãn nào đó ngoài vòng lặp.
- Lệnh return: nhảy ra khỏi vòng lặp và cả hàm chứa vòng lặp.

Ví dụ 3.4: Sử dụng cấu trúc rẽ nhánh và lặp

Chương trình sau cho phép nhập vào một kí hiệu. Nếu kí hiệu vừa nhập là “A” thì thông báo lỗi và thoát khỏi vòng lặp while. Nếu kí hiệu là “0” thì thông báo bình thường và thực hiện lại bắt đầu vòng lặp. Nếu kí tự không phải là “A” hoặc “0” thì thực hiện vòng lặp bình thường.

```
1:  using System;
2:  public class Tester
3:  {
4:      public static int Main( )
5:      {
6:          string signal = "0";
7:          while (signal != "X")
8:          {
9:              Console.Ghi("Nhập vào tín hiệu: ");
10:             signal = Console.ReadLine( );
11:             Console.WriteLine("Tín hiệu vừa nhập: {0}", signal);
12:             if (signal == "A")
13:             {
```

```
14:         Console.WriteLine("Lỗi, bỏ qua\n");
15:         break;
16:     }
17:     if (signal == "0")
18:     {
19:         Console.WriteLine("Bình thường. \n");
20:         continue;
21:     }
22:     Console.WriteLine("{0}Tạo tín hiệu tiếp tục !\n", signal);
23: }
24: return 0;
25: }
26: }
```

3.4 Hàm

Hàm cung cấp cách thức cho phép chúng ta nhóm các khối mã thực hiện một công việc nào đó vào trong một hàm và có thể gọi thực thi tại bất cứ điểm nào trong chương trình. Hàm cho phép thực thi các thao tác giống nhau với các dữ liệu khác nhau. Ví dụ chúng ta có thể viết hàm giải phương trình bậc 2; với hàm này chúng ta có thể giải phương trình bậc với giá trị của các hệ số a, b, c là bất kì.

Ưu điểm của việc sử dụng hàm:

- Cho phép chúng ta tái sử dụng lại mã của chương trình.
- Làm cho chương trình dễ đọc hơn bằng cách nhóm các mã liên quan lại với nhau giúp chương trình ngắn gọn
- Dùng để tạo các đoạn mã đa mục đích, cho phép thực hiện cùng các thao tác nhưng với nhiều dữ liệu khác nhau.

Cú pháp:

```
bổ_từ kiểu_trả_về tên_hàm(tham_số_1, tham_số_2,...)
{
    ...
    return giá_trị_trả_về;
}
```

bổ_từ được giải thích ở các chương sau.

a) Kiểu trả về

Cách đơn giản nhất để trao đổi dữ liệu với một hàm là sử dụng giá trị trả về. Kiểu trả về có thể là các kiểu dữ liệu cơ bản, kiểu mảng hay bất kì kiểu dữ liệu nào do người dùng định nghĩa. Nếu một hàm không có kiểu trả về thì chúng ta sử dụng từ khóa void. Nếu một hàm có khai báo kiểu trả về thì khi thực hiện hàm cần phải trả về giá trị có kiểu dữ liệu trùng với kiểu khai báo của kiểu trả về. Chúng ta có thể trả về giá trị thông qua lệnh return. Ví dụ ta có thể định nghĩa LayGiaTri như sau:

```
int LayGiaTri()
{
    return 10;
}
```

b) Tham số của hàm

Khi định nghĩa hàm chúng ta có thể khai báo các tham số cần thiết để truyền dữ liệu cho hàm hay lấy giá trị trả về từ hàm thông qua tham số, một hàm có thể không có tham số nào. Ví dụ khai báo hàm giải phương trình bậc hai như sau:

```
void GiaiPTBac2(float a, float b, float c)
{
    ...
}
```

c) Cách gọi hàm dùng tham trị

Khi viết hàm, chúng ta muốn sau khi thực hiện hàm giá trị của tham số truyền vào không bị thay đổi, muốn thực hiện điều này chúng ta dùng các gọi hàm theo tham trị.

Ví dụ 3.4: Sử dụng tham trị

```
1:  class Program
2:  {
3:      static void Main(string[] args)
4:      {
5:          int x = 1;
6:          int y = 2;
7:          KiemTra(x, y);
8:          Console.WriteLine("x = {0}, y = {1}", x, y);
9:          Console.ReadKey();
10:     }
11:     static void KiemTra(int a, int b)
12:     {
13:         a = 10;
```

```
14:         b = 20;
15:     }
16: }
```

Trong chương trình trên, kết quả khi thực hiện chương trình là $x=1$, $y=2$ vì hàm KiemTra sử dụng hai tham số kiểu `int` được khai báo là tham trị, do đó khi kết thúc thực hiện hàm giá trị của biến truyền vào x , y không bị thay đổi. Nếu muốn thay đổi ta phải truyền tham biến.

d) Cách gọi hàm dùng tham biến

Để thực hiện khai báo hàm cho phép truyền tham biến trên các tham số nào đó thì tham số này phải được khai báo với từ khóa `ref`.

Ví dụ 3.5: Sử dụng tham biến

```
1:  class Program
2:  {
3:      static void Main(string[] args)
4:      {
5:          int x = 1;
6:          int y = 2;
7:          KiemTra(x, ref y);
8:          Console.WriteLine("x = {0}, y = {1}", x, y);
9:          Console.ReadKey();
10:     }
11:     static void KiemTra(int a, ref int b)
12:     {
13:         a = 10;
14:         b = 20;
15:     }
16: }
```

Kết quả thực hiện chương trình là $x=1$, $y=20$ vì khi thực hiện gọi hàm KiemTra, biến x truyền theo dạng tham trị nên giá trị không thay đổi; biến y truyền tham biến nên giá trị bị thay đổi thành 20.

Chúng ta có thể thay thế từ khóa `ref` bằng từ khóa `out`, sự khác biệt giữa hai cách sử dụng này là khi dùng từ khóa `out` biến không cần phải khởi gán trước khi sử dụng.

Ví dụ 3.6: Sử dụng tham biến kiểu out

```
1:  class Program
2:  {
3:      static void Main(string[] args)
4:      {
```

```
5:         int x = 1;
6:         int y ;
7:         KiemTra(x, out y);
8:         Console.WriteLine("x = {0}, y = {1}", x, y);
9:         Console.ReadKey();
10:    }
11:    static void KiemTra(int a, out int b)
12:    {
13:        a = 10;
14:        b = 20;
15:    }
16: }
```

Trong ví dụ trên biến y không cần phải khởi tạo trước khi sử dụng. Nếu chúng ta dùng từ khóa ref thì chương trình sẽ thông báo lỗi.

3.5 Ứng dụng minh họa

Việc sử dụng thành thạo các cấu trúc điều khiển (rẽ nhánh và lặp) cũng như tổ chức hàm, giúp cho chúng ta xây dựng chương trình một cách nhanh chóng và hiệu quả. Để sinh viên có thể hiểu rõ các cấu trúc dữ liệu trong chương này và các chương trước. Chúng ta cần xây dựng một ứng dụng minh họa có sử dụng các kiểu dữ liệu và cấu trúc dữ liệu với các yêu cầu sau:

- Viết chương trình tính tổng các số nguyên.
- Tìm số nguyên có giá trị lớn nhất.

Chú ý: khi xây dựng một chương trình đầu tiên chúng ta cần quan tâm tới việc nhập, xuất của chương trình. Trong yêu cầu của bài tập trên, trước tiên chúng ta cần quan tâm tới các vấn đề sau:

- Cách khai báo mảng 1 chiều trong C#.
- Cách nhập giá trị cho các phần tử trong mảng 1 chiều.
- Cách xuất các phần tử ra màn hình.

Ví dụ 3.7: Sử dụng mảng 1 chiều

Chúng ta có thể viết chương trình đơn giản như sau:

```
1: using System;
2: namespace Mang1Chieu
```

```
3: {
4:     class Program
5:     {
6:         static void Main(string[] args)
7:         {
8:             int[] a = { 6, 4, 7, -4, -7, 5, 6, 6, 4, 6 };
9:             Console.WriteLine("Cac phan tu trong mang ");
10:            for (int i = 0; i < 10; i++ )
11:            {
12:                Console.WriteLine("a[{0}]=a[{1}]", i, a[i]);
13:            }
14:        }
15:    }
16: }
```

- Dòng 8 khai báo một mảng 1 chiều a với kích thước cố định là 10 phần tử và khởi tạo các giá trị cho các phần tử trong mảng a. Ví dụ a[0]=6, a[8]=4...
- Dòng 9 xuất chuỗi "Cac phan tu trong mang " ra màn hình.
- Từ dòng 10 đến 13, chúng ta sử dụng vòng lặp for để xuất giá trị của 10 phần tử trong mảng ra màn hình.

Để tính tổng các phần tử trong mảng, chúng ta cần duyệt qua các phần tử trong mảng và tính tổng các giá trị này. Với mảng a trong chương trình trên, giá trị tổng sẽ là a[0] + a[2] + .. + a[9].

Thuật toán tính tổng có thể thực hiện như sau:

- Khởi tạo giá trị ban đầu của biến tổng là 0
- Duyệt qua các phần tử trong mảng; ứng với từng phần tử, lấy giá trị của nó cộng dồn với biến tổng và gán lại kết quả cho biến tổng.

Chúng ta có thể thực hiện tính tổng và xuất ra kết quả như sau:

```
int tong = 0;
for (int i = 0; i < 10; i++)
{
    tong += a[i];
}
```

```
}
```

```
Console.WriteLine("Tong la {0} ", tong);
```

Khi bắt đầu thực hiện vòng lặp for; giá trị $i=0$; $tong=0$; khi đó kiểm tra biểu thức $i < 10$ nghĩa là $0 < 10$ luôn đúng nên chương trình sẽ thực hiện lệnh $tong += a[i]$ chính là thực hiện lệnh $tong = tong + a[0] = 0 + 6 = 6$; Kết thúc bước lặp đầu tiên $i=0$ và $tong=6$; khi đó chương trình sẽ thực hiện lệnh $i++$; nghĩa là $i=1$ và thực hiện kiểm tra biểu thức $i < 10$. Vì $i=1$ nên $1 < 10$ luôn đúng nên sẽ thực hiện lệnh $tong += a[i]$ chính là thực hiện lệnh $tong = tong + a[1] = 6 + 4 = 10$; Tương tự thực hiện cho đến khi $i=10$ thì biểu thức $i < 10$ trả về false nên lúc này giá trị của $tong = 33$. Đây chính là tổng giá trị của các phần tử trong mảng a.

Tương tự cho thuật toán tìm phần tử lớn nhất trong mảng, ta có thể viết chương trình hoàn chỉnh trên như sau:

```
1: using System;
2: namespace Mang1Chieu
3: {
4:     class Program
5:     {
6:         static void Main(string[] args)
7:         {
8:             int[] a = { 6, 4, 7, -4, -7, 5, 6, 6, 4, 6 };
9:             Console.WriteLine("Cac phan tu trong mang ");
10:            for (int i = 0; i < 10; i++ )
11:            {
12:                Console.WriteLine("a[{0}]=a[{1}]", i, a[i]);
13:            }
14:            int tong = 0;
15:            for (int i = 0; i < 10; i++)
16:            {
17:                tong += a[i];
18:            }
19:            Console.WriteLine("Tong la {0} ", tong);
20:            int max = a[0];
21:            for (int i = 1; i < 10; i++)
22:            {
```

```
23:             if (max < a[i])
24:                 max = a[i];
25:         }
26:         Console.WriteLine("Phan tu lon nhat {0} ", max);
27:     }
28: }
29: }
```

Từ dòng 20 đến dòng 26 là thuật toán tìm phần tử lớn nhất trong mảng a. Chúng ta thấy thuật toán kết hợp dùng cả cấu trúc lặp và cấu trúc rẽ nhánh trong thuật toán. Các chúng ta sinh viên có thể chạy bằng tay để kiểm tra kết quả hoạt động của thuật toán trên. Ý tưởng của thuật toán trên là giả sử phần tử đầu tiên trong mảng là lớn nhất $\text{max} = a[0] = 6$. Sau đó duyệt qua các phần tử trong mảng nếu phần tử nào trong mảng lớn hơn giá trị max hiện tại thì chúng ta cập nhật lại giá trị của phần tử này cho max. Do đó khi duyệt qua hết các phần tử, giá trị hiện tại của max chính là giá trị lớn nhất của mảng.

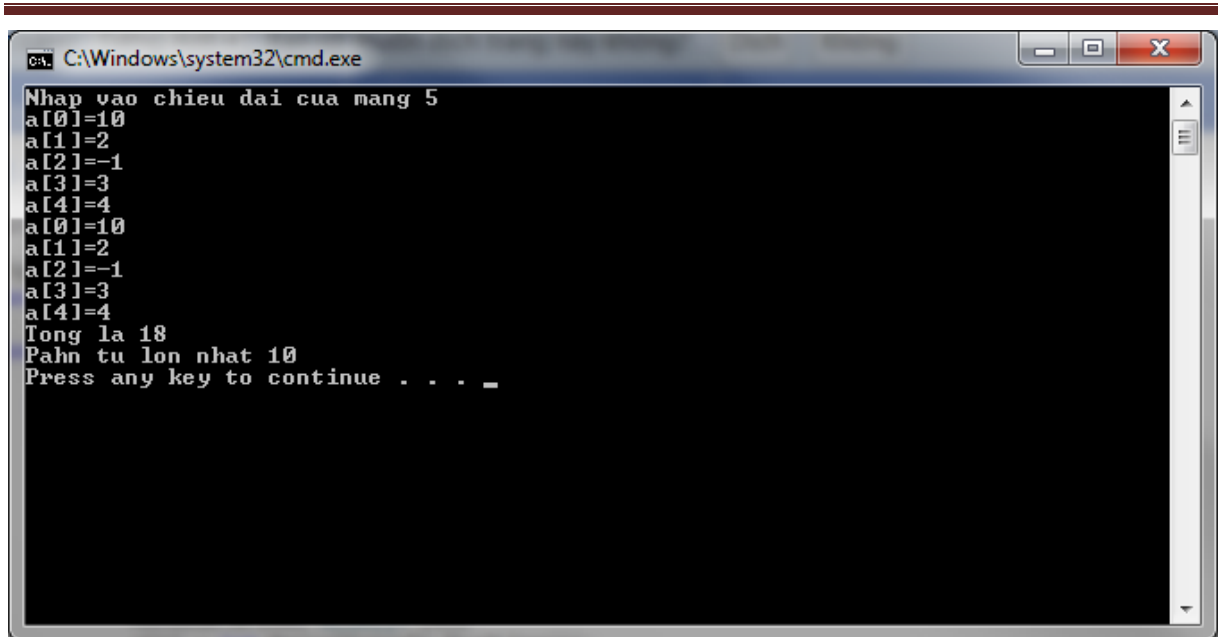
Chú ý: trong môn học này khi viết thuật toán chúng ta chỉ xem xét về kết quả thực hiện đúng của chương trình mà chưa cần quan tâm tới việc tối ưu hay hiệu quả của chương trình.

Trong chương trình trên ta thấy kích thước mảng là cố định. Bây giờ chúng ta muốn chương trình cho phép chúng ta nhập giá trị của mảng a bất kì. Để thực hiện điều này chúng ta có thể viết chương trình như sau:

```
1: using System;
2: namespace Mang1Chieu
3: {
4:     class Program
5:     {
6:         static void Main(string[] args)
7:         {
8:             int[] a = new int[100];
9:             int len = 0;
10:            Console.Write("Nhap vao chieu dai cua mang ");
11:            len = int.Parse(Console.ReadLine());
12:            for (int i = 0; i < len; i++)
```

```
13:         {
14:             Console.Write("a[{0}]=",i);
15:             a[i] = int.Parse(Console.ReadLine());
16:         }
17:         for (int i = 0; i < len; i++ )
18:         {
19:             Console.WriteLine("a[{0}]={1}", i, a[i]);
20:         }
21:         int tong = 0;
22:         for (int i = 0; i < 10; i++)
23:         {
24:             tong += a[i];
25:         }
26:         Console.WriteLine("Tong la {0} ", tong);
27:         int max = a[0];
28:         for (int i = 1; i < len; i++)
29:         {
30:             if (max < a[i])
31:                 max = a[i];
32:         }
33:         Console.WriteLine("Phan tu lon nhat {0} ", max);
34:     }
35: }
36: }
```

Trong chương trình trên dòng 8 dùng để khai báo một mảng 1 chiều kích thước tối đa 100 phần tử. Dòng 9 khai báo biến len = 0 chính là chiều dài thật sự của mảng đang xét. Dòng 10, 11 hiển thị thông báo và nhập chiều dài thật sự của mảng. Để nhập các giá trị cho mảng chúng ta có thể dùng vòng lặp cho thao tác nhập (từ dòng 12 đến 16). Kết quả khi chạy chương trình như sau:



```
C:\Windows\system32\cmd.exe
Nhap vao chieu dai cua mang 5
a[0]=10
a[1]=2
a[2]=-1
a[3]=3
a[4]=4
a[0]=10
a[1]=2
a[2]=-1
a[3]=3
a[4]=4
Tong la 18
Phan tu lon nhat 10
Press any key to continue . . . _
```

Chương trình trên chúng ta thấy tất cả yêu cầu bài toán đều được thực hiện bên trong nội dung của hàm main, điều gì xảy ra khi chương trình có nhiều yêu cầu cần thực hiện, khi đó số lượng mã trong hàm main sẽ rất nhiều do đó rất khó cho việc quản lý và sửa lỗi chương trình. Do đó cách tiếp cận tốt hơn là chúng ta chia nhỏ ứng dụng chúng ta thành các hàm. Bằng cách sử dụng hàm, chương trình có thể được viết lại như sau:

```
1: using System;
2: namespace Mang1Chieu
3: {
4:     class Program
5:     {
6:         static int[] a = new int[100];
7:         static int len = 0;
8:         static void Main(string[] args)
9:         {
10:             NhapMang();
11:             XuatMang();
12:             int tong = TinhTong();
13:             Console.WriteLine("Tong la {0} ", tong);
14:             int max = TimMax();
15:             Console.WriteLine("Phan tu lon nhat {0} ", max);
16:         }
17:     }
18: }
```



```
17:         private static int TimMax()
18:         {
19:             int max = a[0];
20:             for (int i = 1; i < len; i++)
21:             {
22:                 if (max < a[i])
23:                     max = a[i];
24:             }
25:             return max;
26:         }
27:         private static int TinhTong()
28:         {
29:             int tong = 0;
30:             for (int i = 0; i < 10; i++)
31:             {
32:                 tong += a[i];
33:             }
34:             return tong;
35:         }
36:         private static void XuatMang()
37:         {
38:             for (int i = 0; i < len; i++)
39:             {
40:                 Console.WriteLine("a[{0}]={1}", i, a[i]);
41:             }
42:         }
43:         private static void NhapMang()
44:         {
45:             Console.Write("Nhap vao chieu dai cua mang ");
46:             len = int.Parse(Console.ReadLine());
47:             for (int i = 0; i < len; i++)
48:             {
49:                 Console.Write("a[{0}]=", i);
50:                 a[i] = int.Parse(Console.ReadLine());
51:             }
52:         }
53:     }
```

54: }

3.6 Sử dụng công cụ dò lỗi của Visual Studio.NET

Có hai loại lỗi khi soạn thảo chương trình:

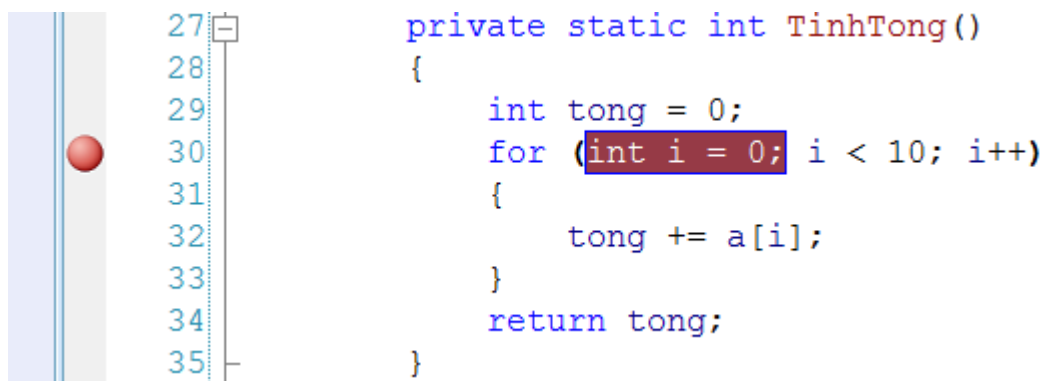
- Lỗi cú pháp: trình biên dịch sẽ tự động tìm ra lỗi và đưa ra các cảnh báo tương ứng. Người lập trình phải sửa tất cả các lỗi cú pháp trước khi chương trình được biên dịch.
- Lỗi thực thi: lỗi này xảy ra khi chạy chương trình. Ví dụ chương trình cho kết quả sai hay bị hỏng.

Do đó, kỹ năng dò lỗi và kiểm thử chương trình rất quan trọng trong quá trình xây dựng và phát triển chương trình.

Các kỹ năng quan trọng khi dò lỗi:

- Cách đặt các điểm dừng (breakpoint) và chạy các điểm dừng như thế nào?
- Cách chạy từng bước qua các lời gọi phương thức.
- Bằng cách nào kiểm tra và thay đổi giá trị của biến, dữ liệu thành viên của lớp.
- Dò lỗi có thể được thực hiện theo nhiều cách. Thông thường qua thực đơn.

Để đặt điểm dừng trong chương trình, chọn điểm dừng và nhấn F9:



Trong hình trên chương trình sẽ dừng lại trước khi thực hiện lệnh for. Khi chương trình thực hiện đến lệnh này sẽ hiển thị có màu vàng như sau:

```
27 private static int TinhTong()  
28 {  
29     int tong = 0;  
30     for (int i = 0; i < 10; i++)  
31     {  
32         tong += a[i];  
33     }  
34     return tong;  
35 }
```

Để xem giá trị của các biến tại thời điểm hiện tại chúng ta có thể hiển thị cửa sổ Watch trong thực đơn Debug\Windows\Watch\Watch1 và nhập vào tên biến cần kiểm tra như sau:

Name	Value	Type
tong	0	int
i	0	int
a	{int[100]}	int[]
[0]	1	int
[1]	2	int
[2]	3	int
[3]	4	int
[4]	5	int
[5]	5	int
[6]	0	int

Để chạy chương trình từng bước nhấn F11. Để thoát ra khỏi hàm hiện tại dùng Shift + F11.

Các chúng ta sinh viên nên sử dụng kỹ thuật này để chạy từng bước qua các thuật toán, xem kết quả của các biến khi thực hiện. Đặc biệt dùng để kiểm tra:

- Cách thức thực hiện của các lệnh điều khiển, lệnh lặp.
- Trình tự thực hiện chương trình.
- Giá trị của biến cục bộ, toàn cục khi thực hiện chương trình.
- Các gọi hàm, truyền tham trị, truyền tham biến.

3.7 Bài tập

Thực hiện các yêu cầu sau bổ sung trong ví dụ mảng một chiều:

1. Tìm phần tử nhỏ nhất trong mảng.
 2. Xóa mảng tại vị trí vt.
 3. Tìm vị trí đầu tiên của phần tử x trong mảng.
 4. Xóa phần tử x đầu tiên trong mảng.
 5. Xóa tất cả phần tử x trong mảng.
 6. Xóa tất cả phần tử lớn nhất trong mảng
 7. Đếm số lượng phần tử trong mảng (không trùng nhau).
 8. Tìm tất cả vị trí của phần tử lớn nhất trong mảng.
 9. Sắp xếp mảng tăng, giảm.
-

Chương 4: Đối tượng và kiểu

Mục đích của chương:

- Định nghĩa lớp, đối tượng.
- Sử dụng nạp chồng phương thức: C# cho phép chúng ta định nghĩa những dạng khác nhau của một phương thức trong một lớp. Trình biên dịch sẽ tự động chọn phương thức nào thích hợp nhất dựa vào tham số truyền vào của nó.
- Phương thức tạo lập và phương thức hủy: chỉ rõ một số hành động tự động kèm theo khi khởi tạo đối tượng và tự động giải phóng khi kết thúc đối tượng.
- Cấu trúc (struct): là những kiểu giá trị cung cấp những tiện ích khi chúng ta cần một số tính năng của lớp nhưng không cần vất vả tạo ra một thực thể lớp.
- Ứng dụng minh họa các khai báo lớp và sử dụng các thành phần của lớp.

4.1 Lớp

Khả năng tạo các kiểu dữ liệu mới là đặc trưng của ngôn ngữ lập trình hướng đối tượng. Chúng ta có thể tạo ra kiểu mới bằng cách khai báo và định nghĩa lớp. Một lớp là một kiểu dữ liệu do người dùng định nghĩa chứa biến, thuộc tính, hay các phương thức trong đó. Một lớp định nghĩa các đặc trưng trừu tượng của một cái gì đó (đối tượng) bao gồm các đặc trưng (thuộc tính, trường) hay hành vi (phương thức, sự kiện). Ví dụ, một lớp SinhVien chứa tất cả các đặc trưng của sinh viên như tên, tuổi, địa chỉ và các phương thức như xem bảng điểm, đăng kí học phần, xem kết quả đăng kí học phần.

Thể hiện của một lớp gọi là một đối tượng. Đối tượng được tạo trong bộ nhớ khi chương thực thi.

Một thuận lợi lớn nhất của các lớp trong ngôn ngữ lập trình hướng đối tượng là chúng có khả năng đóng gói các đặc trưng và khả năng của một thực thể vào trong một đơn vị mã chương trình.

a) Định nghĩa lớp

Để định nghĩa một kiểu mới hay một lớp chúng ta đầu tiên khai báo nó và sau đó định nghĩa các phương thức và trường của nó. Chúng ta khai báo lớp sử dụng từ khóa class.

Cú pháp:

```
[thuộc_tính] [bổ_từ_truy_xuất] class định_danh [:lớp_cơ_sở ]  
{  
    Nội_dung_lớp  
}
```

Thông thường một lớp sẽ dùng bổ_từ_truy_xuất là từ khóa public

Định_danh là tên của lớp. Nội dung của lớp được định nghĩa trong phần{ }.

Trong C# mọi thứ xảy ra trong một lớp. Ví dụ sau định nghĩa một lớp SinhVien:

```
public class SinhVien  
{  
    public static int Main( )  
    {  
        ...  
    }  
}
```

Lúc khai báo một lớp mới, chúng ta định nghĩa thuộc tính của tất cả đối tượng cũng như những hành vi của lớp đó. Ví dụ: một lớp SinhVien có các thuộc tính: maSV, tenSV, ngaySinh, tuoi và các hành vi như sau: XemDiem, XemThongTinHocPhi.

Ngôn ngữ lập trình hướng đối tượng phép chúng ta tạo một kiểu mới SinhVien, đồng thời đóng gói những đặc trưng (thuộc tính) và khả năng (phương thức) của chúng. Lúc đó, lớp SinhVien có thể có các biến thành viên (trường) như maSV, tenSV, ngaySinh, tuoi và các hàm thành viên XemDiem(), XemThongTinHocPhi().

Lớp SinhVien có thể được khai báo như sau:

```
public class SinhVien  
{  
    public string maSV;  
    public string tenSV;  
    public DateTime ngaySinh;  
    public int tuoi;  
    public void XemDiem()
```

```
{
    ///Noi dung ham
}
public void XemThongTinHocPhi()
{
    ///Noi dung ham
}
}
```

b) Định nghĩa đối tượng

Chúng ta có thể tạo thể hiện của một lớp, thể hiện này chính là đối tượng được tạo trong thời gian chương trình thực thi. Một tập các các giá trị của các thuộc tính được gọi là trạng thái của đối tượng. Đối tượng bao gồm các trạng thái và hành vi được định nghĩa trong khai báo lớp.

Trong ví dụ lớp SinhVien trên, chúng ta không thể gán dữ liệu đến kiểu SinhVien, đầu tiên chúng ta phải khai báo một đối tượng kiểu SinhVien theo đoạn mã sau.

```
SinhVien sv1;
```

Một khi chúng ta tạo một thể hiện của lớp SinhVien chúng ta có thể gán dữ liệu đến các trường của nó. Dùng từ khóa new để tạo đối tượng.

```
sv1 = new SinhVien();
sv1.maSV = "012345";
```

Xét một lớp phân số dùng để mô tả các đặc trưng của các đối tượng phân số. Thành phần dữ liệu trong một lớp phân số bao gồm hai thành phần: phần tử số và phần mẫu số. Chúng ta muốn hiển thị phân số ra màn hình. Chúng ta có thể thực hiện được yêu cầu của lớp phân số trên bằng cách khai báo một lớp phân số định nghĩa một phương thức và các biến như sau:

Ví dụ 4.1: Định nghĩa lớp phân số

```
1: class PhanSo
2:     {
3:         ///Khai bao thanh phan du lieu
4:         public int tu;
5:         public int mau;
6:         ///Khai bao cac phuong thuc
```

```
7:         public void Xuat()  
8:         {  
9:             Console.WriteLine("{0}/{1}", tu, mau);  
10:        }  
11:    }  
12:    public class Tester  
13:    {  
14:        static void Main( )  
15:        {  
16:            PhanSo t = new PhanSo( );  
17:            t.Xuat( );  
18:        }  
19:    }
```

Phương thức Xuat() được định nghĩa trả về kiểu void. Nó không trả về giá trị cho phương thức gọi nó. Trong hàm Main() thể hiện của lớp PhanSo được tạo và địa chỉ của nó được gán đến đối tượng t. Bởi vì t là thể hiện của đối tượng PhanSo nên nó có thể sử dụng phương thức Xuat() để gọi phương thức hiển thị phân số ra màn hình:

```
t.Xuat( );
```

4.2 Thành viên của lớp

Các thành phần khai báo bên trong của một lớp (một lớp cho phép khai báo nhiều nơi trong chương trình) được gọi là các thành viên của lớp. Thành viên của lớp bao gồm thành phần dữ liệu, hành vi của lớp, các sự kiện trong lớp, ủy nhiệm....

Dữ liệu bao gồm:

- Trường: các biến được khai báo private.
- Thuộc tính: các biến được khai báo public.

Hành vi gồm các phương thức được định nghĩa trong lớp.

a) Kiểu truy xuất

Kiểu truy xuất xác định các thành viên (bao gồm trường, thuộc tính, phương thức, sự kiện, ủy nhiệm) của lớp có thể được nhìn thấy và sử dụng tại các thành viên bên trong hay bên ngoài lớp hay không. Bảng sau liệt kê các kiểu truy xuất và phạm vi của chúng:

Kiểu truy xuất	Hạn chế
public	Không hạn chế. Thành viên được đánh dấu public được nhìn thấy bởi bất kỳ phương thức của bất kỳ lớp
private	Các thành viên trong lớp A được đánh dấu private được truy xuất chỉ trong các phương thức của lớp A. Mặc định các thành viên được khai báo private.
protected	Các thành viên trong lớp A được đánh dấu protected được truy xuất trong các phương thức của lớp A và các lớp dẫn xuất từ A
internal	Các thành viên trong lớp A được đánh dấu internal được truy xuất trong các phương thức của bất kỳ lớp trong assembly của A
protected internal	Tương đương với protected or internal

Nếu các thành viên không khai báo các chỉ định truy xuất thì được hiểu ngầm là thành viên private.

b) Tạo đối tượng

Trong chương trước chúng ta phân biệt giữa kiểu giá trị và kiểu tham chiếu. Các kiểu cơ bản trong C# là kiểu giá trị và chúng được tạo trên stack. Đối tượng vì nó là một kiểu tham chiếu nên được tạo trên heap. Ví dụ:

```
PhanSo t = new PhanSo( );
```

Trong trường hợp này t không thật sự chứa đối tượng PhanSo, nó chỉ chứa địa chỉ của đối tượng PhanSo được tạo trên Heap. t chỉ thật sự là một tham chiếu của đối tượng đó.

Xét ví dụ sau:

```
int a =10;
```

```
int b = a;  
b = 100;  
Console.WriteLine(b);
```

Khi thực hiện đoạn lệnh trên chương trình sẽ xuất ra giá trị 100 vì b và a được khai báo tại hai vùng nhớ khác nhau trên stack nên khi thay đổi giá trị của b sẽ không ảnh hưởng đến giá trị của a. Tuy nhiên nếu kiểu dữ liệu là đối tượng như sau:

```
PhanSo a = new PhanSo(1, 2);  
a.Xuat();  
PhanSo b = a;  
b.tu = 100;  
b.Xuat();  
a.Xuat();
```

Chúng ta thấy kết quả dữ liệu của phân số a và b là như nhau vì đối tượng a và b cùng tham chiếu đến một vùng nhớ trong heap nên khi thay đổi giá trị của đối tượng này sẽ kéo theo sự thay đổi giá trị của đối tượng khác. Do đó, cần cẩn thận khi thực hiện thao tác trên các đối tượng chia sẻ chung một vùng nhớ.

c) Phương thức tạo lập

Một phương thức được gọi bất cứ lúc nào chúng ta tạo một thể hiện cho một đối tượng gọi là phương thức tạo lập. Nhiệm vụ của phương thức tạo lập là tạo đối tượng chỉ bởi lớp và đặt đối tượng vào trong trạng thái sẵn sàng (thông thường dùng để khởi tạo cho các thành phần dữ liệu). Trước khi phương thức tạo lập chạy, đối tượng chưa tồn tại trong bộ nhớ, sau khi phương thức tạo lập thực hiện hoàn thành, bộ nhớ sẽ lưu trữ một thể hiện hợp lệ của một lớp. Chúng có các đặc trưng sau:

- Phương thức có tên trùng với tên lớp.
- Không có kiểu trả về và thông thường được khai báo public.
- Nếu chúng ta không định nghĩa khi khai báo lớp, phương thức tạo lập mặc định đại diện của hệ thống sẽ được gọi.
- Mỗi lớp có thể định nghĩa nhiều phương thức tạo lập khác nhau nhưng dấu hiệu của các tham số phải khác nhau (kiểu khác nhau).

Trong ví dụ lớp PhanSo, chúng ta không định nghĩa một phương thức tạo lập, trình biên dịch sẽ tự động tạo một phương thức tạo lập mặc định cho nó. Khi sử dụng phương thức tạo lập mặc định các biến được khởi tạo giá trị mặc định như sau:

Kiểu	Giá trị mặc định
Kiểu số (int, long. .)	0
Bool	False
Char	'\0'
Enum	0
Reference	null

Thông thường chúng ta định nghĩa riêng một phương thức tạo lập và cung cấp tham số cho phương thức tạo lập để khởi tạo các biến cho đối tượng của chúng ta.

```
1:  class PhanSo
2:      {
3:          //Khai bao thanh phan du lieu
4:          public int tu;
5:          public int mau;
6:          //Khai bao cac phuong thuc
7:          public void Xuat()
8:          {
9:              Console.WriteLine("{0}/{1}", tu, mau);
10:         }
11:         public PhanSo()
12:         {
13:             tu = 5;
14:             mau = 10;
15:         }
16:         public PhanSo(int t, int m)
17:         {
18:             tu = t;
19:             mau = m;
20:         }
21:         public PhanSo(PhanSo a)
```

```
22:         {
23:             tu = a.mau;
24:             mau = a.tu;
25:         }
26:     }
27:     class Program
28:     {
29:         static void Main(string[] args)
30:         {
31:             PhanSo a = new PhanSo();
32:             a.Xuat();
33:             PhanSo b = new PhanSo(3,5);
34:             b.Xuat();
35:         }
36:     }
```

Trong ví dụ trên dòng 31 và 33 dùng hai cách khác nhau để khởi tạo giá trị cho phân số a và b. Tùy vào mục đích sử dụng ta có thể dùng nhiều cách khác nhau để khởi tạo giá trị dữ liệu cho các đối tượng.

d) Khởi tạo giá trị cho các biến

Thay vì khởi tạo giá trị các biến thành viên thông qua phương thức tạo lập chúng ta có thể thực hiện gán giá trị trực tiếp cho biến. Phương thức tạo lập của chương trình trên có thể được viết như sau:

```
public PhanSo()
{
    Console.Write("Nhập tu ");
    tu = int.Parse(Console.ReadLine());
    Console.Write("Nhập mau ");
    mau = int.Parse(Console.ReadLine());
}
```

e) Phương thức tạo lập sao chép

Phương thức tạo lập sao chép tạo một đối tượng mới bằng cách chép các biến từ đối tượng hiện tại đến đối tượng mới cùng kiểu. Ví dụ chúng ta muốn truyền một đối tượng PhanSo b đến một đối tượng PhanSo a của phương thức tạo lập nhằm tạo ra đối tượng mới có cùng giá trị với đối tượng cũ. C# không cung cấp phương thức tạo lập sao chép, do đó chúng ta phải tự tạo. Ví dụ:

```
public PhanSo(PhanSo a)
{
    tu = a.mau;
    mau = a.tu;
}
```

Dựa trên việc khai báo phương thức tạo lập bên trên ta có thể tạo một đối tượng sinh viên b từ một đối tượng sinh viên a như sau:

```
PhanSo a = new PhanSo(1, 2);
a.Xuat();
PhanSo b = new PhanSo(a);
b.Xuat();
```

Trong ví dụ trên, 2 đối tượng a, b sẽ tham chiếu tới hai vùng nhớ khác nhau nên việc thay đổi dữ liệu của đối tượng này sẽ không ảnh hưởng đến dữ liệu của đối tượng khác.

f) Từ khóa this

Ví dụ 4.1 cần bổ sung thêm phương thức tạo lập và phương thức cộng hai phân số như sau:

```
1:  class PhanSo
2:      {
3:          //Khai báo thành phần dữ liệu
4:          public int tu;
5:          public int mau;
6:          //Khai báo các phương thức
7:          public void Xuat()
8:          {
9:              Console.WriteLine("{0}/{1}", tu, mau);
10:         }
11:         public PhanSo()
12:         {
13:             Console.Write("Nhập tu ");
14:             tu = int.Parse(Console.ReadLine());
15:             Console.Write("Nhập mau ");
16:             mau = int.Parse(Console.ReadLine());
17:         }
18:         public PhanSo(int t, int m)
19:         {
20:             tu = t;
21:             mau = m;
```

```
22:         }
23:     public PhanSo(PhanSo a)
24:     {
25:         tu = a.mau;
26:         mau = a.tu;
27:     }
28:     public PhanSo Cong(PhanSo a, PhanSo b)
29:     {
30:         PhanSo kq = new PhanSo();
31:         kq.tu = a.tu * b.mau + a.mau * b.tu;
32:         kq.mau = a.mau * b.mau;
33:         return kq;
34:     }
35: }
36: class Program
37: {
38:     static void Main(string[] args)
39:     {
40:         PhanSo a = new PhanSo(1, 2);
41:         a.Xuat();
42:         PhanSo b = new PhanSo(a);
43:         b.Xuat();
44:         PhanSo tong = a.Cong(a, b);
45:         tong.Xuat();
46:         Console.ReadKey();
47:     }
48: }
49: }
```

Trong ví dụ trên chúng ta định nghĩa hàm Cong với hai tham số đầu vào là hai phân số. Do đó khi gọi thực hiện lệnh cộng hai phân số trong dòng 44 chúng ta thấy đối tượng a xuất hiện hai lần (đối tượng gọi hàm cộng và là tham số của hàm cộng). Điều này thật sự không cần thiết. Chúng ta có thể dùng đối tượng this để giải quyết vấn đề này. Đối tượng this chỉ ra trạng thái hiện tại của đối tượng. Đối tượng this (con trỏ this) là một con trỏ ẩn dùng để tham chiếu đến các thành viên không tĩnh (không khai báo từ khóa static) của lớp. Chúng ta có thể tham chiếu đến các phương thức và các biến khác dựa trên tham chiếu this.

Tham chiếu this có thể sử dụng theo ba trường hợp sau:

- Tránh xung đột tên.

- Truyền đối tượng hiện tại là tham số của một phương thức khác.
- Dùng với chỉ mục.

Sử dụng đối tượng `this`, chương trình trên có thể viết thêm hàm như sau:

```
public PhanSo Cong(PhanSo b)
{
    PhanSo kq = new PhanSo();
    kq.tu = this.tu * b.mau + this.mau * b.tu;
    kq.mau = this.mau * b.mau;
    return kq;
}
```

Khi đó dòng 44 của chương trình có thể được viết lại như sau: `PhanSo tong = a.Cong(b);`

g) Sử dụng các thành viên tĩnh

Thành viên của một lớp có thể là thành viên của thể hiện (đối tượng) hay thành viên tĩnh của lớp. Thành viên thể hiện được kết hợp với thể hiện của kiểu, trong khi thành viên tĩnh được coi là phần của lớp. Chúng ta truy xuất các thành viên tĩnh thông qua tên lớp chúng ta khai báo. Ví dụ trong lớp `PhanSo` ta định nghĩa thêm thành viên tĩnh dùng để tìm ước số chung lớn nhất của hai số nguyên như sau:

```
public static int UCLN(int a, int b)
{
    a = Math.Abs(a);
    a = Math.Abs(b);
    while(a!=b)
    {
        if (a > b)
            a = a - b;
        if (a < b)
            b = b - a;
    }
    return a;
}
```

Khi đó để thực hiện việc gọi phương thức `UCLN` chúng ta cần gọi thông qua tên lớp chứ không được thông qua đối tượng;

```
int uc = PhanSo.UCLN(10, 20);
```

Trong C#, sẽ không hợp lệ nếu chúng ta truy xuất thành viên tĩnh qua thể hiện.

h) Gọi phương thức tĩnh

Hàm Main() là một phương thức tĩnh. Phương thức tĩnh hoạt động trên lớp hơn là trên thể hiện của lớp. Chúng không có tham chiếu this để trỏ đến phương thức tĩnh.

Phương thức tĩnh không thể truy xuất trực tiếp từ các thành viên không tĩnh.

Trong ví dụ trên phương thức Cong là phương thức không tĩnh, do đó phải được truy cập thông qua đối tượng như sau:

```
PhanSo tong = a.Cong(a, b);
```

i) Sử dụng phương thức tạo lập tĩnh

Nếu chúng ta khai báo một phương thức tạo lập tĩnh, chúng ta phải đảm bảo phương thức tạo lập tĩnh sẽ chạy trước khi thể hiện của lớp được tạo.

Ví dụ, chúng ta có thể thêm phương thức tạo lập tĩnh vào lớp PhanSo

```
static PhanSo( )  
{  
}
```

Chúng ta không được khai báo kiểu truy xuất trước phương thức tạo lập tĩnh. Hơn nữa, vì đây là một phương thức thành viên tĩnh, chúng ta không thể truy xuất vào biến thành viên không tĩnh, do đó tất cả các biến muốn được truy cập thông qua các thành viên tĩnh phải được khai báo static.

j) Sử dụng trường tĩnh

Dùng biến thành viên tĩnh là cách phổ biến để theo dõi số thể hiện hiện tại của lớp. Ví dụ:

```
1: using System;  
2: public class Meo  
3: {  
4:     public Meo( )  
5:     {  
6:         thehien ++;  
7:     }
```



```
8:     public static void SoMeo( )
9:     {
10:         Console.WriteLine("Số thể hiện của mèo la{0} ",
                             thehien);
11:     }
12:     private static int thehien = 0;
13: }
14: public class Tester
15: {
16:     static void Main( )
17:     {
18:         Meo.SoMeo( );
19:         Meo Tom= new Cat( );
20:         Meo.SoMeo( );
21:         Meo Linda = new Cat( );
22:         Meo.SoMeo( );
23:     }
24: }
```

k) Hủy đối tượng

C# cung cấp một cơ chế dọn rác để hủy các đối tượng một cách tự động và do đó chúng ta không cần một phương thức hủy rõ ràng để xóa các đối tượng trong bộ nhớ. Phương thức Finalize() được gọi bởi bộ dọn rác khi đối tượng của chúng ta bị hủy. Hàm này chỉ giải phóng các tài nguyên mà đối tượng này đang giữ.

Ví dụ khai báo phương thức hủy:

```
~ PhanSo ( )
{
    ///Mã
}
```

Hay

```
PhanSo.Finalize( )
{
    ///Mã
}
```

1) Nạp chồng phương thức

Thông thường chúng ta muốn khai báo nhiều hàm với cùng tên. C# cho phép khai báo các hàm trùng tên nhưng dấu hiệu phải khác nhau. Ví dụ chúng ta muốn dùng nhiều phương thức tạo lập khác nhau để khởi tạo các kiểu đối tượng khác nhau. Dấu hiệu của một phương thức được định nghĩa bởi tên và danh sách tham số. Hai phương thức có dấu hiệu khác nhau nếu có tên và danh sách tham số khác nhau.

Tham số khác nhau bởi số tham số hay kiểu khác nhau. Ví dụ các hàm sau có các dấu hiệu khác nhau:

```
void PhuongThuc(int p1);  
void PhuongThuc (int p1, int p2);  
void PhuongThuc (int p1, string s1);
```

Ví dụ minh họa sử dụng quá tải hàm cho phương thức tạo lập của lớp MaTran

```
1: using System;  
2: namespace MaTran  
3: {  
4:     public class MaTran  
5:     {  
6:         public double[, ] mt;  
7:         public int hang;  
8:         public int cot;  
9:         #region Phuong thuc tao lap  
10:        public MaTran()  
11:        {  
12:            //Phuong thuc tao lap mac dinh  
13:        }  
14:        public MaTran(int h, int c)  
15:        {  
16:            this.hang = h;  
17:            this.cot = c;  
18:            mt = new double[h, c];  
19:        }  
20:        public MaTran(MaTran a)  
21:        {  
22:            mt = new double[a.hang, a.cot];  
23:            this.hang = a.hang;  
24:            this.cot = a.cot;  
25:            for (int i = 0; i < a.hang; i++)  
26:                for (int j = 0; j < a.cot; j++)  
27:                    this.mt[i, j] = a.mt[i, j];  
28:
```

```
29:     }
30:     public MaTran(MaTran a, int h, int c)
31:     {
32:         mt = new double[h, c];
33:         this.hang = h;
34:         this.cot = c;
35:         for (int i = 0; i < h; i++)
36:             for (int j = 0; j < c; j++)
37:                 this.mt[i, j] = a.mt[i, j];
38:     }
39:     public void Nhap()
40:     {
41:         Console.WriteLine("Nhap mot ma tran cap {0}x{1} ",
42:                             this.hang, this.cot);
43:         for (int i = 0; i < this.hang; i++)
44:             for (int j = 0; j < this.cot; j++)
45:             {
46:                 //Console.Ghi("Nhap phan tu thu [{0}, {1}] = ",
47:                             i, j);
48:                 this.mt[i, j] = i + j;
49:             }
50:     }
51:     public void Nhap(int h, int c)
52:     {
53:         mt = new double[h, c];
54:         for (int i = 0; i < h; i++)
55:             for (int j = 0; j < c; j++)
56:             {
57:                 Console.Ghi("Nhap phan tu thu [{0}, {1}] = ",
58:                             i, j);
59:                 this.mt[i, j] =
60:                     double.Parse(Console.ReadLine());
61:             }
62:     }
63:     public override string ToString()
64:     {
65:         string kq = "";
66:         for (int i = 0; i < this.hang; i++)
67:         {
68:             kq += "\n";
69:             for (int j = 0; j < this.cot; j++)
70:             {
71:                 kq += "\t" + this.mt[i, j].ToString();
72:             }
73:         }
74:         return kq;
75:     }
76:     public MaTran Tong(MaTran a, MaTran b)
77:     {
78:         MaTran kq = new MaTran(a.hang, a.cot);
```

```
75:         for (int i = 0; i < b.hang; i++)
76:             for (int j = 0; j < b.cot; j++)
77:                 kq.mt[i, j] = a.mt[i, j] + b.mt[i, j];
78:         return kq;
79:     }
80:     public void Tong(MaTran b)
81:     {
82:         for (int i = 0; i < b.hang; i++)
83:             for (int j = 0; j < b.cot; j++)
84:                 this.mt[i, j] = this.mt[i, j] + b.mt[i, j];
85:     }
86: }
87: }
```

Trong ví dụ trên chúng ta khai báo ba phương thức tạo lập khác nhau cho việc khởi tạo một đối tượng kiểu MaTran. Tương tự chúng ta cũng định nghĩa nhiều phương thức Tong với các tham số khác nhau cho việc cộng hai ma trận.

m) Đóng gói dữ liệu với thuộc tính

Các thuộc tính cho phép các client truy xuất tới trạng thái của lớp như là truy xuất các trường thành viên trực tiếp của lớp. Truy xuất thuộc tính tương tự như truy xuất phương thức của lớp. Nghĩa là các client có thể truy xuất trực tiếp đến trạng thái của đối tượng (thuộc tính) mà không cần thông qua việc gọi thực thi các phương thức. Tuy nhiên, những người thiết kế lớp hay muốn che dấu trạng thái bên trong của các thành viên lớp và chỉ muốn các thuộc tính chỉ trực truy xuất gián tiếp thông qua phương thức của lớp.

Bằng cách phân tách thuộc tính với các phương thức của lớp, các nhà thiết kế có thể tự do thay đổi giá trị hay trạng thái của các thuộc tính bên trong của đối tượng khi cần.

Cho ví dụ như sau:

```
1: using System;
2: namespace DoiTuongDoiTuongHinhHoc
3: {
4:     public class HinhTron
5:     {
6:         private double r;
7:         public HinhTron()
8:         {
9:         }
```

```
10:         public HìnhTron(double bankinh)
11:         {
12:             this.R = bankinh;
13:         }
14:         public double DienTich()
15:         {
16:             return Math.Round(Math.PI*R*R, 2);
17:         }
18:         public override string ToString()
19:         {
20:             return "Duong tron co ban kinh " + R + " dien
                tich " + this.DienTich();
21:         }
22:     }
23: }
```

Lúc lớp HìnhTron được tạo, giá trị của thuộc tính bán kính r có thể được lưu trữ là một biến thành viên. Lúc lớp được thiết kế lại, giá trị của thuộc tính r có thể được tính toán lại hay lấy từ cơ sở dữ liệu. Nếu client truy xuất trực tiếp biến thành viên r, những thay đổi giá trị tính toán có thể làm hỏng ứng dụng client. Bằng cách phân tách và bắt buộc client muốn truy cập thuộc tính r phải sử dụng thông qua phương thức. Ta có thể khai báo như sau để cho phép đối tượng có thể truy cập biến r thông qua phương thức:

```
1: using System;
2: namespace DoiTuongDoiTuongHinhHoc
3: {
4:     public class HìnhTron
5:     {
6:         private double r;
7:         public double R
8:         {
9:             get
10:            {
11:                return r;
12:            }
13:            set
14:            {
15:                r = value;
16:            }
17:        }
18:         public HìnhTron()
19:         {
20:         }
```

```
21:         public HìnhTron(double bankinh)
22:         {
23:             this.R = bankinh;
24:         }
25:         public double DienTich()
26:         {
27:             return Math.Round(Math.PI*R*R, 2);
28:         }
29:         public override string ToString()
30:         {
31:             return "Duong tron co ban kinh " + R + " dien
                tich " + this.DienTich();
32:         }
33:     }
34: }
```

Thuộc tính đáp ứng cả hai mục đích:

- Chúng cung cấp một giao tiếp đơn giản với client, xuất hiện như là biến thành viên.
- Chúng thực hiện như là một phương thức. Tuy nhiên, chúng cung cấp cơ chế che dấu dữ liệu bởi một thiết kế hướng đối tượng.

Để khai báo thuộc tính, chúng ta viết kiểu thuộc tính và tên theo sau bởi {}. Bên trong {} chúng ta có thể khai báo cách thức truy xuất get hay set. Qua phương thức set chúng ta có một tham số ẩn value.

Trong ví dụ trên R là một thuộc tính, nó khai báo 2 kiểu truy xuất

```
public int R
{
    get
    {
        return r;
    }
    set
    {
        r = value;
    }
}
```

Giá trị của thuộc tính R có thể được lưu trữ trong cơ sở dữ liệu, trong biến thành viên private thậm chí lưu trữ tại một máy tính hay một dịch vụ nào đó trong mạng.

Truy xuất get

Phần thân của truy xuất get thì tương tự như phần thân của phương thức lớp. Chúng trả về một đối tượng có kiểu là kiểu của thuộc tính. Trong ví dụ trên, truy xuất get của R trả về giá trị double. Nó trả về giá trị của biến thành viên private r.

Bất cứ lúc nào chúng ta tham chiếu đến thuộc tính, truy xuất get được yêu cầu để đọc giá trị của thuộc tính.

```
HinhTron t = new HinhTron (1, 5);  
double bankinh = t.R;
```

Truy xuất set

Truy xuất set dùng để gán giá trị cho thuộc tính, nó tương tự như một phương thức lớp trả về kiểu void. Lúc chúng ta định nghĩa một truy xuất set, chúng ta phải sử dụng từ khóa value (tham số ẩn) để biểu diễn tham số nơi mà giá trị được truyền và lưu trữ trong thuộc tính:

```
set  
{  
    r = value;  
}
```

Lúc chúng ta gán một giá trị đến một thuộc tính, truy xuất set tự động được yêu cầu và tham số ẩn value được đặt giá trị mà chúng ta cần gán cho thuộc tính:

Ví dụ ta có thể dùng lệnh sau trong một phương thức nào đó của lớp HinhTron để tăng giá trị của thuộc tính r lên 1:

```
r++;
```

Thuận lợi của các tiếp cận này là client có thể truy xuất trực tiếp thuộc tính mà không cần quan tâm tới các dữ liệu ẩn bên trong.

Các trường chỉ đọc (readonly)

Nếu chúng ta muốn tạo một phiên bản của lớp ThoiGian có nhiệm vụ cung cấp một giá trị tĩnh, biểu diễn ngày và thời gian hiện tại. Chúng ta có thể dùng khai báo readonly để khai báo các thuộc tính:

```
public class ThoiGian
{
    static ThoiGian()
    {
        System.DateTime dt = System.DateTime.Now;
        Nam = dt.Year;
        Thang = dt.Month;
        Ngay = dt.Day;
        Gio = dt.Hour;
        Phut = dt.Minute;
        Giay = dt.Second;
    }
    public static int Nam;
    public static int Thang;
    public static int Ngay;
    public static int Gio;
    public static int Phut;
    public static int Giay;
}
public class Tester
{
    static void Main()
    {
        System.Console.WriteLine("This Nam: {0}",
            ThoiGian.Nam.ToString());
        ThoiGian.Nam = 2002;
        System.Console.WriteLine("This Nam: {0}",
            ThoiGian.Nam.ToString());
    }
}
```

Trong trường hợp này, giá trị của biến Nam sẽ vẫn thay đổi khi chúng ta thực hiện phép gán Nam = 2002. Chúng ta muốn đánh dấu các giá trị tĩnh là hằng nhưng không thể vì chúng ta không khởi gán giá trị cho chúng cho tới khi phương thức tạo lập tĩnh được thực hiện. C# cung cấp từ khóa readonly dùng cho mục đích này. Nếu chúng ta thay đổi khai báo biến thành viên của lớp như sau:


```
public static readonly int Nam;  
public static readonly int Thang;  
public static readonly int Ngay;  
public static readonly int Gio;  
public static readonly int Phut;  
public static readonly int Giay;;
```

và không dùng lệnh:

```
// ThoiGian. Year = 2002; // Thực hiện lệnh này sẽ gây ra lỗi
```

Trình biên dịch thực hiện và chạy theo đúng yêu cầu của chúng ta.

4.3 Cấu trúc (Struct)

a) Làm việc với kiểu cấu trúc

Kiểu dữ liệu tham chiếu luôn được tạo trên heap. Trong một số trường hợp, một lớp chứa quá ít dữ liệu để cần sự quản lý của heap. Tốt nhất trong trường hợp này chúng ta dùng cấu trúc. Bởi vì cấu trúc được lưu trữ trong stack, điều này giảm bớt chức năng quản lý bộ nhớ.

Cấu trúc cũng có riêng trường, phương thức tạo lập và phương thức giống lớp (nhưng không giống kiểu liệt kê). Tuy nhiên nó là kiểu giá trị không phải kiểu tham chiếu.

b) Các kiểu cấu trúc phổ biến

Trong C#, các kiểu số cơ bản như int, long, float là tên hiệu của cấu trúc System.Int32, System.Int64, và System.Single. Nghĩa là chúng ta có thể gọi phương thức trên biên thuộc kiểu này. Ví dụ, tất cả cấu trúc này cung cấp phương thức ToString để chuyển số sang chuỗi. Các lệnh sau là hợp lệ trong C#:

```
int i = 99;  
Console.WriteLine(i. ToString());  
Console.WriteLine(55. ToString());  
float f = 98.765F;  
Console.WriteLine(f. ToString());  
Console.WriteLine(98.765F. ToString());
```

Console.WriteLine sẽ tự động gọi phương thức ToString khi cần. Sử dụng phương thức tĩnh trong cấu trúc rất phổ biến. Ví dụ phương thức tĩnh Int32.Parse thường được dùng chuyển một chuỗi sang số:

```
string s = "42";  
int i = Int32.Parse(s);
```

Trong những cấu trúc trên cũng chứa những trường tĩnh hữu ích như Int32.MaxValue để lấy giá trị lớn nhất của biến int và Int32.MinValue lấy giá trị nhỏ nhất của biến int.

Bảng sau hiển thị kiểu dữ liệu cơ bản trong C#, nó có thể tương đương kiểu lớp hay cấu trúc:

Từ khóa	Kiểu tương đương	Lớp hoặc cấu trúc
bool	System.Boolean	Cấu trúc
byte	System.Byte	Cấu trúc
decimal	System.Decimal	Cấu trúc
Double	System.Double	Cấu trúc
Float	System.Single	Cấu trúc
Int	System.Int32	Cấu trúc
Long	System.Int64	Cấu trúc
Object	System.Object	Lớp
Sbyte	System.SByte	Cấu trúc
Short	System.Int16	Cấu trúc
String	System.String	Lớp

Từ khóa	Kiểu tương đương	Lớp hoặc cấu trúc
UInt	System.UInt32	Cấu trúc
Ulong	System.UInt64	Cấu trúc
Ushort	System.UInt16	Cấu trúc

c) Khai báo kiểu cấu trúc

Để khai báo kiểu cấu trúc, chúng ta khai báo từ khóa struct theo sau bởi tên và phần thân trong dấu “{}”. Ví dụ, đây là cấu trúc tên ThoiGian chứa 3 trường public: Gio, Phut, và Giay.

```
struct Time
{
    public int Gio, Phut, Giay;
}
```

Tương tự như lớp, chỉ định public cho các trường không được khuyến khích trong hầu hết trường hợp vì không có cách để đảm bảo trường public chứa giá trị hợp lệ. Ví dụ, chúng ta có thể gán trị của Phut hay Giay > 60. Cách tốt hơn là đánh dấu trường private và cung cấp truy xuất cấu trúc của chúng ta qua phương thức tạo lập hay phương thức như sau:

```
struct ThoiGian
{
    public ThoiGian (int hh, int mm, int ss)
    {
        Gio= hh % 24;
        phut = mm % 60;
        giay = ss % 60;
    }
    public int Gio()
    {
        return gio;
    }
    ...
    private int gio, phut, giay;
}
```

Khi chúng ta sao chép một biến kiểu giá trị, chúng ta có hai bản sao của giá trị. Ngược lại, lúc chúng ta sao chép một biến kiểu tham chiếu, chúng ta có hai tham chiếu đến cùng đối tượng.

d) Tìm hiểu sự khác nhau giữa lớp và cấu trúc

Chúng ta không thể khai báo phương thức tạo lập mặc định (phương thức tạo lập không có tham số) cho cấu trúc. Ví dụ sau sẽ biên dịch nếu ThoiGian là lớp nhưng với cấu trúc thì không:

```
struct ThoiGian
{
    public ThoiGian () {... } // Lỗi biên dịch
    ...
}
```

Bởi vì trình biên dịch luôn tạo phương thức tạo lập mặc định cho cấu trúc. Trong khi trong lớp, trình biên dịch chỉ tạo nó khi không có khai báo phương thức tạo lập trong lớp.

Trình biên dịch tạo phương thức tạo lập mặc định cho một cấu trúc luôn gán 0, false, null giống như lớp cho các kiểu dữ liệu tương ứng. Do đó chúng ta muốn khởi tạo các giá trị khác cho trường chúng ta phải dùng phương thức tạo lập không mặc định của chúng ta. Tuy nhiên chúng ta phải khởi tạo tất cả các trường trong cấu trúc, nếu không trình biên dịch sẽ thông báo lỗi. Ví dụ sau sẽ được biên dịch nếu ThoiGian là một lớp và gày được gán bằng 0 nhưng vì ThoiGian là cấu trúc nên lỗi khi biên dịch:

```
struct ThoiGian
{
    public ThoiGian (int hh, int mm)
    {
        gio = hh;
        phut = mm;
    } // lỗi thời gian biên dịch: gày chưa được khởi tạo
    ...
    private int gio, phut, gày;
}
```

Trong một lớp chúng ta có thể khởi tạo trường lúc khai báo nhưng trong cấu trúc chúng ta không thể. Ví dụ sau sẽ được biên dịch nếu ThoiGian là một lớp nhưng vì ThoiGian là cấu trúc nên lỗi khi biên dịch:

```
struct ThoiGian
{
    ...
    private int gio = 0; // lỗi biên dịch
    private int phut;
    private int giay;
}
```

Bảng sau tóm tắt sự khác nhau giữa lớp và cấu trúc

Câu hỏi	Cấu trúc	Lớp
Kiểu của nó là gì?	Cấu trúc là kiểu giá trị	Lớp là kiểu tham chiếu
Thẻ hiện của nó lưu trên stack hay heap?	Thẻ hiện của cấu trúc được gọi là giá trị và lưu trên stack	Thẻ hiện của lớp là đối tượng được lưu trên heap
Chúng ta có thể khai báo phương thức tạo lập mặc định?	Không	Có
Nếu chúng ta khai báo phương thức tạo lập riêng chúng ta, trình biên dịch sẽ tạo phương thức tạo lập mặc định?	Có	Không
Nếu chúng ta không khởi tạo trường trong phương thức tạo lập riêng chúng ta, trình biên dịch sẽ tự động khởi tạo cho chúng ta?	Không	Có
Chúng ta được phép khởi tạo trường thể hiện lúc khai báo?	Không	Có

Còn một sự khác nhau khác giữa lớp và cấu trúc là lớp có thể kế thừa từ lớp cơ sở nhưng cấu trúc thì không.

e) Khai báo biến cấu trúc

Sau khi chúng ta định nghĩa một kiểu cấu trúc, chúng ta có thể dùng nó như các kiểu khác. Ví dụ, nếu chúng ta định nghĩa cấu trúc ThoiGian chúng ta có thể tạo biến, trường, tham số kiểu ThoiGian như sau:

```
struct ThoiGian
{
    ...
    private int gio, phut, giay;
}
class ViDu
{
    public void PhuongThuc(ThoiGian para)
    {
        ThoiGian bienCucBo;
        ...
    }
    private ThoiGian thoiGianHienTai;
}
```

f) Khởi tạo cấu trúc

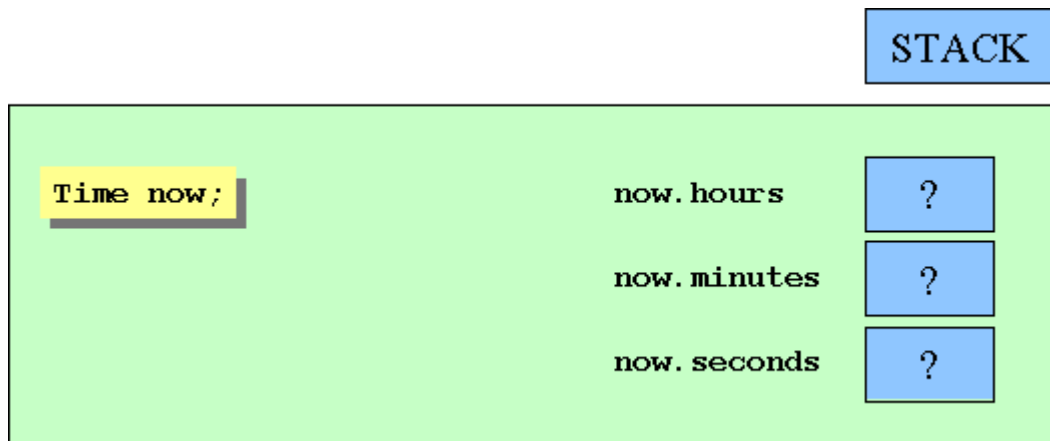
Giả sử chúng ta định nghĩa lớp như sau:

```
struct Time
{
    ...
    private int hours, minutes, seconds;
}
```

Vì cấu trúc là kiểu dữ liệu nên chúng ta có thể tạo biến cấu trúc mà không cần gọi phương thức tạo lập như ví dụ sau:

```
Time now;
```

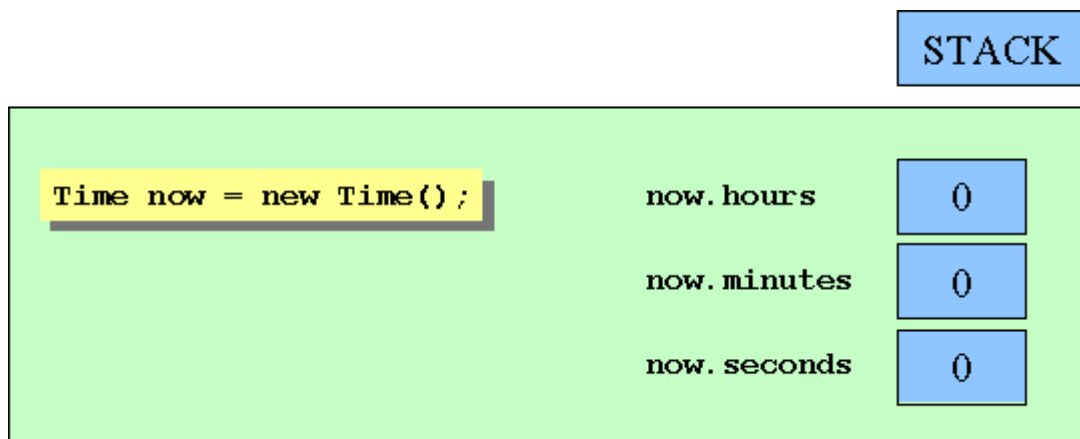
Trong ví dụ này, biến được tạo nhưng trường bên trái của nó ở trạng thái chưa được tạo. Nếu chúng ta truy xuất giá trị của trường này sẽ gây ra lỗi biên dịch. Hình sau mô tả trạng thái của trường trong biến now:



Nếu chúng ta gọi một phương thức tạo lập, quy tắc khởi tạo đảm bảo tất cả các biến được khởi tạo:

```
Time now = new Time();
```

Lúc này phương thức tạo lập mặc định được gọi khởi tạo tất cả các trường trong cấu trúc như hình sau:



Nếu chúng ta muốn viết phương thức tạo lập riêng, chúng ta có thể dùng nó để khởi tạo biến cấu trúc. Một phương thức tạo lập của cấu trúc phải luôn khởi tạo tất cả các trường của nó. Ví dụ:

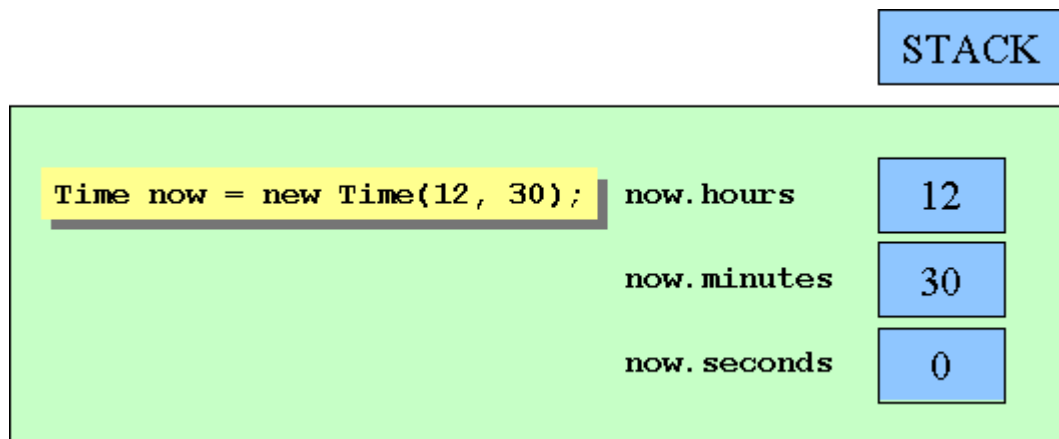
```
struct Time
{
    public Time(int hh, int mm)
    {
        hours = hh;
        minutes = mm;
        seconds = 0;
    }
}
```

```
    }  
    ...  
    private int hours, minutes, seconds;  
}
```

Trong ví dụ sau khởi tạo `now` bởi gọi phương thức tạo lập người dùng định nghĩa:

```
Time now = new Time(12, 30);
```

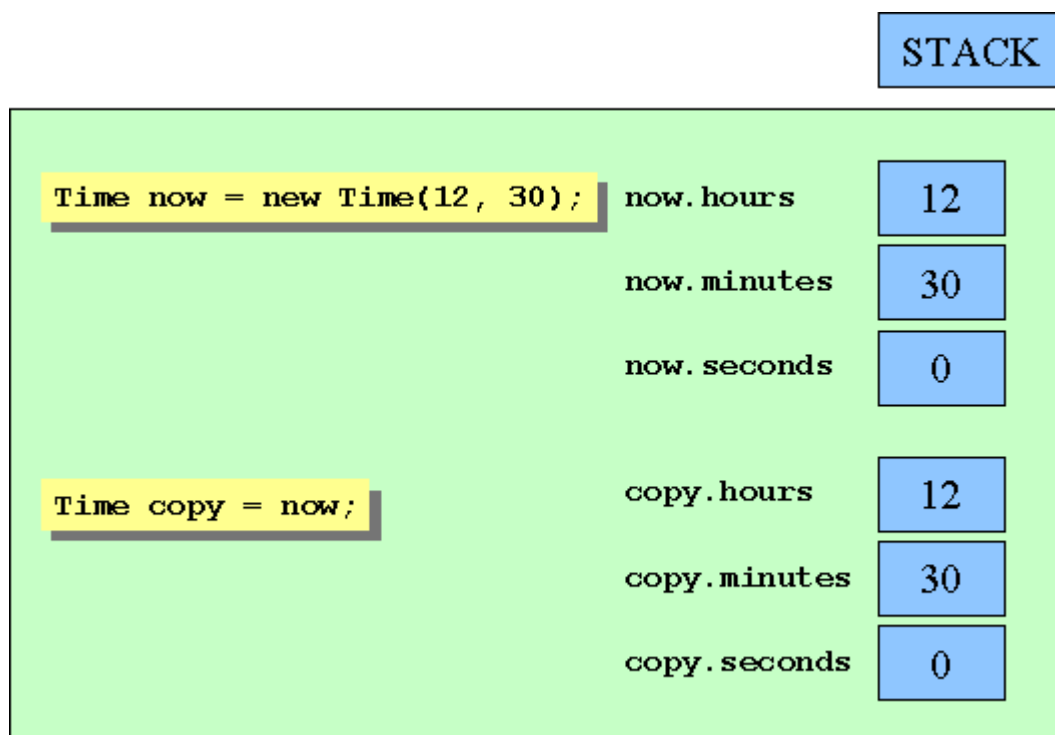
Kết quả trong stack được lưu trữ như trong hình sau:



g) Sao chép biến cấu trúc

Chúng ta được cho phép khởi gán hay gán một biến cấu trúc cho một biến cấu trúc khác nhưng chỉ nếu biến bên phải được khởi tạo hoàn chỉnh (tất cả các trường được khởi tạo). Ví dụ sau sẽ được biên dịch vì `now` được khởi tạo hoàn toàn:

```
Time now = new Time(12, 30);  
Time copy = now;
```

Trong ví dụ sau sẽ lỗi khi biên dịch vì `now` chưa được khởi tạo:

```
Time now;
Time copy = now; // lỗi biên dịch: now chưa được gán
```

Khi chúng ta sao chép một cấu trúc, các trường bên trái được sao chép trực tiếp từ các trường bên phải.

4.4 Quá tải toán tử

Trong C#, toán tử (operator) được định nghĩa là một phương thức tĩnh và phải trả về kết quả. Khi chúng ta tạo một toán tử cho lớp chúng ta có thể quá tải toán tử đó tương tự như cách chúng ta quá tải hàm. Ví dụ, quá tải toán tử cộng chúng ta có thể viết như sau:

```
public static PhanSo operator+(PhanSo lhs, PhanSo rhs)
```

Tham số đầu tiên `lhs` là toán hạng bên trái và tham số thứ hai `rhs` là toán hạng bên phải.

Dùng từ khóa `operator` theo sau bởi toán tử để chỉ quá tải toán tử. Chúng ta có thể tạo các toán tử như các toán tử logic, toán tử tương đương (`==`).

Ví dụ sau minh họa dùng quá tải toán tử +, -, *, /...cho ứng dụng xây dựng kiểu dữ liệu phân số:

```
public static PhanSo operator +(PhanSo a, PhanSo b)
{
    PhanSo kq = new PhanSo();
    kq.tu = a.tu * b.mau + a.mau * b.tu;
    kq.mau = a.mau * b.mau;
    return kq.RutGon();
}
//Cộng phân số với số x... . a/b = (a+x)/(b+x)
public static PhanSo operator +(PhanSo a, int x)
{
    PhanSo kq = new PhanSo();
    kq.tu = a.tu + x;
    kq.mau = a.mau;
    return kq.RutGon();
}
public static PhanSo operator -(PhanSo a, PhanSo b)
{
    PhanSo kq = new PhanSo();
    kq.tu = a.tu * b.mau - a.mau * b.tu;
    kq.mau = a.mau * b.mau;
    return kq.RutGon();
}
public static PhanSo operator -(PhanSo a, int x)
{
    PhanSo kq = new PhanSo();
    kq.tu = a.tu - x;
    kq.mau = a.mau;
    return kq.RutGon();
}

public static PhanSo operator *(PhanSo a, PhanSo b)
{
    PhanSo kq = new PhanSo();
    kq.tu = a.tu * b.tu;
    kq.mau = a.mau * b.mau;
    return kq.RutGon();
}

//nhân phân số a với một số nguyên x a/b * x = a/b*x
public static PhanSo operator *(PhanSo a, int x)
{
    PhanSo tam = new PhanSo((PhanSo)x);
```

```

        return a * tam();
    }

    public static PhanSo operator /(PhanSo a, PhanSo b)
    {
        return a * b.NghichDao();
    }
    //chia phân số a với một số nguyên x a/b chia x = a/b*x
    public static PhanSo operator /(PhanSo a, int x)
    {
        PhanSo tam = new PhanSo((PhanSo)x);
        return a * tam.NghichDao();
    }
    public static PhanSo operator ++(PhanSo a)
    {
        PhanSo kq = new PhanSo();
        kq.tu = a.tu + 1;
        kq.mau = a.mau + 1;
        return kq.RutGon();
    }
    public static PhanSo operator --(PhanSo a)
    {
        PhanSo kq = new PhanSo();
        kq.tu = a.tu--;
        kq.mau = a.mau--;
        return kq;
    }
    //Chuyển số nguyên thành phân số
    public static implicit operator PhanSo(int a)
    {
        PhanSo kq = new PhanSo();
        kq.tu = a;
        kq.mau = 1;
        return kq;
    }
    //Chuyển phân số thành số thực
    public static explicit operator double(PhanSo a)
    {
        return ((double)a.tu / (double)a.mau);
    }
}

```

Kiểm tra thực thi quá tải toán tử và ép kiểu

```

PhanSo ps1 = new PhanSo(-16, 5);
Console.WriteLine("Phan so thu nhat 1"+ ps1);
PhanSo ps2 = new PhanSo(20, 2);

```

```
Console.WriteLine("Phan so thu nhat 2 "+ ps2);  
Console.WriteLine("Cong phan so mot va hai "+ (ps1 + ps2));
```

4.5 Ứng dụng minh họa

Phát triển ứng dụng trên thành ứng dụng cho phép thực hiện các yêu cầu nhập vào một mảng ngẫu nhiên các phân số, tính tổng các phân số và tìm phân số có giá trị lớn nhất.

Lớp phân số được định nghĩa trong tập tin PhanSo.cs như sau:

```
1:  class PhanSo  
2:      {  
3:          //Khái bao thành phần dữ liệu  
4:          public int tu;  
5:          public int mau;  
6:          //Khái bao các phương thức  
7:          public void Xuat()  
8:          {  
9:              Console.Write("{0}/{1}", tu, mau);  
10:         }  
11:         public PhanSo()  
12:         {  
13:             tu = 0;  
14:             mau = 1;  
15:         }  
16:         public PhanSo(int t, int m)  
17:         {  
18:             tu = t;  
19:             mau = m;  
20:         }  
21:         public PhanSo(PhanSo a)  
22:         {  
23:             tu = a.mau;  
24:             mau = a.tu;  
25:         }  
26:         public PhanSo Cong(PhanSo a, PhanSo b)  
27:         {  
28:             PhanSo kq = new PhanSo();  
29:             kq.tu = a.tu * b.mau + a.mau * b.tu;  
30:             kq.mau = a.mau * b.mau;  
31:             return kq;  
32:         }  
33:         public int UCLN(int a, int b)  
34:         {  
35:             a = Math.Abs(a);
```

```

36:         a = Math.Abs(b);
37:         while (a != b)
38:         {
39:             if (a > b)
40:                 a = a - b;
41:             if (a < b)
42:                 b = b - a;
43:         }
44:         return a;
45:     }
46:     public PhanSo RutGon(PhanSo a)
47:     {
48:         int uc = UCLN(a.tu, a.mau);
49:         PhanSo kq = new PhanSo();
50:         kq.tu = a.tu / uc;
51:         kq.mau = a.mau / uc;
52:         return kq;
53:     }
54:     public PhanSo RutGon()
55:     {
56:         int uc = UCLN(this.tu, this.mau);
57:         this.tu = this.tu / uc;
58:         this.mau = this.mau / uc;
59:         return this;
60:     }
61:     public PhanSo Cong(PhanSo b)
62:     {
63:         PhanSo kq = new PhanSo();
64:         kq.tu = this.tu * b.mau + this.mau * b.tu;
65:         kq.mau = this.mau * b.mau;
66:         return kq;
67:     }
68:     public int SoSanh(PhanSo b)
69:     {
70:         int x = this.tu * b.mau;
71:         int y = this.mau * b.tu;
72:         return x.CompareTo(y);
73:     }
74: }

```

Phương thức SoSanh dùng để so sánh hai phân số; nếu phân số b nhỏ hơn phân số cần so sánh thì phương thức trả về 1; nếu bằng nhau trả về 0; ngược lại trả về -1;

Lớp mảng phân số được định nghĩa trong tập tin MangPhanSo.cs như sau:

```
1: class MangPhanSo
2: {
3:     public PhanSo[] ds = new PhanSo[100];
4:     public int len = 0;
5:     public void Them(PhanSo p)
6:     {
7:         ds[len++] = p;
8:     }
9:     public void NhapNgauNhien()
10:    {
11:        Console.Write("Nhap vao chieu dai mang ");
12:        int n = int.Parse(Console.ReadLine());
13:        Random r = new Random();
14:        for (int i = 0; i < n; i++ )
15:        {
16:            Them(new PhanSo(r.Next(20), r.Next(1,20)));
17:        }
18:    }
19:    public void Xuat()
20:    {
21:        Console.WriteLine("Mang phan so ");
22:        for (int i = 0; i < len; i++)
23:            ds[i].Xuat();
24:    }
25:    public PhanSo Tong()
26:    {
27:        PhanSo tong = new PhanSo();
28:        for (int i = 0; i < len; i++)
29:            tong = tong.Cong(ds[i]);
30:        return tong;
31:    }
32:    public PhanSo TimMax()
33:    {
34:        PhanSo max = ds[0];
35:        for (int i = 1; i < len; i++)
36:            if(max.SoSanh(ds[i])== -1)
37:                max = ds[i];
38:        return max;
39:    }
40: }
```

Chương trình kiểm tra:

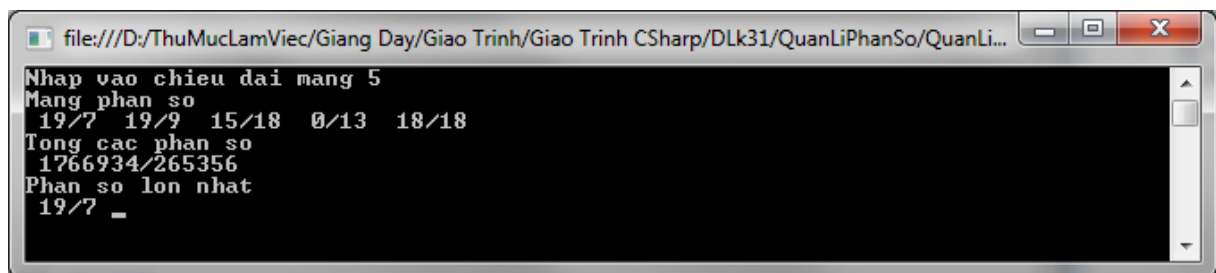
```
1: class Program
2: {
3:     static void Main(string[] args)
```

```

4:      {
5:          MangPhanSo a = new MangPhanSo();
6:          a.NhapNgauNhien();
7:          a.Xuat();
8:          Console.WriteLine("\nTong cac phan so ");
9:          a.Tong().Xuat();
10:         Console.WriteLine("\nPhan so lon nhat ");
11:         a.TimMax().Xuat();
12:         Console.ReadKey();
13:     }
14:
15: }

```

Kết quả thực hiện chương trình:



Nhận xét:

Trong ví dụ trên chúng ta thấy thuật toán tính tổng và tìm phân số lớn nhất tương tự như thuật toán tính tổng và tìm các số nguyên lớn nhất trong ví dụ chương trước là vì mảng phân số được tổ chức theo cấu trúc dữ liệu mảng một chiều. Do đó toàn bộ các thuật toán mảng một chiều đều có thể được áp dụng cho mảng phân số.

4.5 Bài tập

Thực hiện các yêu cầu sau trong lớp MangPhanSo

Phương thức	Mục đích	Trả về
PhanSo TimPhanSoLonNhat()	Tìm phân số lớn nhất trong mảng	
PhanSo TimPhanSoNhoNhat()	Tìm phân số nhỏ nhất trong mảng	
void Them(PhanSo x)	Thêm phân số x vào cuối mảng	

	phân số	
MangPhanSo TimPhanSoTheoMau(int x)	Tất cả các phân số có mẫu bằng x	
int XoaPhanSoTheoMau(int x)	Xóa phân số đầu tiên trong mảng có mẫu là x	0 nếu không xóa được và ngược lại, trả về 1
int TimPhanSo(PhanSo x)	Tìm vị trí đầu tiên của phân số x trong mảng	1: Tìm thành công, 0 không tìm thấy
int XoaTaiViTri(int vt)	Xóa mảng phân số tại vị trí x	0: nếu không xóa, ngược lại trả về 1
int XoaTatCaPhanSo(PhanSo x)	Xóa tất cả phân số x trong mảng	1: xóa thành công, ngược lại: 0
void CapNhat(int x, int y)	Cập nhật tất cả các phân số có tử là x thành y	
void SapXepTang()	Sắp xếp tăng mảng	
void SapXepTangTheoTu()	Sắp xếp tăng theo tử số	
void SapXepTangTheoMau()	Sắp xếp tăng theo mẫu số	
void ThucDon()	Định nghĩa thực đơn gọi các chức năng trên	
int DemPhanSoDuong()	Đếm số phân số dương	

Chương 5: Sự kế thừa

Mục đích của chương:

- Cung cấp kiến thức về sử dụng tính đóng gói trong chương trình.
- Sử dụng tính kế thừa và đa hình trong chương trình.
- Sử dụng giao tiếp (interface) trong quá trình phát triển ứng dụng.
- Nạp chồng toán hạng: kiểm tra cách để định nghĩa những toán hạng cho lớp.
- Chỉ mục: cho phép một lớp được xử lý chỉ mục khi nó là một mảng và đơn giản hoá cách sử dụng những lớp chứa các tập đối tượng.
- Giao diện: C# hỗ trợ kế thừa thông qua giao diện.

Xét ví dụ như sau: xây dựng chương trình nhập vào các đối tượng hình vuông và hình tròn. Tính tổng diện tích của các hình. Chúng ta có thể viết chương trình như sau:

Ví dụ 5.1: Chương trình quản lý các đối tượng hình học

```
1:  class HìnhVuong
2:  {
3:      float canh;
4:      public HìnhVuong(){}
5:      public HìnhVuong(float c) {
6:          canh = c;
7:      }
8:      public float TinhDT()
9:      {
10:         return canh * canh;
11:      }
12:      public void Xuat()
13:      {
14:         Console.WriteLine("Hình vuông canh {0} có diện
15:                             tích {1} ", canh, TinhDT());
16:      }
17:  }
18:  class HìnhTron
19:  {
20:      float r;
21:      public HìnhTron() { }
22:      public HìnhTron(float c)
23:      {
```

```

24:         r = c;
25:     }
26:     public float TinhDT()
27:     {
28:         return (float)Math.PI * r * r;
29:     }
30:     public void Xuat()
31:     {
32:         Console.WriteLine("Hinh tron bk {0} co dien tich
33:                             {1} ", r, TinhDT());
34:     }
35:     class Program
36:     {
37:         static void Main(string[] args)
38:         {
39:             HinhTron[] mt = { new HinhTron(4),
40:                             new HinhTron(5),
41:                             new HinhTron(3) };
42:             HinhVuong[] mv = { new HinhVuong(4),
43:                               new HinhVuong(5) };
44:             float tong = 0;
45:             for (int i = 0; i < mt.Length; i++ )
46:             {
47:                 tong += mt[i].TinhDT();
48:             }
49:             for (int i = 0; i < mv.Length; i++)
50:             {
51:                 tong += mv[i].TinhDT();
52:             }
53:             Console.WriteLine("Tong dien tich cac hinh " + tong);
54:             Console.ReadKey();
55:         }
56:     }

```

Kết quả thực hiện chương trình:

Tong dien tich cac hinh 198.0796

Trong ứng dụng trên chúng ta phải sử dụng 2 mảng một chiều mt, mv để lưu trữ các hình vuông và hình tròn độc lập. Do đó các thuật toán xử lý phải dựa hoàn toàn vào cấu trúc này. Việc tính tổng diện tích các hình được thực hiện khá đơn giản. Tuy nhiên với cấu trúc tổ chức chương trình như vậy, điều gì xảy ra nếu chúng ta muốn hiển thị các hình trên theo chiều tăng của diện tích? Để giải quyết vấn đề này chúng ta có thể áp dụng tính chất kế thừa và đa hình trong lập trình hướng đối tượng.

5.1 Các kiểu kế thừa

a) Sự kế thừa (Inheritance) là gì?

Kế thừa trong ngôn ngữ lập trình liên quan sự phân lớp, đó chính là mối quan hệ giữa các lớp. Ví dụ, bò hay ngựa là những động vật có vú. Bò và ngựa có mọi đặc trưng mà động vật có vú có nhưng nó cũng có riêng những đặc trưng của từng loài.

Trong C#, chúng ta có thể mô hình điều này bằng cách tạo hai lớp, một lớp tên `DongVat` và một lớp tên `Ngua` và khai báo `Ngua` kế thừa từ `DongVatCoVu`. Sự kế thừa sẽ mô hình rằng có một quan hệ và mô tả sự kiện rằng tất cả ngựa là động vật có vú. Tương tự, chúng ta có thể khai báo một lớp tên `Bo` cũng kế thừa từ `DongVatCoVu`. Những đặc trưng chung (như ngày sinh, cân nặng) được đặt trong lớp `DongVatCoVu`. Thuộc tính riêng của từng động vật được đặt tương ứng trong lớp `Ngua` và `Bo`.

c) Lớp cơ sở (base) và lớp dẫn xuất (Derived)

Cú pháp khai báo một lớp kế thừa từ một lớp khác như sau:

```
class Lớp_Dẫn_Xuất: Lớp_Cơ_Sở {  
    ...  
}
```

Lớp dẫn xuất kế thừa từ lớp cơ sở. Trong C# chỉ cho một lớp kế thừa từ một lớp khác duy nhất, nó không cho dẫn xuất từ hai hay nhiều lớp. Trừ khi lớp dẫn xuất được khai báo là `sealed`, chúng ta vẫn có thể khai báo một lớp kế thừa từ lớp dẫn xuất với cùng cú pháp:

```
class Lớp_Con_Dẫn_Xuất: Lớp_Dẫn_Xuất {  
    ...  
}
```

Theo cách này chúng ta có thể tạo một phân cấp kế thừa.

Giả sử chúng ta viết một trình quản lý các đối tượng hình học. Trong đó chúng ta quản lý các đối tượng như là hình tam giác, hình vuông, hình tròn... Chúng ta có thể khai báo một lớp `DoiTuongHinhHoc` trong đó định nghĩa phương thức `TinhDienTich` như sau:

```
public class DoiTuongHinhHoc
{
    public double TinhDienTich()
    {
        return 0;
    }
}
```

Chúng ta có thể định nghĩa các lớp HìnhTron, HìnhVuong, HìnhTamGiac kế thừa từ lớp DoiTuongHinhHoc như sau:

```
public class HìnhTron: DoiTuongHinhHoc
{
    ...
}
public class HìnhTamGiac: DoiTuongHinhHoc
{
    ...
}
public class HìnhVuong: DoiTuongHinhHoc
{
    ...
}
```

Trong C#, đối tượng System.Object là lớp cha của tất cả các lớp. Tất cả các lớp dẫn xuất ngầm định từ lớp System.Object. Nếu chúng ta thực thi lớp DoiTuongHinhHoc như trên:

Trình biên dịch sẽ tự động viết lại theo mã sau:

```
class DoiTuongHinhHoc: System.Object
{
    public DoiTuongHinhHoc()
    {
        ...
    }
    ...
}
```

Nghĩa là tất cả các lớp chúng ta định nghĩa kế thừa tất cả các đặc trưng của lớp System.Object. Điều này bao gồm phương thức ToString để chuyển một đối tượng sang một chuỗi.

Trong C#, quan hệ chuyên biệt hóa thông thường được thực hiện thông qua sự kế thừa. Đây không phải là cách duy nhất để thực hiện sự kế thừa. Tuy nhiên nó là các phổ biến và tự nhiên để thực hiện quan hệ này.

Chúng ta nói HìnhTron kế thừa từ DoiTuongHinhHoc có nghĩa nó là một DoiTuongHinhHoc chuyên biệt. DoiTuongHinhHoc được gọi là một lớp cơ sở (base) và HìnhTron gọi là lớp dẫn xuất (derived). Thật vậy, HìnhTron kế thừa những đặc trưng và hành vi của lớp DoiTuongHinhHoc và rồi chuyên biệt hóa những nhu cầu riêng của nó.

5.2 Thực thi sự kế thừa

Trong C# chúng ta có thể tạo một lớp dẫn xuất bằng cách thêm dấu hai chấm theo sau tên của lớp dẫn xuất cùng với tên của lớp cơ sở.

```
public class HìnhTron: DoiTuongHinhHoc
```

Trong khai báo trên, khai báo HìnhTron là lớp dẫn xuất từ DoiTuongHinhHoc. Lớp dẫn xuất kế thừa tất cả thành viên của lớp cơ sở. Lớp dẫn xuất có thể tự do thực hiện những phiên bản riêng của mình đối với các phương thức lớp cơ sở. Nó thực hiện điều này bằng cách đánh dấu phương thức mới bằng từ khóa new. Dùng từ khóa này để chỉ ra lớp dẫn xuất muốn che dấu và thay thế phương thức ở lớp cơ sở.

Ví dụ sau minh họa sử dụng lớp dẫn xuất:

```
1: public class DoiTuongHinhHoc
2:     {
3:         public float X;
4:         public float Y;
5:         public DoiTuongHinhHoc() { }
6:         public DoiTuongHinhHoc(float x, float y) {
7:             this.X = x;
8:             this.Y = y;
9:         }
10:        public double TinhDienTich()
11:        {
12:            return 0;
13:        }
14:        public void Ve()
```

```

15:     {
16:         Console.WriteLine("Ve tai vi tri {0} {1} ", X, Y);
17:     }
18: }
19: public class HìnhTron:DoiTuongHìnhHoc
20: {
21:     public HìnhTron(){}
22:     public HìnhTron(float x,float y):base(x,y)
23:     {
24:     }
25:     public new void Ve()
26:     {
27:         base.Ve();
28:     }
29: }
30: class ViDu
31: {
32:     static void Main()
33:     {
34:         DoiTuongHìnhHoc h = new DoiTuongHìnhHoc(10, 10);
35:         h.Ve();
36:         HìnhTron r = new HìnhTron(20, 20);
37:         r.Ve();
38:     }
39: }

```

a) Gọi phương thức tạo lập của lớp cơ sở

Trong ví dụ trên lớp HìnhTron dẫn xuất từ lớp DoiTuongHìnhHoc và có phương thức tạo lập riêng của nó với hai tham số. Phương thức tạo lập của lớp HìnhTron gọi phương thức tạo lập của lớp cha bằng cách đặt dấu “:” sau danh sách tham số và gọi lớp cơ sở với từ khoá base:

```
public HìnhTron(float x,float y):base(x,y) // một cách thức khác để gọi phương thức tạo lập của lớp cơ sở
```

Bởi vì các lớp không thể kế thừa phương thức tạo lập của lớp cơ sở nên một lớp dẫn xuất phải thực thi phương thức tạo lập riêng của mình và chúng có thể sử dụng phương thức tạo lập của lớp cơ sở theo cách rõ ràng như trên

Chú ý trong ví dụ trên lớp dẫn xuất HìnhTron thực thi một phiên bản mới của phương thức Ve();

```
public new void Ve();
```

Từ khoá new chỉ ra rằng người lập trình muốn tạo ra một phiên bản mới của phương thức này trong lớp dẫn xuất.

Nếu lớp cơ sở có một phương thức tạo lập có thể truy xuất mặc định. Khi phương thức tạo lập của lớp dẫn xuất không yêu cầu phương thức tạo lập của lớp cơ sở, khi đó phương thức tạo lập mặc định của lớp cơ sở được gọi theo kiểu không tường minh. Tuy nhiên nếu lớp cơ sở không có một phương thức tạo lập mặc định, các lớp dẫn xuất phải gọi rõ ràng một phương thức tạo lập cơ sở sử dụng từ khóa base.

b) Gọi phương thức lớp cơ sở

Trong ví dụ trên phương thức Ve() của lớp HìnhTron ẩn và thay thế phương thức Ve() của lớp cơ sở. Lúc chúng ta gọi phương thức Ve() trong một đối tượng kiểu HìnhTron thì phương thức HìnhTron.Ve() được gọi chứ không phải phương thức HìnhTron () trong lớp ĐốiTuongHìnhHoc. Tuy nhiên, trong lớp HìnhTron ta vẫn có thể gọi phương thức Ve() của lớp ĐốiTuongHìnhHoc thông qua từ khóa base.

c) Khai báo và sử dụng đối tượng của lớp con

Khi thực hiện lệnh gán giữa các đối tượng cha, con cần chú ý:

- Một đối tượng kiểu cha có thể tham chiếu đến một đối tượng kiểu con nhưng ngược lại thì không được.
- Một kiểu con có thể ép kiểu về kiểu cha nhưng ngược lại thì chưa chắc đúng trong một số trường hợp.

```
1: ĐốiTuongHìnhHoc a = new ĐốiTuongHìnhHoc();
2: HìnhTron b = new HìnhTron();
3: ĐốiTuongHìnhHoc c = new HìnhTron(6);
4: b = a; //sai vì kiểu cha không gán được cho kiểu con
5: c = b; //đúng vì cha có thể tham chiếu đến con
6: b = c; //sai vì con không thể tham chiếu đến cha
7: b = (HìnhTron)c; //đúng
8: b.Ve();
9: b = (HìnhTron)a; //sai khi thực thi chương trình vì a thật sự
                    là ĐốiTuongHìnhHoc
```

10: b.Ve();

Trong ví dụ trên khi thực hiện lệnh gán, chúng ta thấy việc thực thi kế thừa sẽ gây khó khăn khi thực hiện thao tác trên kiểu dữ liệu có kế thừa. Do đó chúng ta cần nắm vững nguyên tắc chuyển đổi kiểu và xem xét cụ thể kiểu dữ liệu thật sự mà đối tượng đang tham chiếu đến.

Bằng cách sử dụng kế thừa, chúng ta có thể cải tiến lại chương trình 5.1 như sau:

Ví dụ 5.2: Sử dụng tính đa hình cho ứng dụng quản lý đối tượng hình học

```
1:  public class DoiTuongHinhHoc
2:  {
3:      public float TinhDT()
4:      {
5:          return 0;
6:      }
7:  }
8:  class HinhVuong: DoiTuongHinhHoc
9:  {
10:     float canh;
11:     public HinhVuong() { }
12:     public HinhVuong(float c)
13:     {
14:         canh = c;
15:     }
16:     public new float TinhDT()
17:     {
18:         return canh * canh;
19:     }
20:     public void Xuat()
21:     {
22:         Console.WriteLine("Hinh vuong canh {0} co dien
                                tích {1} ", canh, TinhDT());
23:     }
24:
25: }
26: class HinhTron: DoiTuongHinhHoc
27: {
28:     float r;
29:     public HinhTron() { }
30:     public HinhTron(float c)
31:     {
32:         r = c;
33:     }
34:     public new float TinhDT()
35:     {
```



```

36:         return (float)Math.PI * r * r;
37:     }
38:     public void Xuat()
39:     {
40:         Console.WriteLine("Hình tron bk {0} co dien tich
                               {1} ", r, TinhDT());
41:     }
42: }
43: class Program
44: {
45:     static void Main(string[] args)
46:     {
47:         DoiTuongHinhHoc[] ds = { new HinhTron(4),
                                   new HinhTron(5), new HinhTron(3),
                                   new HinhVuong(4), new HinhVuong(5) };
48:         float tong = 0;
49:         for (int i = 0; i < ds.Length; i++)
50:         {
51:             if(ds[i] is HinhTron)
52:                 tong += ((HinhTron)ds[i]).TinhDT();
53:             if (ds[i] is HinhVuong)
54:                 tong += ((HinhVuong)ds[i]).TinhDT();
55:         }
56:         Console.WriteLine("Tong dien tich cac hinh " + tong);
57:         Console.ReadKey();
58:     }
59: }
60: }

```

Trong ứng dụng trên, áp dụng nguyên lý kế thừa chúng ta chỉ cần lưu trữ tất cả các đối tượng hình học trong một mảng. Do đó thuật toán tính tổng diện tích các hình đơn giản hơn rất nhiều so với thuật trong ví dụ 5.1. Ngoài ra nếu yêu cầu sắp xếp tăng các hình theo diện tích cũng dễ dàng thực hiện vì đây là thao tác trên mảng 1 chiều. Tuy nhiên khi tổ chức theo cách trên ta thấy ds là một mảng 1 chiều được khai báo là chứa các đối tượng hình học; nhưng trong ví dụ nó thật sự lại chứa hình tròn hay hình vuông. Do đó khi cần thao tác trên các phần tử trong mảng này chúng ta phải biết chính xác kiểu dữ liệu của phần tử nên phải dùng từ khóa `is` để kiểm tra kiểu.

Điều gì xảy ra nếu trong chương trình trên yêu cầu tính thêm các hình chữ nhật, bình hành, thang... Để giải quyết vấn đề này, hướng đối tượng sử dụng tính đa hình.

5.3 Đa hình (polymorphism)

Hai khía cạnh mạnh nhất của kế thừa đó là khả năng sử dụng lại mã và đa hình (polymorphism). Poly có nghĩa là nhiều và morph nghĩa là hình thức. Thật vậy đa hình chính là khả năng sử dụng nhiều hình thức của một kiểu mà không cần quan đến chi tiết của nó.

a) Tạo kiểu đa hình

Bởi vì `HinhTron` là một `DoiTuongHinhHoc` và `HinhVuong` là một `DoiTuongHinhHoc`. Đây là những quan hệ được định nghĩa theo sự kế thừa. Chúng ta mong muốn có thể sử dụng một trong hai kiểu này trong tình huống chúng ta gọi `DoiTuongHinhHoc`. Ví dụ, mảng `ds` trong ví dụ 5.2 giữ một tập tất cả các thể hiện của `DoiTuongHinhHoc`. Chúng ta muốn tính tổng diện tích của tất cả các hình có trong mảng bằng cách lấy từng phần tử trong mảng và cộng vào tổng diện tích. Trong trường hợp này chúng ta không cần biết đối tượng trong danh sách là `HinhTron` hay `HinhVuong`, chúng ta chỉ lấy ra một phần tử và yêu cầu nó tự tính diện tích. Chúng ta sẽ xử lý tất cả các đối tượng `DoiTuongHinhHoc` theo đặc trưng đa hình.

b) Tạo phương thức đa hình

Để tạo phương thức hỗ trợ đa hình, chúng ta cần đánh dấu dùng từ khóa `virtual` trong lớp cơ sở của nó. Ví dụ để chỉ ra phương thức `TinhDT()` trong lớp `DoiTuongHinhHoc` là đa hình chúng ta dùng từ khóa theo khai báo sau:

```
public virtual float TinhDT()  
{  
    return 0;  
}
```

Bây giờ các lớp dẫn xuất tự do thực hiện các phiên bản riêng của lớp `TinhDT()`. Để thực hiện điều này chúng ta đơn giản viết đè lên phương thức ảo của lớp cơ sở dùng từ khóa `override` trong lớp dẫn xuất `HinhTron`. Ví dụ:

```
public override float TinhDT()  
{
```

```
        return canh * canh;
    }
```

Tương tự chúng ta có thể thực hiện điều này cho lớp HìnhVuong.

Hàm Main() của chương trình có thể được viết lại như sau

```
DoiTuongHinhHoc[] ds = { new HinhTron(4), new HinhTron(5),
                          new HinhTron(3), new HinhVuong(4),
                          new HinhVuong(5) };

float tong = 0;
for (int i = 0; i < ds.Length; i++)
{
    tong += ds[i].TinhDT();
}
Console.WriteLine("Tong dien tich cac hinh " + tong);
Console.ReadKey();
```

Chương trình có thể thực hiện như ta mong muốn. Phương thức TinhDT được tự động xác định cho các đối tượng sử dụng tính đa hình.

d) Phân biệt giữa từ khóa new và override

Trong C#, quyết định của lập trình viên nhằm ghi đè một phương thức ảo của lớp cơ bản được thực hiện rõ ràng qua từ khóa override. Điều này cho phép chúng ta tạo ra một phiên bản mới trong mã chúng ta. Những thay đổi trong mã của lớp cơ sở không phá vỡ những đoạn mã trong lớp dẫn xuất.

Trong C# một hàm ảo luôn được xem là gốc của các hàm ảo gọi đi. Thật vậy, khi C# tìm một phương thức ảo, nó không tìm kiếm trong phân cấp kế thừa. Nếu một hàm ảo được đưa vào trong lớp DoiTuongHinhHoc, hành vi thực thi của lớp HinhTron vẫn không thay đổi.

Để tránh việc khai báo phương thức ảo bị chồng chéo, chúng ta có thể sử dụng từ khóa new để chỉ ra không có phương thức ghi đè lên phương thức ảo của lớp cơ sở.

e) Lớp trừu tượng (abstract)

Mỗi lớp con của DoiTuongHinhHoc nên thực thi phương thức TinhDT() của riêng nó, nhưng các lớp dẫn xuất không bắt buộc thực hiện điều này trong đoạn chương trình

trên. Để yêu cầu tất cả các lớp dẫn xuất thực thi một phương thức của lớp cơ sở chúng ta cần chỉ ra phương thức đó là trừu tượng (abstract).

Phương thức được khai báo trừu tượng không cần định nghĩa hay khai báo nội dung thực thi của nó. Nó chỉ nhằm tạo một tên phương thức và đánh dấu rằng nó phải được thực thi trong tất cả các lớp dẫn xuất từ nó.

Một lớp cũng có thể được khai báo trừu tượng bằng từ khóa abstract. Một lớp trừu tượng thì không có thể hiện. Trong một lớp trừu tượng vẫn có thể định nghĩa các thành viên bình thường như các lớp khác.

Chúng ta có thể định nghĩa lại lớp *DoiTuongHinhHoc* như sau: `public abstract class DoiTuongHinhHoc`

```
{  
    public abstract float TinhDT();  
}
```

Ví dụ 5.3: Áp dụng sự trừu tượng

```
1: public abstract class DoiTuongHinhHoc  
2: {  
3:     public abstract float TinhDT();  
4: }  
5: class HinhVuong: DoiTuongHinhHoc  
6: {  
7:     float canh;  
8:     public HinhVuong() { }  
9:     public HinhVuong(float c)  
10:    {  
11:        canh = c;  
12:    }  
13:     public override float TinhDT()  
14:    {  
15:        return canh * canh;  
16:    }  
17:     public void Xuat()  
18:    {  
19:        Console.WriteLine("Hinh vuong canh {0} co dien tich {1}  
20:                           ", canh, TinhDT());  
21:    }  
22: }  
23: class HinhTron: DoiTuongHinhHoc  
24: {
```

```
25:         float r;
26:         public HìnhTron() { }
27:         public HìnhTron(float c)
28:         {
29:             r = c;
30:         }
31:         public override float TinhDT()
32:         {
33:             return (float)Math.PI * r * r;
34:         }
35:         public void Xuat()
36:         {
37:             Console.WriteLine("Hình tron bk {0} co dien tich {1} ",
                                r, TinhDT());
38:         }
39:     }
40:     class Program
41:     {
42:         static void Main(string[] args)
43:         {
44:             DoiTuongHinhHoc[] ds = { new HìnhTron(4),
                                        new HìnhTron(5), new HìnhTron(3),
                                        new HìnhVuong(4), new HìnhVuong(5) };
45:             float tong = 0;
46:             for (int i = 0; i < ds.Length; i++)
47:             {
48:                 tong += ds[i].TinhDT();
49:             }
50:             Console.WriteLine("Tong dien tich cac hinh " + tong);
51:             Console.ReadKey();
52:         }
53:     }
```

f) Giới hạn của trừu tượng

Mặc dù phương thức trừu tượng `TinhDT()` bắt buộc tất cả các lớp dẫn xuất thực thi phương thức trừu tượng của nó. Điều này sẽ gây khó khăn nếu lớp dẫn xuất từ lớp `HìnhTron` không muốn thực thi phương thức `Ve()`;

g) Lớp Sealed

Ngược lại của abstract là sealed. Dùng từ khóa sealed, một lớp có thể không cho phép các lớp khác dẫn xuất từ nó.

h) Cha của tất cả các lớp là Object

Tất cả các lớp trong C# đều được dẫn xuất từ lớp Object. Lớp gốc là lớp trên nhất trong cây phân cấp kế thừa. Trong C#, lớp gốc là Object. Object cung cấp một số các phương thức mà các lớp con có thể ghi đè. Chúng gồm các phương thức sau:

Phương thức	Ý nghĩa
Equal()	Kiểm tra hai đối tượng có tương đương nhau không
GetHashCode()	Cho phép đối tượng cung cấp hàm Hash riêng để sử dụng trong kiểu tập hợp.
GetType()	Cung cấp truy xuất đến kiểu đối tượng.
ToString()	Cung cấp chuỗi biểu diễn đối tượng.
Finalize()	Xóa đối tượng trong bộ nhớ.
MemberwiseClone()	Tạo copy của đối tượng.

Ví dụ sau minh họa sử dụng phương thức ToString() trong lớp Object cũng như trong các kiểu dữ liệu cơ bản kế thừa từ Object:

```
1: using System;
2: public class ViDu
3: {
4:     public ViDu (int val)
5:     {
6:         giatri = val;
7:     }
8:     public override string ToString( )
9:     {
10:         return giatri.ToString( );
11:     }
12:     private int giatri;
13: }
14: public class KiemTra
15: {
```

```
16:         static void Main( )
17:         {
18:             int i = 5;
19:             Console.WriteLine("Giá trị của biến i là: " + i));
20:             ViDu s = new ViDu(7);
21:             Console.WriteLine("Giá trị của biến s là " +s );
22:         }
23:     }
```

Trong trường hợp trên hàm ToString() của đối tượng s và i sẽ được gọi tự động.

5.4 Giao diện

Trong C# không cho phép thực hiện đa kế thừa thông qua sử dụng tính kế thừa, đa hình hay trừu tượng. Vì sao??

C# hỗ trợ đa kế thừa thông qua thực thi giao diện (interface). Khi một lớp kế thừa từ một giao diện, một lớp dẫn xuất sẽ phải thực thi tất cả các thành viên trong giao diện. Ví dụ định nghĩa giao diện như sau:

```
public interface IHìnhHoc
{
    float DienTich();
}
```

Trên ví dụ trên ta thấy việc khai báo một giao diện làm việc giống như việc khai báo một lớp trừu tượng, nhưng nó không cho phép thực thi bất kỳ một thành phần nào của giao diện. Một giao diện chỉ có thể chứa những khai báo của phương thức, thuộc tính, chỉ mục và sự kiện.

Chúng ta không thể khởi tạo một giao diện, nó không có phương thức tạo lập hay các trường và nó không cho phép chứa các phương thức ghi chồng.

Nó cũng không cho phép khai báo những bổ từ trên các thành phần trong khi định nghĩa một giao diện. Các thành phần bên trong một giao diện luôn luôn là public và không thể khai báo virtual hay static.

a) Định nghĩa và thực thi một giao diện

Giả sử chúng ta viết mã để cho phép các máy điện toán chuyển khoản qua lại giữa các tài khoản. Có nhiều công ty tham gia vào hệ thống này tài khoản và đều đồng ý với nhau là phải thực thi một giao diện ITaiKhoan có các phương thức nạp hay rút tiền và thuộc tính trả về số tài khoản.

Để bắt đầu, ta định nghĩa giao diện ITaiKhoan:

```
public interface ITaiKhoan
{
    void GuiTien(decimal soLuong);
    bool RutTien(decimal soLuong);
    decimal SoTien
    {
        get;
    }
}
```

Chú ý: Tên của giao diện thường phải có ký tự I đứng đầu để nhận biết đó là một giao diện.

Khai báo lớp TaiKhoanTietKiem kế thừa từ ITaiKhoan:

```
public class TaiKhoanTietKiem: ITaiKhoan
{
    private decimal soTien;
    public void GuiTien (decimal soLuong)
    {
        soTien += soLuong;
    }
    public bool RutTien(decimal soLuong)
    {
        if (soTien >= soLuong)
        {
            soTien -= soLuong;
            return true;
        }
        Console.WriteLine("Rut tien bi loi. ");
        return false;
    }
    public decimal SoTien
    {
        get
```



```
        {  
            return soTien;  
        }  
    }  
    public override string ToString()  
    {  
        return String.Format("Tien tiet kiem: so tien = {0, 6:C}",  
                               soTien);  
    }  
}
```

Trong ví dụ trên chúng ta duy trì một trường private `soTien` và điều chỉnh số lượng này khi tiền được nạp hay rút. Chú ý chúng ta xuất ra một thông báo lỗi khi thao tác rút tiền không thành công vì thiếu số tiền trong tài khoản.

Xét dòng lệnh sau:

```
public class TaiKhoanTietKiem: ITaiKhoan
```

Chúng ta khai báo lớp `TaiKhoanTietKiem` kế thừa giao diện `ITaiKhoan`. Ta cũng có thể khai báo một lớp kế thừa từ một lớp nào đó và từ nhiều giao diện theo cú pháp như sau:

```
public class Lớp_Dẫn_Xuất: Lớp_Cơ_Sở, IGiaoDien1, IGiaoDien2
```

Thừa kế từ `ITaiKhoan` có nghĩa là `TaiKhoanTietKiem` lấy tất cả các thành phần của `ITaiKhoan` nhưng nó không thể sử dụng các phương thức đó nếu nó không định nghĩa lại các hành động của từng phương thức. Nếu bỏ quên một phương thức nào thì trình biên dịch sẽ báo lỗi.

Để minh họa cho các lớp khác nhau có thể thực thi cùng một giao diện ta xét ví dụ về một lớp khác là `TaiKhoanVang`.

```
public class TaiKhoanVang: ITaiKhoan  
{  
    // vv  
}
```

Chúng ta không mô tả chi tiết lớp `TaiKhoanVang` bởi vì về cơ bản nó giống hệt lớp `TaiKhoanTietKiem`. Điểm nhấn mạnh ở đây là `TaiKhoanVang` có thể rút tiền thậm chí trong tài khoản của họ không còn tiền.

Chúng ta có phương thức main():

```
class ViDu
{
    static void Main()
    {
        ITaiKhoan tk1 = new TaiKhoanTietKiem();
        ITaiKhoan tk2 = new TaiKhoanVang();
        tk1.GuiTien(200);
        tk1.RutTien(100);
        Console.WriteLine(tk1.ToString());
        tk2. GuiTien(500);
        tk2. RutTien(600);
        tk2. RutTien(100);
        Console.WriteLine(tk2.ToString());
    }
}
```

Kết quả xuất ra là:

Tien tiet kiem: so tien = £100. 00

Rut tien bi loi.

Tien tiet kiem: so tien = £400. 00

Chúng ta có thể trở đến bất kỳ thể hiện của bất kỳ lớp nào thực thi cùng một giao diện. Nhưng chúng ta chỉ được gọi những phương thức là thành phần của giao diện thông qua sự tham khảo đến giao diện này. Nếu muốn gọi những phương thức mà không là thành phần trong giao diện thì ta phải tham khảo đến những kiểu thích hợp. Như ví dụ trên ta có thể thực thi phương thức ToString() mặc dù nó không là thành phần được khai báo trong giao diện ITaiKhoan bởi vì nó là thành phần của System.Object.

Một giao diện có thể tham khảo đến bất kỳ lớp nào thực thi giao diện đó.

Ví dụ ta có một mảng kiểu giao diện nào đó thì các phần tử của mảng có thể tham khảo đến bất kỳ lớp nào thực thi giao diện đó:

```
ITaiKhoan[] taiKhoan = new ITaiKhoan[2];
taiKhoan[0] = new TaiKhoanTietKiem();
taiKhoan[1] = new TaiKhoanVang();
```

b) Kế thừa giao diện

C# cho phép một giao diện có thể kế thừa các giao diện khác. Khi một giao diện kế thừa một giao diện khác thì nó có thể chứa tất cả các phương thức định nghĩa trong giao diện cha và những phương thức của nó định nghĩa. Ví dụ ta tạo ra một giao diện mới kế thừa giao diện ITaiKhoan:

```
public interface ITaiKhoanChuyenTien: ITaiKhoan
{
    bool ChuyenDen(ITaiKhoan dich, decimal soLuong);
}
```

Như vậy giao diện ITaiKhoanChuyenTien phải có tất cả các phương thức trong giao diện ITaiKhoan và phương thức ChuyenDen.

Chúng ta sẽ minh họa ITaiKhoanChuyenTien bằng một ví dụ bên dưới về một tài khoản hiện thời. Lớp TaiKhoanHienThoi được định nghĩa gần giống hết với các lớp TaiKhoanTietKiem và TaiKhoanVang:

```
1: public class TaiKhoanHienThoi: ITaiKhoanChuyenTien
2: {
3:     private decimal soTien;
4:     public void GuiTien(decimal soLuong)
5:     {
6:         soTien += soLuong;
7:     }
8:     public bool RutTien(decimal soLuong)
9:     {
10:         if (soTien >= soLuong)
11:         {
12:             soTien -= soLuong;
13:             return true;
14:         }
15:         Console.WriteLine("Rut tien loi. ");
16:         return false;
17:     }
18:     public decimal SoTien
19:     {
20:         get
21:         {
22:             return soTien;
23:         }
24:     }
25: }
```

```

24:     }
25:     public bool ChuyenDen(ITaiKhoan dich, decimal soLuong)
26:     {
27:         bool kq;
28:         if ((kq = RutTien(soLuong)) == true)
29:             dich.GuiTien(soLuong);
30:         return kq;
31:     }
32:     public override string ToString()
33:     {
34:         return String.Format("Tai khoan hien thoi: so tien =
                                {0, 6:C}", soTien);
35:     }
36: }
37: static void Main()
38: {
39:     ITaiKhoan tk1 = new TaiKhoanTietKiem();
40:     ITaiKhoanChuyenTien tk2 = new TaiKhoanHienThoi();
41:     tk1.GuiTien(200);
42:     tk2.GuiTien(500);
43:     tk2.ChuyenDen(tk1, 100);
44:     Console.WriteLine(tk1.ToString());
45:     Console.WriteLine(tk2.ToString());
46: }

```

Khi thực thi đoạn mã trên chúng ta sẽ thấy kết quả như sau:

```

TaiKhoanHienThoi
Tien tiet kiem: so tien = £300. 00
Tai khoan hien thoi: so tien = £400. 00

```

5.5 Ứng dụng minh họa

Viết chương trình quản lý giá của thiết bị máy tính. Mỗi máy tính được lắp ghép từ nhiều thiết bị khác nhau. Tìm máy tính có giá cao nhất?

Đối với mỗi máy tính chúng ta thật sự chưa biết được các thành phần thiết bị bên trong. Do đó chúng ta phải tổ chức dữ liệu sao cho thuật toán có thể tính toán nhưng chưa liên quan đến thiết bị cụ thể trong từng máy tính. Chúng ta định nghĩa một giao tiếp `IThietBi` như sau:

```

public interface IThietBi
{
    float Gia

```

```
    {  
        get;  
        set;  
    }  
}
```

Chúng ta xây dựng một lớp máy tính như sau:

```
1:  public class MayTinh  
2:      {  
3:          IThietBi[] ds = new IThietBi[100];  
4:          int len = 0;  
5:          public void Them(IThietBi tb)  
6:          {  
7:              ds[len++] = tb;  
8:          }  
9:          public float TongGia()  
10:         {  
11:             float tong = 0;  
12:             for (int i = 0; i < len; i++ )  
13:             {  
14:                 tong += ds[i].Gia;  
15:             }  
16:             return tong;  
17:         }  
18:     }
```

Trong lớp MayTinh, ta thấy thuật toán tính giá của máy tính không cần quan tâm tới kiểu thiết bị cụ thể nào (sử dụng giao diện).

Xây dựng lớp CPU và RAM

```
1:  public class CPU : IThietBi  
2:      {  
3:          float gia;  
4:          public float Gia  
5:          {  
6:              get  
7:              {  
8:                  return gia;  
9:              }  
10:             set  
11:             {  
12:                 gia = value;  
13:             }  
14:         }
```

```
15:         public CPU() { }
16:         public CPU(float g) { Gia = g; }
17:         public override string ToString()
18:         {
19:             return "CPU co gia " + Gia;
20:         }
21:     }
22: }
1: public class Ram:IThietBi
2: {
3:     float gia;
4:     public float Gia
5:     {
6:         get
7:         {
8:             return gia;
9:         }
10:        set
11:        {
12:            gia = value;
13:        }
14:    }
15:    public Ram(){ }
16:    public Ram(float g) { Gia = g; }
17:    public override string ToString()
18:    {
19:        return "Ram co gia " + Gia;
20:    }
21: }
22: }
```

Lớp DanhSachMayTinh chứa các máy tính có nội dung như sau:

```
1: using System;
2: using System.Collections.Generic;
3: using System.Text;
4:
5: namespace GiaMayTinh
6: {
7:     class DanhSachMayTinh
8:     {
9:         public MayTinh[] ds = new MayTinh[100];
10:        int len = 0;
11:        public void Nhap()
12:        {
13:            MayTinh acer = new MayTinh();
```

```
14:         acer.Them(new Ram(50000));
15:         acer.Them(new Ram(50000));
16:         acer.Them(new CPU(100000));
17:         Them(acer);
18:         Them(acer);
19:         MayTinh acer1 = new MayTinh();
20:         acer1.Them(new Ram(5000));
21:         acer1.Them(new Ram(15000));
22:         acer1.Them(new CPU(50000));
23:         MayTinh acer2 = new MayTinh();
24:         acer2.Them(new Ram(50000));
25:         acer2.Them(new Ram(15000));
26:         acer2.Them(new CPU(150000));
27:         MayTinh acer3 = new MayTinh();
28:         acer3.Them(new Ram(50000));
29:         acer3.Them(new Ram(15000));
30:         acer3.Them(new CPU(150000));
31:         Them(acer1);
32:         Them(acer2);
33:         Them(acer3);
34:
35:     }
36:     public void Them(MayTinh tb)
37:     {
38:         ds[len++] = tb;
39:     }
40:     public float GiaMax()
41:     {
42:         float max = ds[0].TongGia();
43:         for (int i = 1; i < len; i++)
44:         {
45:             float tam = ds[i].TongGia();
46:             if (max < tam)
47:                 max = tam;
48:         }
49:         return max;
50:     }
51:     public DanhSachMayTinh MayTinhGiaCaoNhat()
52:     {
53:         DanhSachMayTinh kq = new DanhSachMayTinh();
54:         float max = GiaMax();
55:         for (int i = 1; i < len; i++)
56:         {
57:             if (ds[i].TongGia() == max)
58:                 kq.Them(ds[i]);
59:         }
```

```
60:         return kq;
61:     }
62:     public override string ToString()
63:     {
64:         string s = "\nDanh sach may tinh ";
65:         for (int i = 0; i < len; i++)
66:         {
67:             s += "\n" + ds[i];
68:         }
69:         return s;
70:     }
71: }
72: }
```

Mã kiểm tra chương trình

```
1: using System;
2: using System.Collections.Generic;
3: using System.Text;
4: namespace GiaMayTinh
5: {
6:     class Program
7:     {
8:         static void Main(string[] args)
9:         {
10:             DanhSachMayTinh ds = new DanhSachMayTinh();
11:             ds.Nhap();
12:             Console.WriteLine("Danh sach may tinh gia cao
nhat " + ds.MayTinhGiaCaoNhat());
13:             Console.ReadKey();
14:         }
15:     }
16: }
```

Kết quả thực hiện:

Danh sach may tinh gia cao nhat

Danh sach may tinh

Ram co gia 50000Ram co gia 15000CPU co gia 150000

Ram co gia 50000Ram co gia 15000CPU co gia 150000

Trong ứng dụng trên cho thấy được sự linh động và hữu dụng khi các ứng dụng áp dụng được tính kế thừa và đa hình.

Thực hiện bổ sung các yêu cầu sau cho:

1. Hiển thị máy tính thao chiều tăng của giá, giá RAM, giá CPU.

2. Tìm các máy tính có giá RAM cao nhất, thất nhấp, CPU thấp nhất, CPU cao nhất.
 3. Tìm các máy tính có nhiều linh kiện nhất.
 4. Tìm các máy tính có linh kiện theo hãng.
 5. Mở rộng ứng dụng trên cho quản lý thêm phần mềm trong máy tính.
-

Chương 6: Sự ủy nhiệm, sự kiện và quản lý lỗi

Mục đích của chương:

- Gọi phương thức gần giống với con trỏ hàm trong C++.
- Khai báo và sử dụng sự kiện.
- Cơ chế quản lý lỗi, bắt lỗi chương trình.
- Xây dựng ứng dụng minh họa sử dụng delegate và event.

6.1 Sự ủy nhiệm (delegate)

Nhiều đoạn mã mà chúng ta đã viết trong các bài tập trước được giả định thực thi tuần tự. Nhưng thỉnh thoảng chúng ta thấy cần thiết ngắt luồng thực thi hiện tại và thực hiện các nhiệm vụ khác quan trọng hơn. Lúc nhiệm vụ hoàn thành, chương trình có thể tiếp tục quay lại nơi nó gọi thực thi nhiệm vụ.

Để điều khiển các ứng dụng dạng này, CLR phải cung cấp 2 điều: một phương tiện để chỉ ra có một cái gì đó khẩn cấp đã xảy ra và một điều chỉ ra phương thức nào nên chạy khi nó xảy ra. Đây là mục đích của ủy nhiệm và sự kiện (event).

a) Khai báo và sử dụng delegate

Một delegate là một con trỏ đến một phương thức. Một delegate trông giống và cư xử như phương thức lúc nó được gọi. Tuy nhiên, khi gọi một delegate, CLR thực thi phương thức mà delegate tham chiếu tới. Chúng ta có thể thay đổi tham chiếu của delegate một cách tự động vì vậy mã để gọi một delegate chạy các phương thức khác nhau mỗi lần nó thực thi.

Khai báo delegate

```
delegate kiểu_trả_về Tên(danh_sách_tham_số);
```

Ví dụ 6.1: Khai báo và sử dụng delegate

```
1: using System;  
2: delegate void HamDeGoi();  
3: class ViDu
```

```
4: {
5:     public void PhuongThuc()
6:     {
7:         Console.WriteLine("Phuong thuc");
8:     }
9:
10:    public static void staticMethod()
11:    {
12:        Console.WriteLine("Phuong thuc tinh");
13:    }
14: }
15: class MainClass
16: {
17:     static void Main()
18:     {
19:         ViDu t = new ViDu();
20:         HamDeGoi hamDelegate;
21:         hamDelegate = t.PhuongThuc;
22:         hamDelegate += ViDu.staticMethod;
23:         hamDelegate += t.PhuongThuc;
24:         hamDelegate += ViDu.staticMethod;
25:         hamDelegate();
26:     }
27: }
```

Kết quả thực hiện chương trình

Phuong thuc

Phuong thuc tinh

Phuong thuc

Phuong thuc tinh

b) Sử dụng Delegate như là con trỏ hàm

```
1: using System;
2: class MainClass
3: {
4:     delegate int Hamdelegate(string s);
5:     static void Main(string[] args)
6:     {
7:         Hamdelegate del1 = new Hamdelegate(PhuongThuc);
8:         Hamdelegate del2 = new Hamdelegate(PhuongThuc2);
9:         string str = "Xin Chao";
10:        del1(str);
11:        del2(str);
12:    }
13:    static int PhuongThuc(string s)
```

```
14:    {
15:        Console.WriteLine("Phuong Thuc");
16:        return 0;
17:    }
18:    static int PhuongThuc2(string s)
19:    {
20:        Console.WriteLine("Phuong Thuc 2");
21:        return 0;
22:    }
23: }
```

Kết quả thực hiện chương trình

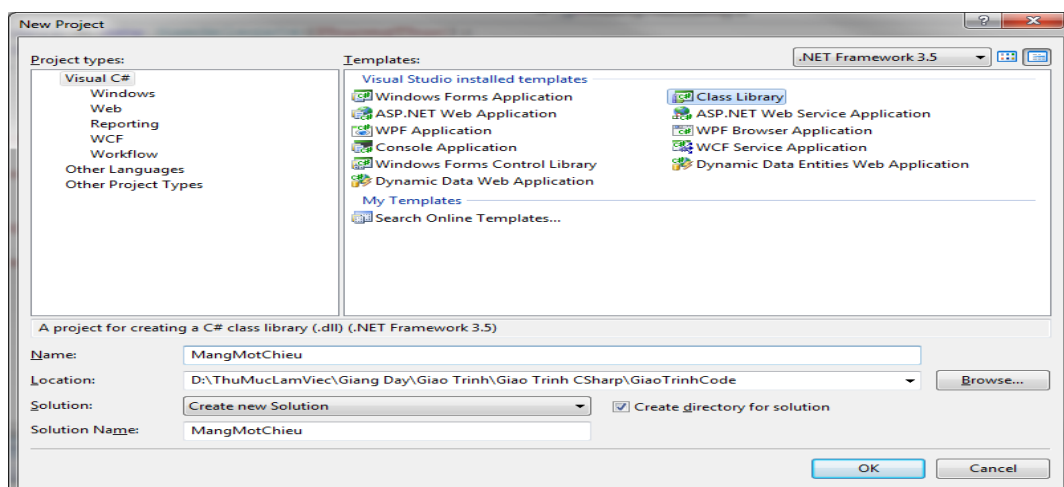
Phuong Thuc

Phuong Thuc 2

c) Ứng dụng minh họa sử dụng delegate

Trong các ví dụ minh họa của chúng ta ở các chương trước, ta thấy chúng ta thường sử dụng cấu trúc mảng một chiều để lưu trữ các đối tượng dữ liệu và thực hiện các thao tác trên cấu trúc dữ liệu này. Câu hỏi đặt ra là chúng ta có thể xây dựng một thư viện mảng một chiều cho phép làm việc với kiểu dữ liệu bất kì bên trong nó hay không? Ví dụ thuật toán sắp xếp các phần tử trong mảng. Để thực hiện điều này chúng ta có thể sử dụng giao tiếp hay delegate. Trong ví dụ này minh họa việc sử dụng delegate nhằm tạo ra một lớp mảng một chiều có thể lưu trữ bất cứ kiểu dữ liệu nào và thực hiện sắp xếp tăng hay giảm tùy theo yêu cầu.

Cách tạo một thư viện liên kết động dll tương tự như cách tạo một ứng dụng kiểu console bằng cách chọn kiểu Class Library như hình sau:

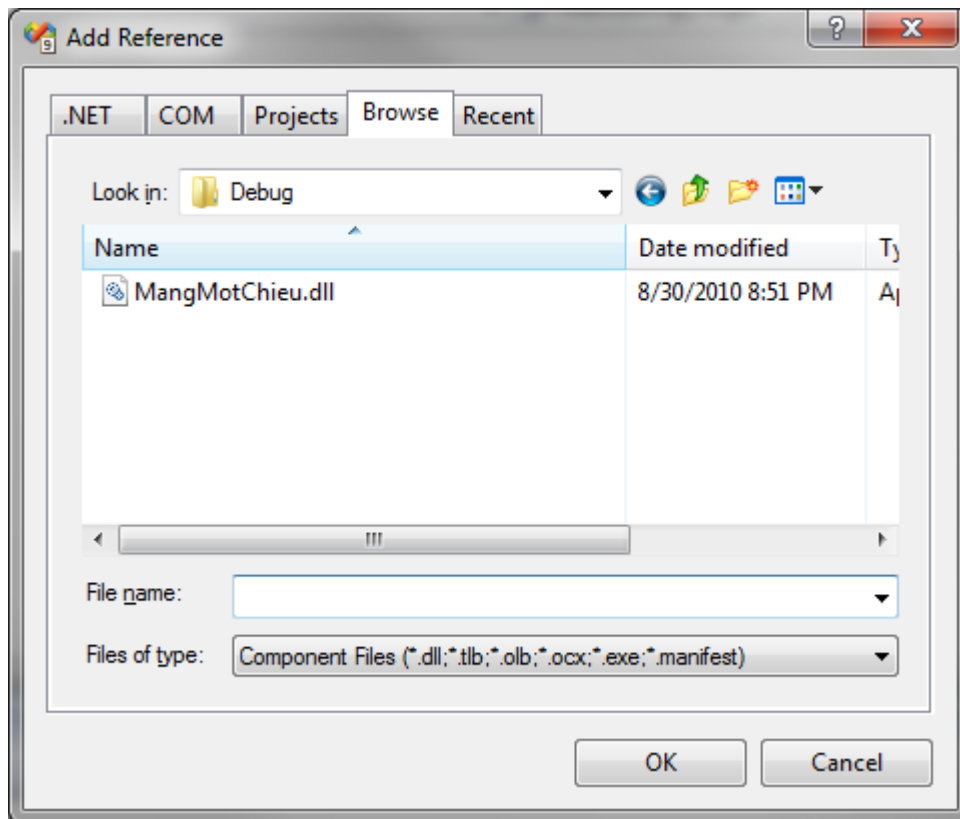


Mã của thư viện mảng 1 chiều như sau:

```
1: namespace MangMotChieu
2: {
3:     public delegate int SoSanh(object x, object y);
4:     public class Mang
5:     {
6:         public object[] ds = new object[100];
7:         public int len = 0;
8:         public void Them(object o)
9:         {
10:             ds[len++] = o;
11:         }
12:         public void SapXep(SoSanh ss)
13:         {
14:             for (int i=0; i< len -1; i++)
15:             {
16:                 for (int j=i+1; j< len; j++)
17:                 {
18:                     if(ss(ds[i],ds[j])==1)
19:                     {
20:                         object tam = ds[i];
21:                         ds[i] = ds[j];
22:                         ds[j] = tam;
23:                     }
24:                 }
25:             }
26:         }
27:         public void Xuat()
28:         {
29:             for (int i = 0; i < len; i++ )
30:             {
31:                 Console.Write(" {0} ", ds[i]);
32:             }
33:         }
34:     }
35: }
```

Nhấn F5 để biên dịch và tạo tập tin dll “MangMotChieu.dll”.

Để liên kết thư viện vào trong ứng dụng cần sử dụng chúng ta thực hiện như sau: chọn Project\Add Reference\ và chọn đường dẫn chứa tập tin dll.



Để sử dụng được thư viện chúng ta cần khai báo chính xác namespace của thư viện mảng 1 chiều.

Để sử dụng mảng 1 chiều cho kiểu số nguyên ta có thể viết hàm như sau:

```
1: using System;
2: using MangMotChieu;
3: class MainClass
4: {
5:     static void Main(string[] args)
6:     {
7:         Mang a = new Mang();
8:         a.Them(1);
9:         a.Them(3);
10:        a.Them(2);
11:        a.Them(5);
12:        a.Them(6);
13:        Console.WriteLine("\n Mảng tăng ");
14:        a.SapXep(new SoSanh(SapTangInt));
15:        a.Xuat();
16:        Console.WriteLine("\n Mảng giảm ");
17:        a.SapXep(new SoSanh(SapGiamInt));
```

```
18:         a.Xuat();
19:         Console.ReadKey();
20:     }
21:     static int SapTangInt(object x, object y)
22:     {
23:         int a = (int)x;
24:         int b = (int)y;
25:         return a.CompareTo(b);
26:     }
27:     static int SapGiamInt(object x, object y)
28:     {
29:         int a = (int)x;
30:         int b = (int)y;
31:         return - a.CompareTo(b);
32:     }
33: }
```

Kết quả của chương trình:

Mang tang

1 2 3 5 6

Mang giam

6 5 3 2 1

Để sử dụng mảng 1 chiều cho kiểu số nguyên ta có thể viết hàm như sau:

```
1: using System;
2: using MangMotChieu;
3: class MainClass
4: {
5:     static void Main(string[] args)
6:     {
7:         Mang a = new Mang();
8:         a.Them("anh");
9:         a.Them("thong");
10:        a.Them("tibo");
11:        a.Them("thanh");
12:        a.Them("xuan");
13:        Console.WriteLine("\n Mang tang ");
14:        a.SapXep(new SoSanh(SapTangChuoi));
15:        a.Xuat();
16:        Console.WriteLine("\n Mang giam ");
17:        a.SapXep(new SoSanh(SapGiamChuoi));
18:        a.Xuat();
19:        Console.ReadKey();
20:    }
21:     static int SapTangChuoi(object x, object y)
```

```
22:     {
23:         string a = (string)x;
24:         string b = (string)y;
25:         return a.CompareTo(b);
26:     }
27:     static int SapGiamChuoai(object x, object y)
28:     {
29:         string a = (string)x;
30:         string b = (string)y;
31:         return - a.CompareTo(b);
32:     }
33: }
```

Kết quả của chương trình:

Mang tang

anh thanh thong tibo xuan

Mang giam

xuan tibo thong thanh anh

Trong các ví dụ trên chúng ta thấy sự thuận lợi khi sử dụng delegate. Chúng ta có thể xử lý thuật toán trong mảng 1 chiều mà không cần quan tâm tới kiểu dữ liệu thật sự được lưu trữ trong mảng. Để sắp xếp chúng ta chỉ cần truyền vào cách so sánh kiểu dữ liệu bằng cách sử dụng delegate. Điều này cho phép các ứng dụng khi sử dụng có thể mở rộng, tùy biến mà không cần bổ sung hay quan tâm tới mã của thư viện nhưng vẫn đảm bảo sự tương thích lúc phát triển chương trình.

6.2 Sự kiện (Event)

Mặc dù delegate cho phép gọi bất kỳ số phương thức gián tiếp, nhưng chúng ta phải gọi các delegate tường minh. Trong nhiều trường hợp, sẽ rất hữu ích nếu delegate chạy tự động khi có một cái gì đó đặc biệt xảy ra. Trong .Net cho phép chúng ta định nghĩa và bất các hành động đặc biệt và sắp xếp để gọi một delegate để quản lý tình huống.

Nhiều lớp trong .NET đưa ra sự kiện. Hầu hết các điều khiển đặt trên form dùng sự kiện cho phép chạy mã chương trình tương ứng như: khi người dùng nhấp chuột vào nút lệnh hay nhập gì đó trên một trường. Chúng ta có thể định nghĩa riêng sự kiện.

a) Khai báo sự kiện

Chúng ta khai báo một sự kiện trong một lớp để nhằm đến một hành động là nguồn của sự kiện. Nguồn sự kiện thường là một lớp quan sát môi trường và phát sinh sự kiện khi có một dấu hiệu đặc biệt xảy ra. Một sự kiện chứa danh sách các phương thức để gọi khi sự kiện được tạo.

Vì sự kiện thường được dùng với delegate nên kiểu của sự kiện phải là delegate và bắt đầu với từ khóa event.

```
public delegate void BatCongTac(bool state);
public class CongTac
{
    public event BatCongTac OnBatCongTac;
}
```

Trong ví dụ trên, sự kiện OnBatCongTac cung cấp một giao tiếp cho phép theo dõi trạng thái của công tắc để thực hiện thao tác tắt mở bóng đèn tương ứng. Một sự kiện quản lý riêng các delegate của nó và không cần quản lý thủ công biến delegate.

b) Gán một sự kiện

Giống delegate, sự kiện có thể được gán với toán tử += .

```
CongTac c = new CongTac();
c.OnBatCongTac += new BatCongTac(d.TrangThaiDen);
```

c) Bỏ gán sự kiện

Chúng ta có thể dùng toán tử -= để xóa phương thức từ tập hợp delegate bên trong.

d) Tạo một sự kiện

Một sự kiện có thể được tạo giống delegate bằng cách gọi nó như là một phương thức. Khi chúng ta gọi một sự kiện, tất cả các delegate gán với nó được gọi tuần tự. Ví dụ:

```
public class CongTac
{
    public event BatCongTac OnBatCongTac;
    public bool state;
    public void KhiBatCongTac()
    {
```

```

        if(OnBatCongTac!=null)
        {
            OnBatCongTac(state);
            state = state ? false: true;
        }
    }
}

```

Kiểm tra null là cần thiết vì trường sự kiện ngầm định là null và nó khác null khi ta thực hiện gán một phương thức dùng toán tử +=. Nếu chúng ta tạo một sự kiện null, chúng ta sẽ có một ngoại lệ `NullReferenceException`. Chúng ta cũng phải truyền các tham số tương ứng khi chúng ta tạo sự kiện ứng với các delegate đã định nghĩa.

Ví dụ 6.2: sử dụng sự kiện và ủy nhiệm:

```

1: using System;
2: using System.Collections.Generic;
3: using System.Text;
4: using System.Windows.Forms;
5: namespace SuDungDelegate
6: {
7:     public delegate void BatCongTac(bool state);
8:     public class CongTac
9:     {
10:         public event BatCongTac OnBatCongTac;
11:         public bool state;
12:         public void KhiBatCongTac()
13:         {
14:             OnBatCongTac(state);
15:             state = state ? false: true;
16:         }
17:     }
18:     public class BongDen
19:     {
20:         public void TrangThaiDen(bool state)
21:         {
22:             if (state)
23:                 Console.WriteLine("Den Sang");
24:             else
25:                 Console.WriteLine("Den tat");
26:         }
27:     }
28:     public class TV
29:     {

```

```
30:         public void TrangThaiTV(bool state)
31:         {
32:             if (state)
33:                 Console.WriteLine("Mo TV");
34:             else
35:                 Console.WriteLine("Tat tivi");
36:         }
37:     }
38:     class Program
39:     {
40:         static CongTac c = new CongTac();
41:         static BongDen d = new BongDen();
42:         static TV t = new TV();
43:         static void Main(string []args)
44:         {
45:             c.OnBatCongTac += new BatCongTac(d.TrangThaiDen);
46:             c.OnBatCongTac += new BatCongTac(d.TrangThaiDen);
47:             c.KhiBatCongTac();
48:             c.KhiBatCongTac();
49:         }
50:     }
51: }
```

Trong .Net đã định nghĩa sẵn một delegate EventHandler khai báo như sau:

```
public delegate void EventHandler(Object sender, EventArgs e);
```

Trong đó sender là đối tượng phát sinh sự kiện và e chính là tham số của sự kiện. Khi gọi sự kiện 2 tham số này có thể bằng null.

Ví dụ 6.3: Sử dụng EventHandler có dùng tham số thứ 1

```
1: using System;
2: using System.Text;
3: namespace SuDungDelegate
4: {
5:     public class CongTac
6:     {
7:         public event EventHandler BatCongTac;
8:         public bool state;
9:         public void KhiBatCongTac()
10:        {
11:            BatCongTac(this, null);
12:            state = state ? false: true;
13:        }
14:    }
```

```
15:     public class BongDen
16:     {
17:         public void TrangThaiDen(bool state)
18:         {
19:             if (state)
20:                 Console.WriteLine("Den Sang");
21:             else
22:                 Console.WriteLine("Den tat");
23:         }
24:     }
25:     public class TV
26:     {
27:         public void TrangThaiTV(bool state)
28:         {
29:             if (state)
30:                 Console.WriteLine("Mo TV");
31:             else
32:                 Console.WriteLine("Tat tivi");
33:         }
34:     }
35:     class Program
36:     {
37:         static CongTac c = new CongTac();
38:         static BongDen d = new BongDen();
39:         static TV t = new TV();
40:         static void Main(string[] args)
41:         {
42:             c.BatCongTac += new EventHandler(c_BatCongTac);
43:             c.KhiBatCongTac();
44:             c.KhiBatCongTac();
45:         }
46:         static void c_BatCongTac(object sender, EventArgs e)
47:         {
48:             CongTac tam = sender as CongTac;
49:             d.TrangThaiDen(tam.state);
50:             t.TrangThaiTV(tam.state);
51:         }
52:     }
53: }
```

Kết quả thực hiện

Den tat

Tat tivi

Den Sang

Mo TV

Ví dụ 6.4: Sử dụng EventHandler có dùng tham số thứ 2

```
1: using System;
2: using System.Text;
3: namespace SuDungDelegate
4: {
5:     public class MyEventArgs: EventArgs
6:     {
7:         public bool state = false;
8:         public MyEventArgs() { }
9:         public MyEventArgs(bool s)
10:        {
11:            state = s;
12:        }
13:    }
14:    public class CongTac
15:    {
16:        public event EventHandler BatCongTac;
17:        public bool state;
18:        public void KhiBatCongTac()
19:        {
20:            BatCongTac(null,new MyEventArgs(this.state));
21:            state = state ? false: true;
22:        }
23:    }
24:    public class BongDen
25:    {
26:        public void TrangThaiDen(bool state)
27:        {
28:            if (state)
29:                Console.WriteLine("Den Sang");
30:            else
31:                Console.WriteLine("Den tat");
32:        }
33:    }
34:    public class TV
35:    {
36:        public void TrangThaiTV(bool state)
37:        {
38:            if (state)
39:                Console.WriteLine("Mo TV");
40:            else
41:                Console.WriteLine("Tat tivi");
42:        }
43:    }
44:    class Program
```

```
45:     {
46:         static CongTac c = new CongTac();
47:         static BongDen d = new BongDen();
48:         static TV t = new TV();
49:         static void Main(string[] args)
50:         {
51:             c.BatCongTac += new EventHandler(c_BatCongTac);
52:             c.KhiBatCongTac();
53:             c.KhiBatCongTac();
54:         }
55:         static void c_BatCongTac(object sender, EventArgs e)
56:         {
57:             MyEventArgs tam = e as MyEventArgs;
58:             d.TrangThaiDen(tam.state);
59:             t.TrangThaiTV(tam.state);
60:         }
61:     }
62: }
```

Chương trình cho cùng kết quả thực hiện với ví dụ 6.3. Đây là cách cho phép chúng ta có thể mở rộng sử dụng EventHandler tùy biến theo yêu cầu của mình.

6.3 Quản lý lỗi và ngoại lệ

Không gì quan trọng bằng một đoạn mã tốt, chương trình của chúng ta phải luôn có khả năng xử lý những lỗi có thể xảy ra. Ví dụ, trong một quy trình xử lý phức tạp, một đoạn mã nào đó phát hiện nó không được phép đọc một tập tin, hoặc trong khi nó đang gửi yêu cầu đến mạng thì mạng rớt. Trong những tình huống ngoại lệ (exception) như vậy, chương trình không có cách nào khác dù đơn giản là trả về một mã lỗi tương đương. C# có những kỹ thuật tốt để xử lý những loại tình huống này, bằng cơ chế xử lý ngoại lệ (exception handling).

a) Những lớp ngoại lệ của lớp cơ sở

Trong C#, một ngoại lệ là một đối tượng được tạo ra (hoặc được ném) khi một trạng thái lỗi ngoại lệ cụ thể xuất hiện. Những đối tượng này chứa đựng những thông tin giúp ích cho việc truy ngược lại vấn đề. Mặc dù chúng ta có thể tự tạo ra những lớp ngoại lệ riêng, .NET cũng cung cấp cho chúng ta nhiều lớp ngoại lệ được định nghĩa sẵn.

b) Những lớp ngoại lệ cơ bản

Một phần của lớp ngoại lệ nằm trong `IOexception` và những lớp dẫn xuất từ `IOException`. Một phần nằm trong `System.IO` nó liên quan đến việc đọc và viết dữ liệu lên tập tin. Nói chung, không có không gian tên cụ thể cho ngoại lệ; những lớp ngoại lệ nên được đặt trong lớp nơi mà chúng có thể được sinh ra ngoại lệ, vì lý do đó những ngoại lệ có liên quan đến IO thì nằm trong namespace `System.IO`, chúng ta có thể tìm thấy những lớp ngoại lệ nằm trong một vài không gian tên lớp cơ sở.

Những lớp ngoại lệ có đặc điểm chung là đều dẫn xuất từ `System.Exception`. Có 2 lớp quan trọng trong hệ thống các lớp được dẫn xuất từ `System.Exception` là:

- `System.SystemException`: sử dụng cho những ngoại lệ thường xuyên được phát sinh trong thời gian chạy của .NET, hoặc là những lỗi chung thường được sinh ra bởi hầu hết những ứng dụng. Ví dụ như là `StackOverflowException` được phát sinh trong thời gian chạy .NET nếu nó thăm dò thấy vùng nhớ dành cho Stack bị đầy (cấp phát hết). Chúng ta có thể chọn kiểu phát sinh của `ArgumentException`, hoặc là phát sinh từ một lớp con của nó trong chương trình. Những lớp con của `System.SystemException` có thể trình bày những lỗi nghiêm trọng và không nghiêm trọng.
- `System.ApplicationException`: đây là một lớp quan trọng bởi vì nó được dùng cho các lớp ngoại lệ được định nghĩa bởi những hãng thứ ba. Do đó, nếu chúng ta định nghĩa bất kỳ ngoại lệ nào bao phủ trạng thái lỗi của ứng dụng, chúng ta nên dẫn xuất một cách trực tiếp hay gián tiếp từ `System.ApplicationException`.

c) Đón bắt ngoại lệ

Giả sử chúng ta có những đối tượng ngoại lệ có giá trị, vậy làm thế nào chúng ta có thể sử dụng chúng trong đoạn mã để bày những trạng thái lỗi? Để có thể giải quyết điều này trong C# chúng ta thường là phải chia chương trình của chúng ta thành những khối thuộc 3 kiểu khác nhau:

- Khối try chứa đựng đoạn mã mà có dạng là một phần thao tác bình thường trong chương trình của chúng ta, nhưng đoạn mã này có thể gặp phải một vài trạng thái lỗi nghiêm trọng.
- Khối catch chứa đựng đoạn mã mà giải quyết những trạng thái lỗi nghiêm trọng trong đoạn try.
- Khối finally chứa đựng đoạn mã mà dọn dẹp tài nguyên hoặc làm bất kì hành động nào mà chúng ta thường muốn làm xong vào cuối khối try hay catch... điều quan trọng phải hiểu rằng khối finally được thực thi dù có hay không có bất kì ngoại lệ nào được ném ra. Bởi vì mục tiêu chính của khối finally là chứa đựng đoạn mã rõ ràng mà luôn được thực thi, trình biên dịch sẽ báo lỗi nếu chúng ta đặt một lệnh return bên trong khối finally.

Cú pháp C# được sử dụng để thể hiện tất cả điều này cụ thể như sau:

```
try
{
    // mã cho việc thực thi bình thường
}
catch
{
    // xử lí lỗi
}
finally
{
    // dọn dẹp
}
```

Có một vài điều có thể thay đổi trong cú pháp trên:

- Chúng ta có thể bỏ qua khối finally.
- Chúng ta có thể cung cấp nhiều khối catch mà ta muốn xử lí những kiểu lỗi khác nhau.

- Chúng ta có thể bỏ qua khối catch, trong trường hợp cú pháp phục vụ không xác định ngoại lệ, nhưng phải đảm bảo rằng mã trong khối finally sẽ được thực thi khi việc thực thi rời khỏi khối try.

Nhưng điều này đặt ra một câu hỏi: nếu đoạn mã đang chạy trong khối try, làm thế nào nó biết chuyển đến khối catch nếu một lỗi xuất hiện? nếu một lỗi được phát sinh, mã chương trình sẽ ném ra một ngoại lệ. Nói cách khác, nó chỉ ra một lớp đối tượng ngoại lệ và ném nó:

```
throw new OverflowException();
```

Ở đây chúng ta có một thể hiện của đối tượng ngoại lệ của lớp OverflowException. Ngay khi máy tính gặp một câu lệnh throw bên trong khối try, nó ngay lập tức tìm khối catch kết hợp với khối try, nó xác định khối catch đúng bởi việc kiểm tra lớp ngoại lệ nào mà khối catch kết hợp với. Ví dụ, khi đối tượng OverflowException được ném ra việc thực thi sẽ nhảy vào khối catch sau:

```
catch (OverflowException e)
{
}
```

Nói cách khác, máy tính sẽ tìm khối catch mà chỉ định một thể hiện lớp ngoại lệ phù hợp trong cùng một lớp (hoặc của lớp cơ sở).

Giả sử, lúc xem xét đối số, có 2 lỗi nghiêm trọng có thể xảy ra trong nó: lỗi tràn và mảng ngoài biên. Giả sử rằng đoạn mã của chúng ta chứa đựng hai biến kiểu bool. Overflow và OutOfBounds, mà chỉ định liệu trạng thái lỗi này có tồn tại không. Chúng ta đã thấy lớp ngoại lệ đã định nghĩa trước đây tồn tại để chỉ định tràn (OverflowException); tương tự một lớp IndexOutOfRangeException tồn tại để xử lý mảng ngoài biên.

Bây giờ hãy nhìn vào khối try sau:

```
try
{
    // mã cho thực thi bình thường
    if (Overflow == true)
```

```
throw new OverflowException();
// xử lý nhiều hơn
if (OutOfBounds == true)
throw new IndexOutOfRangeException();
// hoặc tiếp tục xử lý bình thường
}
catch (OverflowException e)
{
// xử lý lỗi cho trạng thái lỗi tràn
}
catch (IndexOutOfRangeException e)
{
// xử lý lỗi cho trạng thái lỗi chỉ mục nằm ngoài vùng
}
finally
{
// dọn dẹp
}
```

d) Thực thi nhiều khối catch

Xem ví dụ sau, nó lặp lại việc hỏi người sử dụng nhập vào một số và xuất giá trị của nó. Tuy nhiên vì mục đích của ví dụ, chúng ta sẽ yêu cầu số cần nhập phải trong khoảng từ 0 đến 5 hoặc là chương trình sẽ không thể xử lý số một cách chính xác. Do đó chúng ta sẽ ném ra một ngoại lệ nếu người sử dụng nhập một thứ gì đó ngoài vùng này.

Chương trình sẽ tiếp tục hỏi số cho đến khi người sử dụng nhấn enter mà không gõ bất kì phím gì khác.

```
1: using System;
2: public class ViDu
3: {
4:     public static void Main()
5:     {
6:         string str;
7:         while (true)
8:         {
9:             try
10:            {
11:                Console.Write("Nhập số từ 1 đến 5" +
12:                    "(hay nhấn enter để thoát)> ");
13:                str = Console.ReadLine();
14:                if (str == "")
```

```
15:         break;
16:         int index = Convert.ToInt32(str);
17:         if (index < 0 || index > 5)
18:             throw new IndexOutOfRangeException(
19:                 "Ban nhap " + str);
20:         Console.WriteLine("So vua nhap la " + index);
21:     }
22:     catch (IndexOutOfRangeException e)
23:     {
24:         Console.WriteLine("Exception: " +
25:             "Nen nhap so giua 0 va 5. " + e.Message);
26:     }
27:     catch (Exception e)
28:     {
29:         Console.WriteLine(
30:             "Ngoai le duoc nem ra la: " + e.Message);
31:     }
32:     catch
33:     {
34:         Console.WriteLine("Mot so biet le khac xuat hien");
35:     }
36:     finally
37:     {
38:         Console.WriteLine("Cam on");
39:     }
40: }
41: }
42: }
```

Chapter 7: Quản lý bộ nhớ và con trỏ

Mục đích của chương:

- Tìm hiểu về heap và stack và cách thức lưu trữ các biến tham trị và tham chiếu
- Khai báo các khối mã “không an toàn” để truy xuất bộ nhớ trực tiếp.
- Tổ chức và hoạt động của cơ chế thu gom rác trong .Net.

7.1 Quản lý bộ nhớ

Một trong những ưu điểm của C# là chúng ta không cần quan tâm về việc quản lý bộ nhớ bên dưới vì điều này đã được bộ gom rác (garbage collector) của C# làm rồi. Mặc dù vậy nếu ta muốn viết các đoạn mã tốt, có hiệu suất cao, ta cần tìm hiểu về cách quản lý bộ nhớ bên dưới.

a) Giá trị các kiểu dữ liệu

Windows dùng hệ thống địa chỉ ảo (virtual addressing) để ánh xạ từ địa chỉ bộ nhớ đến vị trí thực sự trong bộ nhớ vật lý hoặc trên đĩa được quản lý phía sau Windows. Kết quả là mỗi ứng dụng trên nền xử lý 32-bit thấy được 4GB bộ nhớ, không cần biết bộ nhớ vật lý thực sự có kích thước bao nhiêu (nền xử lý 64 bit thì bộ nhớ này lớn hơn). 4GB bộ nhớ này được gọi là không gian địa chỉ ảo (virtual address space) hay bộ nhớ ảo (virtual memory). Để đơn giản ta gọi nó là bộ nhớ, mỗi vùng nhớ từ 4GB này được đánh số từ 0. Nếu ta muốn chỉ định một giá trị lưu trữ trên 1 phần cụ thể trong bộ nhớ, ta cần cung cấp số đại diện cho vùng nhớ này. Trong ngôn ngữ cấp cao như là C#, VB, C++, Java... một trong những cách mà trình biên dịch làm là chuyển đổi tên đọc được (ví dụ tên biến) thành địa chỉ vùng nhớ mà bộ xử lý hiểu. 4GB bộ nhớ này thực sự chứa tất cả các phần của chương trình bao gồm mã thực thi và nội dung của biến được dùng khi chương trình chạy. Bất kì thư viện liên kết động (DLL-Dynamic Link Library) được gọi sẽ nằm trong cùng không gian địa chỉ này, mỗi mục của mã hoặc dữ liệu sẽ có vùng định nghĩa riêng.

Stack là một vùng nhớ được cấp phát cho chương trình để lưu trữ các biến kiểu giá trị. Khi ta gọi phương thức, stack cũng được dùng để sao chép các tham số được truyền.

b) Các kiểu dữ liệu tham chiếu

Chúng ta có thể dùng một số phương thức để cấp phát vùng nhớ để lưu trữ dữ liệu, và giữ cho dữ liệu còn nguyên giá trị ngay cả khi phương thức kết thúc. Điều này có thể làm với kiểu tham chiếu.

Xét đoạn mã sau:

```
void ThucHien()  
{  
    KháchHang kh;  
    kh = new KháchHang();  
    KháchHang m = new KháchHang60();  
}
```

Trong đoạn mã này ta giả sử gồm 2 lớp KháchHang và KháchHang60. Ta khai báo một tham chiếu gọi là kh, được cấp phát trong Stack nhưng nên nhớ rằng đây là một tham chiếu, không phải là một thể hiện KháchHang. Không gian mà tham chiếu kh chiếm là 4 byte. Ta cần 4 byte để có thể lưu một số nguyên giá trị từ 0 đến 4GB. Sau đó ta có dòng:

```
kh = new KháchHang();
```

Dòng này đầu tiên cấp phát vùng nhớ trong heap để lưu trữ một thể hiện của KháchHang. Sau đó nó đặt biến kh để lưu địa chỉ của vùng nhớ được cấp phát, đồng thời nó cũng gọi phương thức tạo lập KháchHang() để khởi tạo một thể hiện lớp.

Thể hiện của KháchHang không đặt trong stack mà sẽ đặt trong heap. Ta không biết chính xác một thể hiện KháchHang chiếm bao nhiêu byte, ta xem nó là 32 byte. 32 byte này chứa các trường thể hiện của KháchHang cùng vài thông tin mà .NET dùng để xác định danh tính và quản lý các thể hiện lớp của nó. .NET tìm trong heap khối 32 byte còn trống, giả sử nằm ở địa chỉ 200000, và tham chiếu kh nằm ở vị trí 799996-799999 trên stack. Không như stack, bộ nhớ trong heap được cấp phát theo chiều từ dưới lên, vì thế không gian trống được tìm thấy phía trên không gian đã dùng.

Dòng kế tiếp thực hiện tương tự, ngoại trừ không gian trên stack cho tham chiếu m cần được cấp phát vào cùng lúc cấp phát cho m trên heap:

```
KhachHang m = new KhachHang60();
```

4 byte được cấp phát trên stack cho tham chiếu m lưu ở địa chỉ 799992-799995. Trong khi thể hiện m sẽ được cấp phát từ vị trí 200032 đi lên trên heap.

.NET runtime sẽ cần duy trì thông tin về trạng thái của heap, thông tin này cũng cần được cập nhật khi dữ liệu mới được thêm vào heap. Để minh họa điều này, ta hãy xem điều gì xảy ra khi ta thoát phương thức và tham chiếu kh và m nằm ngoài phạm vi.

Theo cách làm việc bình thường thì con trỏ stack sẽ được tăng để những biến này không còn tồn tại nữa. Tuy nhiên các biến này chỉ lưu địa chỉ, không phải thể hiện lớp, dữ liệu của nó vẫn nằm trong heap. Ta có thể thiết lập các biến tham chiếu khác nhau để trỏ đến cùng một đối tượng; nghĩa là những đối tượng đó sẽ có giá trị sau khi tham chiếu kh và m nằm ngoài phạm vi và sự khác biệt quan trọng giữa stack và heap: đối tượng được cấp phát liên tiếp trên heap, thời gian sống không lồng nhau.

Khi ta giải thích cách hoạt động trên heap, ta nhấn mạnh rằng chỉ stack mới có khả năng lồng thời gian sống của các biến. Vậy khi thời gian sống của các tham chiếu nằm ngoài phạm vi thì heap làm việc như thế nào trên các biến này? Câu trả lời là bộ thu gom rác sẽ làm điều này. Khi bộ gom rác chạy, nó sẽ gỡ bỏ tất cả những đối tượng từ heap mà không còn tham chiếu nữa. Ngay sau khi nó làm điều này, heap sẽ có các đối tượng rải rác trên nó, nằm lẫn với các khoảng trống.

Bộ gom rác không để heap trong tình trạng này, ngay khi nó giải phóng tất cả các đối tượng có thể, nó sẽ di chuyển tất cả chúng trở về cuối của heap để thành một khối liên tục lại. Tất nhiên khi những đối tượng này được di chuyển tất cả các tham chiếu của nó đều được cập nhật lại.

7.2 Giải phóng tài nguyên

a) Chu kì sống và thời gian của một đối tượng

Điều gì xảy ra khi chúng ta tạo và hủy một đối tượng.

Chúng ta tạo đối tượng như sau:

```
SinhVien a = new SinhVien(); // a là kiểu tham chiếu
```

Tiến trình tạo một đối tượng gồm hai giai đoạn. Đầu tiên, hoạt động new cấp phát bộ nhớ thô từ heap. Chúng ta không thể điều khiển giai đoạn này khi đối tượng được tạo. Thứ hai, hoạt động new chuyển bộ nhớ thô vào trong một đối tượng; nó phải khởi tạo đối tượng. Chúng ta có thể điều khiển giai đoạn này dùng phương thức tạo lập.

Khi chúng ta đã tạo đối tượng, chúng ta có thể truy xuất thành viên của nó dùng toán tử “.”. Ví dụ:

```
a.Ten = "Nguyen Van A";
```

Chúng ta có thể khai báo biến tham chiếu đến cùng đối tượng:

```
SinhVien ref = a;
```

Chúng ta có thể tạo bao nhiêu tham chiếu đến đối tượng cũng được. CLR sẽ theo dõi tất cả những tham chiếu này. Nếu biến a biến mất (ngoài phạm vi), những biến khác (như là ref) vẫn tồn tại. Vì vậy thời gian sống của một đối tượng không bị ràng buộc vào một biến tham chiếu cụ thể. Một đối tượng chỉ có thể bị hủy khi tất cả các tham chiếu đến nó biến mất.

Tương tự như tạo, hủy đối tượng cũng gồm 2 giai đoạn. Đầu tiên, chúng ta phải thực hiện một số dọn dẹp được viết trong phương thức hủy. Thứ hai, bộ nhớ thô được trả lại cho heap, chúng ta không thể kiểm soát giai đoạn này. Quá trình hủy một đối tượng và trả bộ nhớ lại heap được biết là cơ chế thu dọn (garbage collection).

b) Viết phương thức hủy

Chúng ta có thể sử dụng một phương thức hủy để thực hiện việc dọn dẹp khi đối tượng cần được gom rác. Chúng ta viết dấu “~” theo sau bởi tên lớp. Trong ví dụ sau đếm số thể hiện của lớp bằng cách tăng biến tĩnh count trong phương thức tạo lập và giảm biến này trong phương thức hủy:

```
class ViDu
{
    public ViDu()
```

```
{
    this.soTheHien++;
}
~ViDu()
{
    this.soTheHien--;
}
public static int SoTheHien()
{
    return this.soTheHien;
}
...
private static int soTheHien = 0;
}
```

Một số hạn chế khi sử dụng phương thức hủy:

- Chúng ta không thể khai báo một phương thức hủy trong cấu trúc vì cấu trúc là một kiểu giá trị lưu trên stack không phải trên heap nên không cần sử dụng cơ chế thu dọn:

```
struct ViDu
{
    ~ViDu() {... } // lỗi biên dịch
}
```

- Chúng ta không thể khai báo chỉ định truy xuất (như là public) cho phương thức hủy bởi vì chúng ta không thể gọi phương thức hủy, nó được gọi bởi cơ chế thu dọn.
public ~ViDu() {... } // lỗi biên dịch
- Chúng ta không được khai báo phương thức hủy với tham số.

```
~ViDu(int parameter) {... } //lỗi biên dịch
```

- Trình biên dịch tự động dịch phương thức hủy thành phương thức Object.Finalize.

```
class ViDu
{
    ~ViDu() {... }
}
```

Dịch thành:

```
class ViDu
```



```
{  
    protected override void Finalize()  
    {  
        try { ... }  
        finally { base.Finalize(); }  
    }  
}
```

c) Tại sao sử dụng cơ chế thu dọn

Trong C#, chúng ta không thể chủ động hủy đối tượng, lý do tại sao các nhà thiết kế C# cấm chúng ta thực hiện điều này:

- Nếu chúng ta quên xóa đối tượng, điều này có nghĩa là phương thức hủy không được chạy và bộ nhớ sẽ không được thu hồi lại cho heap và chúng ta có thể nhanh chóng cạn kiệt bộ nhớ.
- Khi chúng ta đang cố xóa một đối tượng đang hoạt động. Nhớ rằng đối tượng là kiểu tham chiếu. Nếu một lớp giữ một tham chiếu đến một đối tượng đã bị hủy. Nó có thể tham chiếu đối tượng không sử dụng hay tham chiếu đến một đối tượng hoàn toàn khác trong cùng vùng nhớ.
- Chúng ta muốn xóa cùng đối tượng nhiều lần. Điều này có thể rất tai hại dựa trên mã trong phương thức hủy.

Cơ chế thu dọn có nhiệm vụ hủy đối tượng cho chúng ta và nó đảm bảo các vấn đề sau:

- Mỗi đối tượng sẽ bị hủy và khi chương trình kết thúc tất cả các đối tượng đang tồn tại sẽ bị hủy.
- Mỗi đối tượng chính xác bị hủy một lần.
- Mỗi đối tượng bị hủy khi không còn tham chiếu đến nó.

Những đảm bảo này rất hữu ích và tiện lợi cho các lập trình viên, chúng ta chỉ tập trung vào phần logic của chương trình.

Một đặc trưng quan trọng nữa của cơ chế thu dọn là phương thức hủy sẽ không chạy cho tới khi đối tượng là rác cần được thu dọn. Nếu chúng ta viết một phương thức tạo lập, chúng ta sẽ biết nó sẽ chạy nhưng không biết lúc nào chạy?

d) Cách thức làm việc của cơ chế thu dọn

Cơ chế thu dọn chạy trên một tiến trình riêng của nó và chỉ chạy ở một thời gian nào đó (thông thường khi ứng dụng chạy đến cuối phương thức). Khi nó chạy, các tiến trình khác đang chạy trong ứng dụng của chúng ta sẽ tạm thời treo, bởi vì cơ chế thu dọn cần di chuyển các đối tượng và cập nhật các tham chiếu đến các đối tượng. Cơ chế thu dọn thực hiện các bước sau:

1. Xây dựng một bản đồ của tất cả các đối tượng có thể đến được. Cơ chế thu dọn xây dựng bản đồ này rất cẩn thận vì tránh tham chiếu vòng gây ra lập vô hạn. Bất kì đối tượng nào không có trong bản đồ này được xem là không đến được.
2. Kiểm tra các đối tượng không thể đến có phương thức hủy cần để chạy hay không. Nếu có nó đưa vào trong một hàng đợi đặc biệt gọi là F-reachable.
3. Thu hồi các đối tượng không thể đến còn lại, bằng cách di chuyển các đối tượng xuống dưới heap. Vì vậy sự phân mảnh và giải phóng bộ nhớ ở đầu heap. Khi cơ chế thu dọn di chuyển một đối tượng, nó cũng cập nhật tất cả tham chiếu đến đối tượng này. Ở thời điểm này, nó cho phép các tiến trình khác chạy lại.
4. Thu hồi các đối tượng trong F-reachable trong một tiến trình độc lập.

7.3 Mã không an toàn

Có những trường hợp ta cần truy xuất bộ nhớ trực tiếp như khi ta muốn truy xuất vào các hàm bên ngoài (không thuộc .NET) hay tham số yêu cầu truyền vào là con trỏ, hoặc là vì ta muốn truy nhập vào nội dung bộ nhớ để sửa lỗi. Trong phần này ta sẽ xem xét cách C# đáp ứng những điều này như thế nào.

a) Con trỏ

Con trỏ đơn giản là một biến lưu địa chỉ như là một tham chiếu. Sự khác biệt là cú pháp C# trong tham chiếu không cho phép ta truy xuất vào địa chỉ bộ nhớ.

b) Ưu điểm của con trỏ:

- Cải thiện sự thực thi: cho ta biết những gì ta đang làm, đảm bảo rằng dữ liệu được truy xuất hay thao tác theo cách hiệu quả nhất - đó là lí do mà C và C++ cho phép dùng con trỏ trong ngôn ngữ của mình.
- Khả năng tương thích ngược: đôi khi ta phải sử dụng lại các hàm API cho mục đích của ta. Mà các hàm API được viết bằng C, ngôn ngữ dùng con trỏ rất nhiều, nghĩa là nhiều hàm lấy con trỏ như tham số. Hoặc là các DLL do một hãng nào đó cung cấp chứa các hàm lấy con trỏ làm tham số. Trong nhiều trường hợp ta có thể viết các khai báo `DllImport` theo cách tránh sử dụng con trỏ, ví dụ như dùng lớp `System.IntPtr`.
- Ta có thể cần tạo ra các địa chỉ vùng nhớ có giá trị cho người dùng - ví dụ nếu ta muốn phát triển một ứng dụng mà cho phép người dùng tương tác trực tiếp đến bộ nhớ, như là một debugger.

c) Nhược điểm:

- Cú pháp để lấy các hàm phức tạp hơn.
- Con trỏ khó sử dụng.
- Nếu không cẩn thận ta có thể viết lên các biến khác, làm tràn stack, mất thông tin, đùng độ...
- C# có thể từ chối thực thi những đoạn mã không an toàn này (đoạn mã có sử dụng con trỏ)

Ta có thể đánh dấu đoạn mã có sử dụng con trỏ bằng cách dùng từ khoá **unsafe**.

- Dùng cho hàm

```
unsafe int PhuongThuc()  
{  
    // mã có thể sử dụng con trỏ  
}  
- Dùng cho lớp hay struct  
unsafe class ViDu  
{  
    // bất kì phương thức nào trong lớp cũng có thể dùng con trỏ  
}  
- Dùng cho một trường  
class ViDu  
{  
    unsafe int *pX; //khai báo một trường con trỏ trong lớp  
}
```

Hoặc một khối mã

```
void PhuongThuc()  
{  
    // mã không sử dụng con trỏ  
    unsafe  
    {  
        // Mã sử dụng con trỏ  
    }  
    // Mã không sử dụng con trỏ  
}
```

Tuy nhiên ta không thể đánh dấu một biến cục bộ là unsafe

```
int PhuongThuc()  
{  
    unsafe int *pX; // Sai  
}
```

Để biên dịch các mã chứa khối unsafe ta dùng lệnh sau:

```
csc /unsafe Nguồn.cs
```

hay

```
csc -unsafe Nguồn.cs
```

d) Cú pháp con trỏ

```
int * pWidth, pHeight;  
double *pResult;
```

Lưu ý khác với C++, kí tự * kết hợp với kiểu hơn là kết hợp với biến - nghĩa là khi ta khai báo như ở trên thì pWidth và pHeight đều là con trỏ do có * sau kiểu int, khác với C++ ta phải khai báo * cho cả hai biến trên thì cả hai mới là con trỏ.

Cách dùng * và & giống như trong C++:

- &: lấy địa chỉ
- *: lấy nội dung của địa chỉ

e) Ép kiểu con trỏ thành kiểu int

Vì con trỏ là một số int lưu địa chỉ nên ta có thể chuyển tường minh con trỏ thành kiểu int hay ngược lại. Ví dụ:

```
int x = 10;
int *pX, pY;
pX = &x;
pY = pX;
*pY = 20;
uint y = (uint)pX;
int *pD = (int*)y;
```

Một lý do để ta phải ép kiểu là Console.WriteLine không có nạp chồng hàm nào nhận thông số là con trỏ do đó ta phải ép nó sang kiểu số nguyên int

```
Console.WriteLine("Địa chỉ là " + pX); // sai Lỗi biên dịch
Console.WriteLine("Địa chỉ là " + (uint) pX); // Đúng
```

f) Ép kiểu giữa những kiểu con trỏ

Ta cũng có thể chuyển đổi tường minh giữa các con trỏ trỏ đến một kiểu khác ví dụ:

```
byte aByte = 8;
byte *pByte = &aByte;
double *pDouble = (double*)pByte;
```

g) Con trỏ void

Nếu ta muốn giữ một con trỏ, nhưng không muốn đặc tả kiểu cho con trỏ ta có thể khai báo con trỏ là void:

```
void *pointerToVoid;
```

```
pointerToVoid = (void*)pointerToInt;
```

mục đích là khi ta cần gọi các hàm API mà đòi hỏi thông số void*.

h) Toán tử sizeof

Lấy thông số là tên của kiểu và trả về số byte của kiểu đó ví dụ:

```
int x = sizeof(double);
```

x có giá trị là 8

Bảng kích thước kiểu:

sizeof(sbyte) = 1;	sizeof(byte) = 1;
sizeof(short) = 2;	sizeof(ushort) = 2;
sizeof(int) = 4;	sizeof(uint) = 4;
sizeof(long) = 8;	sizeof(ulong) = 8;
sizeof(char) = 2;	sizeof(float) = 4;
sizeof(double) = 8;	sizeof(bool) = 1;

Ta cũng có thể dùng sizeof cho struct nhưng không dùng được cho lớp.

Chương 8: Chuỗi, biểu thức quy tắc và tập hợp

Mục đích của chương:

- Sử dụng bộ thư viện thao tác trên chuỗi.
- Sử dụng biểu thức quy tắc trong việc kiểm tra hợp lệ dữ liệu.
- Làm việc với các cấu trúc dữ liệu động như ArrayList, Từ điển, Generic
- Sử dụng siêu dữ liệu của .Net.
- Vai trò của thuộc tính trong quá trình xây dựng ứng dụng.
- Sử dụng thuộc tính trong quá trình phát triển.
- Sử dụng reflection để lấy tất cả thông tin về lớp.
- Thao tác trên tập tin văn bản dùng luồng.

8.1 System.String

Trước khi kiểm tra các lớp chuỗi khác, ta sẽ xem lại nhanh những phương thức trong lớp chuỗi. System.String là lớp được thiết kế để lưu trữ chuỗi, bao gồm một số lớn các thao tác trên chuỗi. Không chỉ thế mà còn bởi vì tầm quan trọng của kiểu dữ liệu này, C# có từ khoá riêng cho nó và kết hợp với cú pháp để tạo nên cách dễ dàng trong thao tác chuỗi.

Ta có thể nối chuỗi:

```
string message1 = "Hello";  
message1 += ", There";  
string message2 = message1 + "!";  
Trích 1 phần chuỗi dùng chỉ mục:  
char char4 = message[4]; // trả về 'a'. lưu ý rằng kí tự bắt đầu  
tính từ chỉ mục 0  
các phương thức khác ( sơ lược):
```

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 1 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
------	-------------------------------------

Xây dựng chuỗi

Chuỗi là một lớp mạnh với nhiều phương thức hữu ích. Tuy nhiên, nó thực sự là kiểu dữ liệu cố định, nghĩa là mỗi lần ta khởi động một đối tượng chuỗi, thì đối tượng chuỗi đó không bao giờ được thay đổi. Những phương thức hoặc toán tử mà cập nhật nội dung của chuỗi thực sự là tạo ra một chuỗi mới, sao chép chuỗi cũ vào nếu cần thiết.

Ví dụ:

```
string str = "Lap trinh huong doi tuong.";
str += "Ngon ngu C#";
```

Đầu tiên lớp System.String được tạo và khởi tạo giá trị "Lap trinh huong doi tuong." . Trong thời gian thực thi .NET sẽ định vị đủ bộ nhớ trong chuỗi để chứa đoạn kí tự này và tạo ra một biến str để chuyển đến một thể hiện chuỗi.

Ở dòng tiếp theo, khi ta thêm kí tự vào, ta sẽ tạo ra một chuỗi mới với kích thước đủ để lưu trữ cả hai chuỗi; chuỗi gốc " Lap trinh huong doi tuong.", sẽ được sao chép vào chuỗi mới với chuỗi bổ sung " Ngon ngu C#". Sau đó địa chỉ trong biến str được cập nhật, vì vậy biến sẽ trỏ đúng đến đối tượng chuỗi mới. Chuỗi cũ không còn được tham chiếu, không có biến nào truy cập vào nó, và vì vậy nó sẽ được bộ thu gom rác gỡ bỏ.

Ví dụ 8.1: Sử dụng thư viện chuỗi

```
1: using System;
2: using System.Text;
3: namespaceNhapXuat
4: {
5:     class Program
6:     {
7:         static void Main(string[] args)
8:         {
9:             string s = "    Lap trinh huong doi tuong    ";
10:            Console.WriteLine(s);
11:            Console.WriteLine(s.ToUpper());
12:            Console.WriteLine(s.ToLower());
13:            Console.WriteLine(s.Length);
```

```

14:         Console.WriteLine(s.IndexOf("trinh")); //Tim vi
           tri dau tien
15:         Console.WriteLine(s.LastIndexOf("tuong")); //Tim
           vi tri cuoi cung
16:         Console.WriteLine(s.Trim()); //Xoa khoang trang
           dau va cuoi
17:         Console.WriteLine(s.Replace("huong doi tuong",
           "OOP")); //thay the chuoi con
18:         string b = s.ToUpper().Trim().Replace("huong doi
           tuong", "OOP");
19:         string c = s.Trim();
20:         string[] tu = c.Split(' ');
21:         Console.WriteLine("So tu trong chuoi la {0}",
           tu.Length);
22:         for (int i = 0; i < tu.Length; i++)
23:             Console.WriteLine(tu[i]);
24:         Console.WriteLine("Nhap vao ho ten ");
25:         string s = Console.ReadLine().Trim();
26:         int dau = s.IndexOf(' ');
27:         int cuoi = s.LastIndexOf(' ');
28:         Console.WriteLine("Ho la " + s.Substring(0,
           dau));
29:         Console.WriteLine("Ten la " + s.Substring(cuoi,
           s.Length - cuoi));
30:         Console.ReadKey();
31:     }
32: }
33: }

```

8.2 Biểu thức quy tắc

Ngôn ngữ biểu thức chính quy là ngôn ngữ được thiết kế đặc biệt cho việc xử lý 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 tùy 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. 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 lý, 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 phân kí tự mà tìm thấy bởi thao tác tìm kiếm.

Bảng sau thể hiện một số kí tự hoặc chuỗi escape mà ta có thể dùng

	Ý nghĩa	Ví dụ	Các mẫu sẽ so khớp
<code>^</code>	Bắt đầu của chuỗi nhập	<code>^B</code>	B, nhưng chỉ nếu kí tự đầu tiên trong chuỗi
<code>\$</code>	Kết thúc của chuỗi nhập	<code>X\$</code>	X, nhưng chỉ nếu kí tự cuối cùng trong chuỗi
<code>.</code>	Bất kì kí tự nào ngoại trừ kí tự xuống dòng(<code>\n</code>)	<code>i. ation</code>	isation, ization
<code>*</code>	Kí tự trước có thể được lặp lại 0 hoặc nhiều lần	<code>ra*t</code>	rt, rat, raat, raaat, ...
<code>+</code>	Kí tự trước có thể được lặp lại một hoặc nhiều lần	<code>ra+t</code>	rat, raat, raaat ..., (nhưng không rt)
<code>?</code>	Kí tự trước có thể được lặp lại 0 hoặc một lần	<code>ra?t</code>	Chỉ rt và rat

\s	Bất kì kí tự khoảng trắng	\sa	[space]a, \ta, \na (\t và \n có ý nghĩa giống như trong C#)
\S	Bất kì kí tự nào không phải là khoảng trắng	\SF	aF, rF, cF, nhưng không \tf
\b	Từ biên	ion\b	Bất kì từ kết thúc với ion
\B	bất kì vị trí nào không phải là từ biên	\BX\B	bất kì kí tự X ở giữa của một từ

Ví dụ chương trình sau dùng biểu thức quy tắc kiểm tra một chuỗi có phải là địa chỉ email hay không?

```

1: using System;
2: using System.Text.RegularExpressions;
3: public class MainClass{
4:
5:     public static void Main(){
6:         Regex r = new Regex(@"\w+@(\w+\.)+\w+");
7:         Console.WriteLine(r.IsMatch("j@by.com"));
8:         Console.WriteLine(r.IsMatch("jn@b.com"));
9:         Console.WriteLine(r.IsMatch("o47@l.com"));
10:        Console.WriteLine(r.IsMatch("oop.com"));
11:    }
12: }
```

8.3 Tập hợp đối tượng

a) Array List

ArrayList giống như mảng, ngoại trừ nó có khả năng mở rộng, được đại diện bởi lớp System.Collection.ArrayList.

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 2 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 SinhVien:

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

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

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

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

```
a.Add(new SinhVien());  
a.Add(new SinhVien());
```

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

```
SinhVien x = (SinhVien)a[1];
```

Ví dụ này cũng chỉ ra ArrayList định nghĩa một chỉ mục, 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 ArrayList:

```
a.Insert(1, new SinhVien()); // chèn vào vị trí 1
```

Đây là một phương thức nạp chồng, vì vậy rất có ích khi ta muốn chèn tất cả các phần tử trong một tập hợp vào ArrayList. Ta có thể bỏ một phần tử:

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

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:

```
a.Capacity = 30;
```

Tuy nhiên việc thay đổi dung lượng 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 num = a.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. Ví dụ nếu mảng được xem là một tham số của phương thức, ta chỉ phải sao chép tham chiếu chứ không phải đối tượng:

```
SinhVien [] ds = new SinhVien[a.Count];  
for (int i=0 ; i< a.Count ; i++)  
    ds[i] = (SinhVien)a[i];
```

b) Tập hợp (Collection)

Ý tưởng của Collection là nó trình bày một tập các đối tượng mà ta có thể truy xuất bằng việc lặp qua từng phần tử. Cụ thể là một tập đối tượng mà ta có thể truy nhập sử dụng vòng lặp foreach. Ví dụ:

```
foreach (SinhVien v in a)  
{  
    //Thực hiện thao tác nào đó cho SinhVien v  
}
```

Ta xem biến a là một tập hợp, khả năng để dùng vòng lặp foreach là mục đích chính của collection.

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 enumerator, mà có thể duyệt qua từng mục trong collection. Đặc biệt hơn, một collection phải thực thi một interface System.Collections.IEnumerable. IEnumerable định nghĩa chỉ một phương thức như sau:

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

Mục đích của GetEnumerator() là để trả về đối tượng enumerator. Khi ta tập hợp những đoạn mã trên đối tượng enumerator được mong đợi để thực thi một interface, System.Collections. IEnumerator.

Ngoài ra còn có một interface khác, ICollection, được dẫn xuất từ IEnumerable.

Những collection phức tạp hơn sẽ thực thi interface này. Bên cạnh GetEnumerator(), nó thực thi một thuộc tính trả về trực tiếp số phần tử trong collection. Nó cũng hỗ trợ việc sao chép một collection đến một mảng và cung cấp thông tin đặc tả nếu đó là một luồng an toàn.

IEnumerator có cấu trúc sau:

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

IEnumerator 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 kỳ một phần tử nào trong collection, và ta phải gọi MoveNext() 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 lặp lại cho đến khi hết các 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().

Một 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 sử dụng trong collection làm việc tốt hơn trên mảng.

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

```
IEnumerator enumerator = a.GetEnumerator();
SinhVien v;
enumerator.MoveNext();
while ( (v = (SinhVien)enumerator.Current) != null)
{
    //Thao tác sinh viên v
    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.

Ví dụ sử dụng IEnumerable duyệt qua các phần tử trong mảng theo một quy tắc nào đó:

```
1: using System;
2: using System.Collections;
3: using System.ComponentModel;
4: class ViduLapIterator: IEnumerator
5: {
6:     public ViduLap cha;
7:     public int vt;
8:
9:     internal ViduLapIterator(ViduLap cha)
10:    {
11:        this.cha = cha;
12:        vt = -1;
13:    }
14:    public bool MoveNext()
15:    {
16:        if (vt != cha.a.Length)
17:        {
18:            vt++;
19:        }
20:        return vt < cha.a.Length;
```



```
21:     }
22:     public object Current
23:     {
24:         get
25:         {
26:             if (vt == -1 || vt == cha.a.Length)
27:             {
28:                 throw new InvalidOperationException();
29:             }
30:             int index = (vt + cha.vt);
31:             index = index % cha.a.Length;
32:             return cha.a[index];
33:         }
34:     }
35:     public void Reset()
36:     {
37:         vt = -1;
38:     }
39: }
40: class ViduLap: IEnumerable
41: {
42:     public object[] a;
43:     public int vt;
44:
45:     public ViduLap(object[] a, int vt)
46:     {
47:         this.a = a;
48:         this.vt = vt;
49:     }
50:
51:     public IEnumerator GetEnumerator()
52:     {
53:         return new ViduLapIterator(this);
54:     }
55:     public static void Main()
56:     {
57:         object[] a = { "a", "b", "c", "d", "e" };
58:         ViduLap collection = new ViduLap(a, 3);
59:         foreach (object x in collection)
60:         {
61:             Console.WriteLine(x);
62:         }
63:     }
64: }
```

Kết quả chương trình:

d

e
a
b
c

c) Generic

Chúng ta có thể sử dụng kiểu object là kiểu tham chiếu đến bất kỳ kiểu giá trị hay biến. Tất cả các kiểu tham chiếu tự động kế thừa (trực tiếp hay gián tiếp) từ lớp System.Object trong .Net. Chúng ta có thể sử dụng những thông tin này để tạo các lớp hay phương thức tổng quát hóa cao. Ví dụ, nhiều lớp trong namespace System.Collections cho phép chúng ta tạo tập hợp với kiểu dữ liệu bất kỳ. Lớp hàng đợi System.Collections.Queue cho phép chúng ta tạo hàng đợi chứa kiểu bất kỳ. Ví dụ sau tạo và thao tác một hàng đợi của các đối tượng SinhVien:

```
using System.Collections;
...
Queue myQueue = new Queue();
SinhVien a = new SinhVien();
myQueue.Enqueue(a);
...
a = (SinhVien)myQueue.Dequeue();
```

Phương thức Enqueue thêm một đối tượng vào đầu hàng đợi và phương thức Dequeue xóa đối tượng ở cuối hàng đợi. Các phương thức này định nghĩa như sau:

```
public void Enqueue( object item );
public object Dequeue();
```

Bởi vì phương thức Enqueue và Dequeue thao tác trên object nên chúng ta có thể làm việc trên hàng đợi với bất cứ kiểu dữ liệu nào SinhVien, PhanSo, HinhHoc hay bất kỳ lớp nào mà chúng ta nhìn thấy trước đây. Tuy nhiên, chúng ta phải chuyển kiểu giá trị trả về từ phương thức Dequeue đến kiểu giá trị thích hợp vì trình biên dịch sẽ không thực hiện chuyển kiểu tự động các đối tượng này. Nếu chúng ta không ép kiểu giá trị trả về, trình biên dịch sẽ tạo lỗi “Cannot implicitly convert type 'object' to 'Circle’”:

```
SinhVien a = new SinhVien();
myQueue.Enqueue(a);
```

```
...  
a = (SinhVien)myQueue.Dequeue(); // ép kiểu là bắt buộc
```

Chúng ta phải thực hiện ép kiểu tương minh các kiểu object. Và rất dễ dàng viết mã như sau:

```
Queue myQueue = new Queue();  
SinhVien a = new SinhVien();  
myQueue.Enqueue(a);  
...  
PhanSo p = (PhanSo)myQueue.Dequeue();
```

Mặc dù mã này sẽ biên dịch nhưng nó không hợp lệ và sẽ tạo ra một ngoại lệ `System.InvalidCastException` ở thời gian thực thi. Lỗi này là vì chúng ta đang lưu trữ một tham chiếu đến một `SinhVien` trong biến `p` và hai kiểu này là không tương thích. Lỗi này chỉ xuất hiện trong thời gian chạy vì trình biên dịch không có đủ thông tin. Nó chỉ có thể xác định kiểu thật sự của đối tượng được lấy ra khỏi hàng đợi trong thời gian chạy.

Một điều không thuận lợi của tiếp cận sử dụng object để tạo lớp hay phương thức tổng quát hóa cao là nó có thể sử dụng thêm bộ nhớ và thời gian xử lý ở thời gian thực thi cần chuyển kiểu object thành kiểu giá trị và ngược lại. Xem xét mã sau thao tác trên hàng đợi các số nguyên:

```
Queue myQueue = new Queue();  
int myInt = 99;  
myQueue.Enqueue(myInt); // đóng hộp int trong một object  
...  
myInt = (int)myQueue.Dequeue(); // mở hộp object đến int.
```

Kiểu dữ liệu hàng đợi mong muốn các mục lưu trữ của nó là các kiểu tham chiếu. Nếu hàng đợi lưu trữ các kiểu giá trị như là `int` nó yêu cầu thực hiện đóng hộp và mở hộp và trong một kiểu tham chiếu và nó có thể ảnh hưởng đến hiệu suất chương trình khi làm việc với một hàng đợi có một số lượng lớn các số kiểu giá trị.

Giải pháp Generic

Generic được thêm vào C# 2.0 nhằm loại bỏ nhu cầu cho việc ép kiểu, nâng cao an toàn của kiểu, giảm số lượng mở hộp yêu cầu và giúp dễ dàng hơn trong việc tạo các phương thức và lớp có tính tổng quát cao. Lớp và phương thức generic chấp nhận kiểu tham số mà chỉ ra kiểu của đối tượng mà nó hoạt động trên. Net 2.0 cung cấp các phiên bản generic của nhiều lớp tập hợp và giao diện trong namespace `System.Collections.Generic`. Đoạn mã sau chỉ ra các sử dụng lớp hàng đợi (Queue) generic trong namespace `System.Collections.Generic` để tạo hàng đợi các đối tượng `SinhVien`:

```
using System.Collections.Generic;
...
Queue< SinhVien > myQueue = new Queue< SinhVien >();
SinhVien a = new SinhVien ();
myQueue.Enqueue(a);
...
a = myQueue.Dequeue();
```

Có hai điều cần chú ý trong mã trên:

- Sử dụng kiểu tham số giữa dấu “<>” khi khai báo biến `myQueue`.
- Không cần chuyển kiểu khi thực thi phương thức `Dequeue`.

Tham số kiểu chỉ ra kiểu đối tượng chấp nhận bởi hàng đợi. Tất cả tham chiếu đến phương thức trong hàng đợi này sẽ sử dụng kiểu này hơn là object.

Nếu chúng ta kiểm tra mô tả của lớp `Queue` generic trong Visual Studio 2005 Documentation. Chúng ta sẽ thấy nó được định nghĩa là:

```
public class Queue<T>: ...
```

T chỉ ra kiểu tham số, nó hoạt động là một nơi giữ kiểu thật sự ở thời gian biên dịch. Khi chúng ta viết mã để tạo thể hiện của lớp `Queue` generic, chúng ta phải cung cấp kiểu cho T như trong ví dụ trước. Hơn nữa, nếu chúng ta nhìn các phương thức trong lớp `Queue<T>` chỉ ra T là kiểu tham số hay kiểu giá trị trả về.

```
public void Enqueue( T item );  
public T Dequeue();
```

Kiểu tham số T sẽ được thay thế với kiểu chúng ta chỉ ra khi khai báo hàng đợi. Trình biên dịch bây giờ đủ thông tin để thực hiện kiểm tra kiểu khi chúng ta xây dựng ứng dụng và có thể bắt các lỗi sớm hơn.

Trình biên dịch cho phép chúng ta chỉ ra kiểu giá trị bất kì cho T như trong các ví dụ sau:

```
struct Person  
{  
    ...  
}  
...  
Queue<int> intQueue = new Queue<int>();  
Queue<Person> personQueue = new Queue<Person>();  
Queue<Queue<int>> queueQueue = new Queue<Queue<int>>();
```

Hai ví dụ đầu tiên tạo hàng đợi kiểu giá trị, trong khi ví dụ thứ ba tạo hàng đợi của các hàng đợi số nguyên. Nếu chúng ta lấy biến intQueue làm một ví dụ, trình biên dịch sẽ tạo các phiên bản sau cho phương thức Enqueue and Dequeue:

```
public void Enqueue( int item );  
public int Dequeue();
```

Một lớp generic cũng cho phép nhiều tham số kiểu. Ví dụ, lớp generic System.Collections.Generic.Dictionary cần hai tham số: một cho khóa, một cho giá trị. Định nghĩa sau khai báo tham số nhiều kiểu:

```
public class Dictionary<T, U>
```

Một Dictionary cung cấp một tập hợp các cặp khóa/giá trị. Chúng ta lưu trữ giá trị (kiểu U) kết hợp với khóa (kiểu T) và lấy chúng theo khóa cần tìm. Lớp Dictionary cung cấp một chỉ mục cho phép chúng ta truy xuất các phần tử dùng chú thích mảng. Nó định nghĩa như sau:

```
public virtual U this[ T key ] { get; set; }
```

Chú ý rằng chỉ mục truy xuất giá trị kiểu U bởi dùng khóa kiểu T. Để tạo và sử dụng một Dictionary tên directory chứa giá trị kiểu Person và khóa kiểu string, chúng ta có thể dùng mã sau:

```
struct Person
{
    ...
}
...
Dictionary<string, Person> directory = new Dictionary<string,
Person>();
Person p = new Person();
directory["Ten"] = p;
...
p = directory["Ten"];
```

d) Từ điển (Dictionary)

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

Ví dụ viết chương trình đếm số từ trong một chuỗi:

```
1: using System;
2: using System.Collections.Generic;
3: using System.Text.RegularExpressions;
4: class DictionaryDemo
5: {
6:     static Dictionary<string, int> CountWords(string text)
7:     {
```

```

8:         Dictionary<string, int> frequencies = new
           Dictionary<string, int>();
9:
10:        string[] words = Regex.Split(text, @"\W+");
11:
12:        foreach (string word in words)
13:        {
14:            if (frequencies.ContainsKey(word))
15:            {
16:                frequencies[word]++;
17:            }
18:            else
19:            {
20:                frequencies[word] = 1;
21:            }
22:        }
23:        return frequencies;
24:    }
25:
26:    static void Main()
27:    {
28:        string text = "Lap trinh huong doi tuong";
29:        Dictionary<string, int> frequencies =
           CountWords(text);
30:        foreach (KeyValuePair<string, int> entry in
           frequencies)
31:        {
32:            string word = entry.Key;
33:            int frequency = entry.Value;
34:            Console.WriteLine("{0}: {1}", word, frequency);
35:        }
36:    }

```

8.4 Thuộc tính (attribute) tùy chọn

a) Viết thuộc tính tùy chọn

Để hiểu cách viết thuộc tính tùy chọn, ta cần xem trình biên dịch làm gì khi nó gặp một mục trong mã được đánh dấu với một attribute. Giả sử ta có thuộc tính khai báo như sau:

```

[TenTruong("SoCMND")]
public string SoCMND
{

```

```
get {  
    // vv...
```

Như ta thấy thuộc tính SoCMND có một thuộc tính TenTruong, trình biên dịch sẽ nối chuỗi attribute với tên này thành TenTruongAttribute, sau đó tìm trong tất cả các namespace lớp có tên này. Tuy nhiên nếu ta đánh dấu một mục với một thuộc tính mà tên của nó có phần cuối là attribute thì trình biên dịch sẽ không thêm chuỗi attribute lần nữa ví dụ:

```
[TenTruongattribute("SoCMND")]  
public string SoCMND  
{  
  
    // vv...
```

Nếu trình biên dịch không tìm thấy một lớp thuộc tính tương ứng, hoặc thấy nhưng cách mà ta dùng thuộc tính không phù hợp với thông tin trong lớp thuộc tính, thì trình biên dịch sẽ sinh ra lỗi.

b) Các lớp thuộc tính tùy chọn

Giả sử ta đã định nghĩa một thuộc tính TenTruong như sau:

```
[AttributeUsage(AttributeTargets.Property,  
    AllowMultiple=false,  
    Inherited=false)]  
public class TenTruongAttribute: Attribute  
{  
    private string ten;  
    public TenTruongAttribute(string ten)  
    {  
        this.ten = ten;  
    }  
}
```

Điều đầu tiên ta chú ý là lớp attribute được đánh dấu với một thuộc tính “AttributeUsage”. AttributeUsage chỉ định các mục nào trong mã của chúng ta áp dụng thuộc tính tùy chọn. Thông tin này được cho bởi thông số đầu tiên. Thông số này

là một kiểu liệt kê `attributeTargets`. Trong ví dụ trên ta chỉ định thuộc tính `TenTruong` chỉ được áp dụng đến các thuộc tính.

Định nghĩa của kiểu liệt kê `attributeTargets` là:

```
public enum attributeTargets
{
    All = 0x00003FFF,
    Assembly = 0x00000001,
    Class = 0x00000004,
    Constructor = 0x00000020,
    Delegate = 0x00001000,
    Enum = 0x00000010,
    Event = 0x00000200,
    Field = 0x00000100,
    Interface = 0x00000400,
    Method = 0x00000040,
    Module = 0x00000002,
    Parameter = 0x00000800,
    Property = 0x00000080,
    ReturnValue = 0x00002000,
    Struct = 0x00000008
}
```

Khi áp dụng thuộc tính đến các phần tử chương trình, ta đặt thuộc tính trong ngoặc vuông ngay trước phần tử. Một thuộc tính có thể được áp dụng đến một `Assembly` nhưng cần được đánh dấu với từ khoá `Assembly`:

```
[assembly: SomeAssemblyattribute(Parameters)]
```

Để kết hợp nhiều kiểu khác nhau trên một phần tử nào đó, ta viết như sau:

```
[attributeUsage(attributeTargets.Property | attributeTargets.Field,
    AllowMultiple=false,
    Inherited=false)]
public class TenTruongattribute: attribute
```

Ta cũng có thể dùng `attributeTargets.All` để áp dụng thuộc tính cho tất cả các trường hợp. Thuộc tính `attributeUsage` còn chứa hai thông số khác là `AllowMultiple` and `Inherited`, chỉ định với cú pháp khác của `<tên_thuộc_tính>=<giá_trị_thuộc_tính>`.

Thông số này là tùy chọn, thông số AllowMultiple chỉ định một attribute có thể áp dụng nhiều hơn một lần đến cùng một mục. Nếu thiết đặt là false thì trình biên dịch sẽ thông báo lỗi nếu nó thấy:

```
[TenTruong("SOCMND")]  
[TenTruong("SOBaoHiem")]  
public string SOCMND  
{  
  
// vv...
```

Nếu thông số Inherited là true, thì một thuộc tính có thể áp dụng đến một lớp hay một giao diện cũng sẽ được áp dụng đến tất cả các lớp hay giao diện được kế thừa. Nếu thuộc tính được áp dụng cho phương thức hay thuộc tính thì nó tự động áp dụng đến bất kì phương thức hay thuộc tính nào được khai báo override.

c) Đặc tả các thông số thuộc tính

Ta sẽ kiểm tra làm thế nào ta có thể chỉ định thông số cho thuộc tính tùy chọn, Khi trình biên dịch gặp lệnh:

```
[TenTruong("SOCMND")]  
public string SOCMND  
{  
...
```

Nó kiểm tra thông số truyền vào attribute, trong trường hợp này là chuỗi và tìm phương thức tạo lập của thuộc tính mà nhận các thông số này, nếu thấy thì không có vấn đề gì ngược lại trình biên dịch sẽ sinh ra lỗi.

d) Các thông số tùy chọn

Ta thấy thuộc tính attributeUsage có một cú pháp cho phép thêm các giá trị vào trong thuộc tính. Cú pháp này có liên quan đến việc chỉ định tên của các thông số được chọn. Giả sử ta cập nhật lại thuộc tính SoCMND như sau:

```
[TenTruong("SoCMND ", ChuThich="Day la truong khoa")]  
public string SoCMND
```

```
{  
...  
}
```

Trong trường hợp này, trình biên dịch sẽ nhận ra `<tên_tham_số>=` cú pháp của thông số thứ hai. Nó sẽ tìm một thuộc tính public (hoặc field) của tên đó mà nó có thể dùng để đặt giá trị của thông số này. Nếu ta muốn đoạn mã trên làm việc ta thêm mã sau vào `TenTruongattribute`:

```
[attributeUsage(attributeTargets.Property,  
AllowMultiple=false,  
Inherited=false)]  
public class TenTruongattribute: attribute  
{  
    private string chuthich;  
    public string ChuThich  
{  
        ...  
    }  
}
```

8.5 Reflection

Reflection là một kỹ thuật cho phép ta tìm ra thông tin về các kiểu dữ liệu trong chương trình. Hầu hết những lớp này nằm trong namespace `System.Reflection`.

Lớp `System.Type` cho phép ta truy nhập thông tin liên quan đến việc định nghĩa bất kỳ kiểu dữ liệu nào.

a) Lớp `System.Type`

Ta dùng lớp `Type` để lấy tên của một kiểu:

```
Type t = typeof(double);
```

Mặc dù ta cho rằng `Type` là một lớp nhưng thực sự nó là một lớp cơ sở trừu tượng, bất cứ khi nào ta khởi tạo một đối tượng `Type` ta thực sự khởi tạo một lớp dẫn xuất của `Type`. `Type` có một lớp dẫn xuất đáp ứng mỗi kiểu dữ liệu. Có 3 cách lấy một tham chiếu `Type` cho kiểu dữ liệu bất kỳ:

- Dùng tác tử `typeof`, tác tử này lấy tên của kiểu như là thông số.
- Dùng phương thức `GetType()`, mà tất cả các lớp kế thừa từ `System.Object`:

```
double d = 10;
Type t = d.GetType();
```

GetType() hữu ích khi ta có một tham chiếu đối tượng và không chắc đối tượng thực sự là thể hiện của lớp nào.

- Ta cũng có thể gọi phương thức static của lớp type, GetType():

```
Type t = Type.GetType("System.Double");
```

b) Các thuộc tính của Type

Một số thuộc tính lấy chuỗi chứa các tên khác nhau kết hợp với lớp:

Thuộc tính	Trả về
Name	tên của kiểu dữ liệu
FullName	tên đầy đủ bao gồm cả namespace
Namespace	tên namespace của kiểu dữ liệu.

Có thể lấy tham chiếu đến kiểu đối tượng của các lớp liên quan:

Thuộc tính	Kiểu tham chiếu trả về tương ứng
BaseType	kiểu cơ sở trực tiếp của kiểu này
UnderlyingSystemType	kiểu mà kiểu này ánh xạ trong thời gian chạy.NET

Một số thuộc tính luận lý kiểm tra kiểu, ví dụ là một lớp hay một kiểu liệt kê... những thuộc tính này bao gồm: **IsAbstract**, **IsArray**, **IsClass**, **IsEnum**, **IsInterface**, **IsPointer**, **IsPrimitive**, **IsPublic**, **IsSealed**, and **IsValueType**.

Ví dụ dùng kiểu dữ liệu cơ bản:

```
Type intType = typeof(int);
Console.WriteLine(intType.IsAbstract); // false
Console.WriteLine(intType.IsClass); // false
Console.WriteLine(intType.IsEnum); // false
Console.WriteLine(intType.IsPrimitive); // true
Console.WriteLine(intType.IsValueType); // true
```

hoặc dùng lớp Vector:

```
Type intType = typeof(Vector); Console.WriteLine(intType.IsAbstract);  
// false  
Console.WriteLine(intType.IsClassembly); // true  
Console.WriteLine(intType.IsEnum); // false  
Console.WriteLine(intType.IsPrimitive); // false  
Console.WriteLine(intType.IsValueType); // false
```

c) Các phương thức

Hầu hết các phương thức của System.Type được sử dụng để chứa chi tiết các thành viên của kiểu dữ liệu tương ứng - hàm tạo lập, thuộc tính, phương thức, sự kiện... có nhiều phương thức nhưng tất cả chúng đều theo nền chung. Ví dụ, có hai phương thức mà nhận chi tiết phương thức của kiểu dữ liệu: GetMethod() và GetMethods().

GetMethod() trả về một tham chiếu đến đối tượng System.Reflection. MethodInfo chứa chi tiết của một phương thức. GetMethods() trả về một mảng tham chiếu.

Ví dụ phương thức GetMethods() không lấy thông số nào và trả về chi tiết của tất cả phương thức thành viên của kiểu dữ liệu:

```
Type t = typeof(double);  
MethodInfo [] methods = t.GetMethods();  
foreach (MethodInfo nextMethod in methods)  
{  
    ...  
}
```

Kiểu đối tượng trả về	Các phương thức (phương thức số nhiều (có 's' ở cuối tên) trả về một mảng)
ConstructorInfo	Gvvonstructor(), Gvvonstructors()
EventInfo	GetEvent(), GetEvents()
FieldInfo	GetField(), GetFields()
InterfaceInfo	GetInterface(), GetInterfaces()
MemberInfo	GetMember(), GetMembers()
MethodInfo	GetMethod(), GetMethods()

Kiểu đối tượng trả về	Các phương thức (phương thức số nhiều (có 's' ở cuối tên) trả về một mảng)
PropertyInfo	GetProperty(), GetProperties()

Phương thức GetMember() và GetMembers() trả về chi tiết của bất kì hay tất cả thành viên của kiểu dữ liệu không cần biết đó là hàm tạo lập hay thuộc tính phương thức.

Chương trình minh họa lấy tên của tất cả các phương thức của một lớp dùng reflection:

```

1: using System;
2: using System.Reflection;
3: public interface IGiaoDien1
4: {
5:     void PhuongThucA();
6: }
7: public interface IGiaoDien2
8: {
9:     void PhuongThucB();
10: }
11: public class ViDu: IGiaoDien1, IGiaoDien2
12: {
13:     public enum KieuLietKe { }
14:     public int nguyen;
15:     public string chuoi;
16:     public void PhuongThuc(int p1, string p2)
17:     {
18:     }
19:     public int ThuocTinh
20:     {
21:         get { return nguyen; }
22:         set { nguyen = value; }
23:     }
24:     void IGiaoDien1.PhuongThucA() { }
25:     void IGiaoDien2.PhuongThucB() { }
26: }
27: public class MainClass
28: {
29:     public static void Main(string[] args)
30:     {
31:         ViDu f = new ViDu();
32:         Type t = f.GetType();
33:         MethodInfo[] mi = t.GetMethods();
34:         foreach (MethodInfo m in mi)

```

```
35:             Console.WriteLine("Phuong Thuc: {0}", m.Name);
36:         }
37:     }
```

8.6 Thao tác tập tin qua luồng

Trong C#, để thao tác dữ liệu trên tập tin, chúng ta có thể sử dụng qua luồng (Stream). Stream là dòng dữ liệu chảy đi. Đây là một thực thể có khả năng nhận hoặc tạo ra một nhóm dữ liệu. System.IO.Stream là một lớp abstract định nghĩa một số thành viên chịu hỗ trợ việc đọc/viết đồng bộ hoặc không đồng bộ đối với khối trữ tin.

Vì Stream là một lớp abstract, nên bạn chỉ có thể làm việc với những lớp được dẫn xuất từ Stream. Các lớp kế thừa của Stream tượng trưng dữ liệu như là một dòng dữ liệu thô dạng bytes (thay vì dữ liệu dạng văn bản). Ngoài ra, các lớp được dẫn xuất từ Stream hỗ trợ việc truy tìm nghĩa là một tiến trình nhận lấy và điều chỉnh vị trí trên một luồng.

Ví dụ minh họa đọc nội dung tập tin văn bản vào trong mảng một chiều a:

```
1: using System;
2: using System.IO;
3: namespace ConsoleApplication6.File
4: {
5:     public class SoNguyenText
6:     {
7:         int[] a = new int[100]; //khai báo một mảng số nguyên
                                //100 phần tử
8:         int size; //kích thước thật sự của mảng đang sử dụng
9:         public SoNguyenText(){}
10:        public void TaoFile()
11:        {
12:            //Tạo một tập tin data.txt để lưu trữ dữ liệu
                                //nhập vào
13:            FileStream fs = new FileStream("data.txt",
                                FileMode.Create);
14:            //Tạo một luồng thông tin sử dụng StreamWriter(
                                //sử dụng luồng Stream
15:            //cho phép thao tác tập tin đơn giản hơn
16:            StreamWriter br = new StreamWriter(fs);
17:            for (int i = 0; i < 10; i++)
18:                br.WriteLine(i); //Ghi giá trị lên tập tin,
```

```
        tương tự như ghi ra màn hình
19:        br.Close();
20:        fs.Close(); //Thực hiện xong, đóng tập tin
21:    }
22:    public void DocFile()
23:    {
24:        //Mở tập tin để đọc dữ liệu
25:        FileStream fs = new FileStream("data.txt",
        FileMode.Open);
26:        //Tạo luồng để đọc thông tin từ tập tin
27:        StreamReader br = new StreamReader(fs);
28:        string text;
29:        size = 0;
30:        //Đọc các hàng trong tập tin bằng hàng ReadLine()
31:        //Nếu kết thúc tập tin, hàm sẽ trả về giá trị
        null
32:        while ((text = br.ReadLine()) != null)
33:        {
34:            a[size] = int.Parse(text); //chuyển chuỗi
            text thành số nguyên
35:            size++; //tăng kích thước của mảng lên một
36:        }
37:        br.Close();
38:        fs.Close(); //Thực hiện xong, đóng tập tin
39:    }
40:    public override string ToString()
41:    {
42:        //xuất mảng có chiều dài size
43:        string kq = "";
44:        for (int i = 0; i < size; i++)
45:            kq += " " + a[i];
46:
47:        return kq;
48:    }
49:    static void Main()
50:    {
51:        SoNguyenText t = new SoNguyenText();
52:        t.DocFile();
53:        Console.WriteLine("Xuat mang ");
54:        Console.WriteLine(t);
55:    }
56: }
57: }
```

Nội dung của tập tin data.txt được lưu trong thư mục bin\Debug của dự án như sau:


```
1
8
7
9
5
Kết quả thực hiện chương trình:
Xuất mang
1 8 7 9 5
```

Hướng dẫn phần thực hành

- Sinh viên hoàn thành các chương trình mẫu được minh họa trên lớp.
- Sinh viên hoàn thành các bài thực hành từ bài Lab1 đến Lab5. Các bài thực hành này sẽ được cung cấp theo các buổi thực hành.

Bài thực hành 1

1. Viết chương trình nhập vào hai chuỗi ký tự và hiển thị ra màn hình.
2. Viết chương trình nhập vào hai chuỗi sa và sb và xuất ra màn hình dùng toán tử + (sa+sb).
3. Chuyển chương trình * thành chương trình sau và đánh giá kết quả.
4. Viết chương trình hiển thị tam giác các ngôi sao ra màn hình.
5. Viết chương trình tính tổng n số nguyên âm trong đoạn $[-n, 0]$ với n nhập vào từ bàn phím. Thực hiện:
 - Nhập vào một số nguyên n
 - Kiểm tra số nguyên nhập vào là số âm hay không?
 - Nếu n là số âm thì tính tổng các phần tử rồi gán vào trong biến kết quả. Ngược lại thông báo nhập liệu không hợp lệ.
 - Hiển thị kết quả thực hiện

```
using System;
namespace NguyenLyLapTrinhII
{
    class Class1
    {
        static void Main(string[] args)
        {
            int n;
            Console.Write("Nhap vao so nguyen am n = ");
            n = int.Parse(Console.ReadLine());
            if(n > 0)
            {
                Console.WriteLine("Phai nhap vao so nguyen am ");
            }
        }
    }
}
```

```
        return;
    }
    int kq = 0;
    for(int i = n; i <=0; i++)
        kq +=i;
    Console.WriteLine(" Ket qua = " + kq);
    Console.Read();
}
}
```

Thay đổi quả câu lệnh

```
Console.WriteLine(" Ket qua = " + kq);
```

thành câu lệnh sau:

```
Console.WriteLine(" Ket qua tong so nguyen am {0} = {1}",n,kq);
```

Chạy, xem kết quả và so sánh chương trình.

6. Viết chương trình ở bài tập 5 dùng vòng While .
7. Viết chương trình kiểm tra một số n có phải là số nguyên tố hay không?
8. Viết chương trình tính n giai thừa?
9. Viết chương trình tính tổng các số lẻ và số chẵn trong khoảng[0,n] với n nhập vào từ bàn phím.
10. Chuyển tất cả các chương trình trên vào trong các thủ tục hay hàm.
11. Viết chương trình nhập vào một mảng các số nguyên, xuất các phần tử trong mảng và tính tổng các phần tử này. Thực hiện:
 - Khai báo một mảng a kiểu số nguyên với kích thước là 5.
 - Nhập các giá trị của mảng.
 - Tính tổng
 - Xuất ra màn hình.

```
using System;
```

```
namespace NguyenLyLapTrinhII
{
    class Class1
    {
        static void Main(string[] args)
        {
            int []a = new int[5];
            Console.WriteLine("Nhap vao cac phan tu ");
            for(int i =0; i<5; i++)
            {
                Console.Write("Phan tu a[{0}]= ",i);
                a[i] = int.Parse(Console.ReadLine());
            }

            int tong = 0;
            for(int i = 0; i < a.Length; i++)
                tong +=a[i];
            //////////Xuat mang

            Console.WriteLine("Danh sach cac phan tu ");
            for(int i =0; i<5; i++)
            {
                Console.Write(" {0} ",a[i]);
            }
            Console.WriteLine(" Tong cac phan tu = {0}",tong);
            Console.Read();
        }
    }
}
```

Bài tập:

Viết chương trình trên với mảng có kích thước do người dùng nhập vào.

Viết chương trình trên với kiểu dữ liệu của mảng là double, long, string....

Bài thực hành 2

Sau khi thực hành 2 sinh viên cần nắm vững phần kỹ thuật và kiến thức về:

- Cách tạo một lớp mới trong ngôn ngữ C#.
- Cách tạo một đối tượng trong ngôn ngữ C#.
- Phương thức tạo lập.
- Quá tải hàm.

- Con trỏ this.
 - Phương pháp truy xuất.
 - Các truyền tham số trong C#
1. Viết chương trình xử lý các phép toán trên phân số.
 2. Viết chương trình xử lý các phép toán trên số phức.
 3. Viết chương trình quản lý danh bạ điện thoại trong đó người dùng có thể thực hiện các yêu cầu sau:
 - Nhập một khách hàng mới.
 - Xóa khách hàng.
 - Tìm kiếm khách hàng theo tên, địa chỉ hay một số thông tin khác liên quan đến khách hàng.
 - Tìm khách hàng theo số điện thoại.
 - Xuất danh sách khách hàng theo Alphabet
 - Sắp xếp danh sách khách hàng theo tên.
 4. Viết chương trình quản lý lương nhân viên của công ty A với các yêu cầu sau:
 - Nhân viên được chia thành 3 loại: nhân viên ăn lương theo tháng, nhân viên ăn lương theo giờ và nhân viên hợp đồng.
 - Mỗi nhân viên có một hệ số lương khác nhau nhau. Lương của nhân viên được tính theo công thức:
 - $Lương = số\ ngày\ công \times hệ\ số\ lương + thưởng.$
 - Hệ thống cần lưu trữ một số các thông tin của nhân viên như tên, ngày sinh....
 - Xây dựng các dữ liệu và phương thức cần thiết cho hệ thống trên(tương tự như bài 3).

5. Viết chương trình quản lý các mặt hàng của một công ty A. Chương trình cho phép người sử dụng nhập, tìm kiếm, hiển thị các mặt hàng. Hệ thống cũng cần lưu trữ giá của các mặt hàng.
6. Viết chương trình quản lý khách hàng của một công ty B như sau:
 - Khi một khách hàng mua hàng của công ty hệ thống cần lưu trữ thông tin khách hàng.
 - Lưu trữ các mặt hàng mà khách hàng mua
 - Lưu trữ thời gian mua hàng.
 - Chương trình có khả năng tìm kiếm, nhập, xuất và hiển thị khách hàng....
7. Viết chương trình quản lý các mặt hàng của một cửa hàng bán sách A như sau:
 - Các mặt hàng được phân thành các loại; sách, tạp chí, văn phòng phẩm.
 - Khi khách hàng mua hàng hệ thống cần lưu trữ thông tin của khách hàng.
 - Xây dựng chương trình cho phép tìm kiếm khách hàng theo
 - Tên khách hàng
 - Tìm khách hàng theo sản phẩm
 - Tìm theo số tiền của khách hàng mua.
 - Giả sử công ty có chương trình khuyến mãi như sau: cty sẽ thưởng một mặt hàng trị giá 200.000 cho tất cả khách hàng mua có của cty có số tiền > 2triệu đồng. Viết phương thức hiển thị danh sách các khách hàng trên.
8. Viết chương trình quản lý các chuyến bay của một hãng hàng không A với các yêu cầu:
 - Xem thông tin các chuyến bay theo ngày, giờ, theo điểm xuất phát hay đích đến.
 - Tìm kiếm thông tin chuyến bay theo các yêu cầu tương tự như trên.

- Đặt vé.
9. Viết chương trình quản lý tình trạng của nhân viên của một cty A với các yêu cầu sau:

Xem tình trạng của nhân viên:

- Theo ngày hợp đồng
 - Nhân viên bị sa thải hay không? nếu có thì cần biết lý do và thời gian.
 - Nếu một nhân viên đã chuyển công việc thì hệ thống cần thông báo ngày, lý do. công việc trước đây làm gì, hiện tại như thế nào?
 - Sinh viên có thể bổ sung các phương thức cần thiết.
10. Viết chương trình thực hiện cơ chế Stack lưu trữ các số nguyên như sau:
- Phần tử nào được đưa vào trong Stack sau cùng thì sẽ được lấy ra trước(Một phần tử đưa vào Stack luôn được chèn đầu)
 - Hiện thị các phần tử trong Stack
 - Lấy một phần tử trong Stack
 - Xóa một phần tử trong Stack
 - Xóa tất cả phần tử trong Stack
11. Viết chương trình thực hiện cơ chế Queue lưu trữ các số nguyên như sau:
- Phần tử nào được đưa vào trong Queue trước thì sẽ được lấy ra trước(Một phần tử đưa vào Queue luôn được chèn cuối)
 - Hiện thị các phần tử trong Queue
 - Lấy một phần tử trong Queue
 - Xóa một phần tử trong Queue
 - Xóa tất cả phần tử trong Queue

12. Tạo một lớp tập hợp các số nguyên với các các phép toán sau:

- Chèn 1 phần tử vào trong tập hợp
- Chèn một tập hợp vào trong một tập hợp
- Xóa một phần tử trong tập hợp theo vị trí hay giá trị của phần tử.
- Xóa tất cả tập hợp
- Kiểm tra một phần tử có chứa trong tập hợp hay không
- Đếm số phần tử trong tập hợp
- Xác định số phần tử tối đa mà tập hợp có thể chứa
- Lấy hay gán giá trị của một phần tử tại một vị trí bất kỳ
- Lấy vị trí của một phần tử trong tập hợp
- Chuyển một tập hợp sang mảng một chiều
- Kiểm tra tập hợp có tương đương với một tập hợp khác hay không
- Sắp xếp tập hợp

13. Giả sử A và B là 2 tập hợp. Viết chương trình thực hiện các phép toán trên tập hợp các số nguyên:

- Giao 2 tập hợp A và B
- Giao 2 tập hợp A và phần bù của B
- Hợp 2 tập hợp A và B
- Hiệu hai tập hợp A và B
- Hiệu hai tập hợp A và phần bù của B

Bài thực hành 3

Sau khi thực hành bài lab4 sinh viên cần hiểu và sử dụng được:

- Tính kế thừa
 - Tính đa hình
 - Sử dụng con trỏ this nhằm truy cập dữ liệu hiệu quả
 - Phương pháp truy xuất public, private và protected
1. Chuyển đổi các bài tập trong bài thực hành 2 dùng các đặc trưng kế thừa, đóng gói, đa hình.
 2. Viết chương trình đổi một số thập phân sang dạng nhị phân và thập lục phân.
 3. Viết chương trình quản lý các đối tượng hình học như hình tròn, hình vuông, hình chữ nhật, tam giác.
 4. Viết chương trình quản lý sinh viên trong đó sinh viên được chia thành 2 loại sinh viên chính quy và sinh viên cao đẳng.
 5. Viết chương trình quản lý các môn học trong đó môn học được chia thành 2 loại: môn học đại cương và môn học chuyên ngành.
 6. Viết chương trình quản lý đăng ký học phần cho sinh viên.

Phụ lục

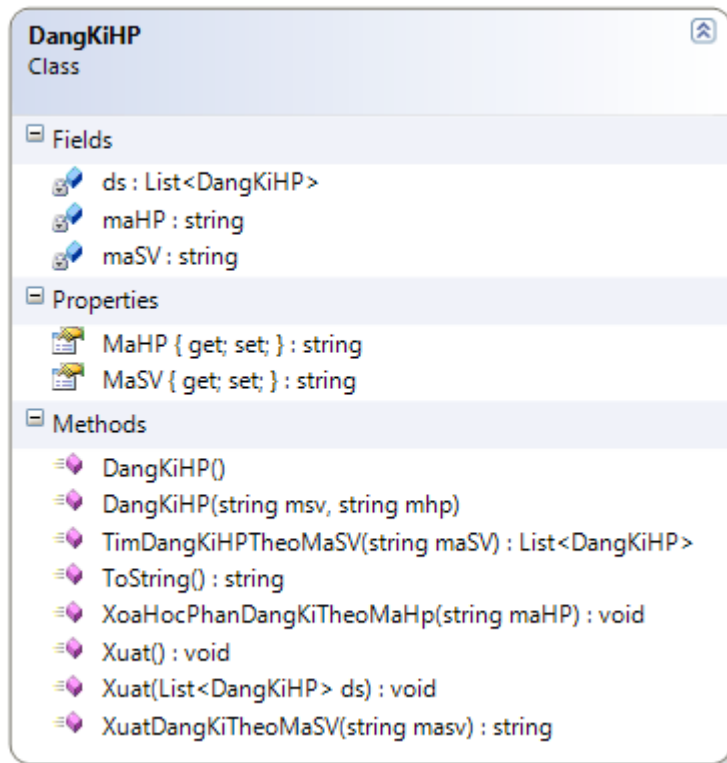
Xây dựng ứng dụng hoàn chỉnh áp dụng các kỹ thuật trong lập trình hướng đối tượng.

Xây dựng chương trình quản lý đăng ký học phần của sinh viên. Ứng dụng này chỉ giới hạn trong phạm vi nhỏ với mục đích minh họa các tiếp cận và xây dựng ứng dụng theo kỹ thuật lập trình hướng đối tượng. Sinh viên có thể bổ sung và mở rộng thêm các chức năng chưa xây dựng trong chương trình.

- Ứng dụng cho phép dữ liệu được lưu trữ trên tập tin hay nhập tĩnh tùy theo yêu cầu.
- Các kiểu mẫu dữ liệu kết xuất được lưu trữ trên tập tin.

Lược đồ lớp được tổ chức như sau:





Tạo một dự án kiểu thư viện để lưu trữ các lớp sau:

Nội dung của lớp sinh viên

```
1: using System;
2: using System.Collections.Generic;
3: using System.Text;
4:
5: namespace QuanLiHocPhan
6: {
7:     public class SinhVien
8:     {
9:         #region Khai bao du lieu
10:         string maSV;
11:
12:         public string MaSV
13:         {
14:             get { return maSV; }
15:             set { maSV = value; }
16:         }
17:         string hoTenSV;
18:
19:         public string HoTenSV
```

```
20:         {
21:             get { return hoTenSV; }
22:             set { hoTenSV = value; }
23:         }
24:         string tenLop;
25:
26:         public string TenLop
27:         {
28:             get { return tenLop; }
29:             set { tenLop = value; }
30:         }
31:         #endregion ket thuc khai bao du lieu
32:         #region Phuong thuc tao lap
33:         public SinhVien() { }
34:         public SinhVien(string m, string t, string l)
35:         {
36:             MaSV = m;
37:             HoTenSV = t;
38:             TenLop = l;
39:         }
40:         #endregion Phuong thuc tao lap
41:         static List<SinhVien> ds =
42:             DuLieu.TheHien.DocSinhViens();
43:         public static SinhVien TimTheoMa(string ma)
44:         {
45:             //foreach (SinhVien s in ds)
46:             //    if (s.MaSV == ma) return s;
47:             //return null;
48:             return ds.Find(delegate(SinhVien s)
49:             {
50:                 return s.MaSV == ma;
51:             });
52:         }
53:         //public static List<SinhVien> TimTheoTenLop(string
54:             ten)
55:         //{
56:         //    List<SinhVien> kq = new List<SinhVien>();
57:         //    foreach (SinhVien s in ds)
58:         //        if (s.TenLop == ten)
59:         //            kq.Add(s);
60:         //    return kq;
61:         //}
62:         public static List<SinhVien> TimTheoTenLop(string ten)
63:         {
64:             return ds.FindAll(delegate(SinhVien s)
```

```
64:         {
65:             return s.TenLop == ten;
66:         }
67:     );
68: }
69: public static void Xuat(List<SinhVien> ds)
70: {
71:     foreach (SinhVien s in ds)
72:         Console.WriteLine(s);
73: }
74: public static List<SinhVien> DanhSachSinhVien()
75: {
76:     return ds;
77: }
78: public override string ToString()
79: {
80:     return MaSV + "\t|" + hoTenSV + "\t|" + TenLop;
81: }
82: }
83:
84: }
```

Nội dung lớp học phần:

```
1: using System;
2: using System.Collections.Generic;
3: using System.Text;
4:
5: namespace QuanLiHocPhan
6: {
7:     public class HocPhan
8:     {
9:         #region Khai bao du lieu
10:        string maHP;
11:
12:        public string MaHP
13:        {
14:            get { return maHP; }
15:            set { maHP = value; }
16:        }
17:        string tenHP;
18:
19:        public string TenHP
20:        {
21:            get { return tenHP; }
22:            set { tenHP = value; }
```

```
23:     }
24:     int soTC;
25:
26:     public int SoTC
27:     {
28:         get { return soTC; }
29:         set { soTC = value; }
30:     }
31: #endregion ket thuc khai bao du lieu
32: #region Phuong thuc tao lap
33: public HocPhan(){ }
34: public HocPhan(string m, string t, int tc)
35: {
36:     MaHP = m;
37:     TenHP = t;
38:     SoTC = tc;
39: }
40: #endregion Phuong thuc tao lap
41: static List<HocPhan> ds = DuLieu.TheHien.DocHocPhans();
42: public static HocPhan TimTheoMa(string ma)
43: {
44:     return ds.Find(delegate(HocPhan s)
45:     {
46:         return s.MaHP == ma;
47:     }
48:     );
49: }
50: public static void Xuat(List<HocPhan> ds)
51: {
52:     foreach (HocPhan s in ds)
53:         Console.WriteLine(s);
54: }
55: public static void Xuat()
56: {
57:     Xuat(ds);
58: }
59: static public event EventHandler Xoa;
60: static public void OnXoa(string MaHP)
61: {
62:     if (Xoa != null)
63:         Xoa(MaHP, null);
64: }
65: static public void XoaHocPhanTheoMa(string maHP)
66: {
67:     HocPhan h = TimTheoMa(maHP);
68:     OnXoa(h.MaHP);
```

```
69:         ds.Remove(h);
70:     }
71:     public override string ToString()
72:     {
73:         return MaHP + "\t|" + TenHP + "\t|" + SoTC;
74:     }
75: }
76: }
```

Nội dung lớp đăng kí học phần

```
1: using System;
2: using System.Collections.Generic;
3: using System.Text;
4:
5: namespace QuanLiHocPhan
6: {
7:     public class DangKiHP
8:     {
9:         string maSV;
10:
11:         public string MaSV
12:         {
13:             get { return maSV; }
14:             set { maSV = value; }
15:         }
16:         string maHP;
17:
18:         public string MaHP
19:         {
20:             get { return maHP; }
21:             set { maHP = value; }
22:         }
23:         public DangKiHP(){}
24:         public DangKiHP(string msv, string mhp) {
25:             MaSV = msv;
26:             MaHP = mhp;
27:         }
28:         static List<DangKiHP> ds = DuLieu.TheHien.DocDangKiHPs();
29:         public static void Xuat(List<DangKiHP> ds)
30:         {
31:             foreach (DangKiHP s in ds)
32:                 Console.WriteLine(s);
33:         }
34:
35:         public static List<DangKiHP>
```



```

        TimDangKiHPTheoMaSV(string maSV)
36:    {
37:        return ds.FindAll(delegate(DangKiHP d)
38:        {
39:            return d.MaSV == maSV;
40:        }
41:        );
42:    }
43:    public static string XuatDangKiTheoMaSV(string masv)
44:    {
45:        List<DangKiHP> kq = TimDangKiHPTheoMaSV(masv);
46:        string hienthi = ThuVien.GetFromFile("template.txt");
47:        string dshp = "";
48:        int i = 1;
49:        int tong = 0;
50:        foreach (DangKiHP d in kq)
51:        {
52:            HocPhan h = HocPhan.TimTheoMa(d.MaHP);
53:            dshp += i++.ToString() + ")" + h + "\n";
54:            tong += h.SoTC;
55:        }
56:        SinhVien s = SinhVien.TimTheoMa(masv);
57:        return string.Format(hienthi, s.MaSV, s.HoTenSV, dshp, tong);
58:    }
59:    public static void XoaHocPhanDangKiTheoMaHp(string maHP)
60:    {
61:        ds.RemoveAll(delegate(DangKiHP d)
62:        {
63:            return d.MaHP == maHP;
64:        }
65:        );
66:    }
67:    public static void Xuat()
68:    {
69:        Xuat(ds);
70:    }
71:    public override string ToString()
72:    {
73:        return SinhVien.TimTheoMa(MaSV) + "\t" +
            HocPhan.TimTheoMa(MaHP);
74:    }
75:    }
76: }

```

Xây dựng lớp kết nối dữ liệu:

```
1: using System;
2: using System.Collections.Generic;
3: using System.Text;
4: using System.IO;
5: namespace QuanLiHocPhan
6: {
7:     public abstract class KetNoiDuLieu
8:     {
9:         public abstract List<SinhVien> DocSinhViens();
10:        public abstract bool XoaSinhViens(SinhVien s);
11:        public abstract bool ThemSinhViens(SinhVien s);
12:        public abstract bool SuaSinhViens(SinhVien s);
13:
14:        public abstract List<HocPhan> DocHocPhans();
15:        public abstract bool XoaHocPhans(HocPhan s);
16:        public abstract bool ThemHocPhans(HocPhan s);
17:        public abstract bool SuaHocPhans(HocPhan s);
18:
19:        public abstract List<DangKiHP> DocDangKiHPs();
20:        public abstract bool XoaDangKiHPs(DangKiHP s);
21:        public abstract bool ThemDangKiHPs(DangKiHP s);
22:        public abstract bool SuaDangKiHPs(DangKiHP s);
23:
24:    }
25:    public class DuLieu
26:    {
27:        static KetNoiDuLieu thehien;
28:        public static KetNoiDuLieu TheHien
29:        {
30:            get
31:            {
32:                return thehien;
33:            }
34:        }
35:        static DuLieu()
36:        {
37:            if (thehien == null)
38:            {
39:                StreamReader sr = new StreamReader("config.txt");
40:                string line = sr.ReadLine();
41:                if (line == "0")
42:                    thehien = new KetNoiDuLieuFile();
43:                if (line == "1")
44:                    thehien = new KetNoiDuLieuTinh();
```

```
45:         }
46:     }
47: }
48: }
```

Xây dựng lớp kết nối dữ liệu từ tập tin:

```
1: using System;
2: using System.Collections.Generic;
3: using System.Text;
4: using System.IO;
5: namespace QuanLiHocPhan
6: {
7:     public class KetNoiDuLieuFile:KetNoiDuLieu
8:     {
9:         public override List<SinhVien> DocSinhViens()
10:        {
11:            List<SinhVien> ds = new List<SinhVien>();
12:            StreamReader sr = new StreamReader("Sinhvien.txt");
13:            string line = "";
14:            while((line= sr.ReadLine())!=null)
15:            {
16:                string[] s = line.Split('*');
17:                ds.Add(new SinhVien(s[0].Trim(), s[1].Trim(), s[2].Trim()));
18:            }
19:            return ds;
20:        }
21:        public override bool XoaSinhViens(SinhVien s)
22:        {
23:            throw new Exception("The method or operation is not implemented.");
24:        }
25:        public override bool ThemSinhViens(SinhVien s)
26:        {
27:            throw new Exception("The method or operation is not implemented.");
28:        }
29:        public override bool SuaSinhViens(SinhVien s)
30:        {
31:            throw new Exception("The method or operation is not implemented.");
32:        }
33:        public override List<HocPhan> DocHocPhans()
34:        {
35:            List<HocPhan> ds = new List<HocPhan>();
36:            StreamReader sr = new StreamReader("Hocphan.txt");
37:            string line = "";
38:            while ((line = sr.ReadLine()) != null)
39:            {
```

```
40:         string[] s = line.Split('*');
41:         ds.Add(new HocPhan(s[0].Trim(),
        s[1].Trim(),int.Parse(s[2].Trim())));
42:     }
43:     return ds;
44: }
45:
46: public override bool XoaHocPhans(HocPhan s)
47: {
48:     throw new Exception("The method or operation is not implemented.");
49: }
50:
51: public override bool ThemHocPhans(HocPhan s)
52: {
53:     throw new Exception("The method or operation is not implemented.");
54: }
55:
56: public override bool SuaHocPhans(HocPhan s)
57: {
58:     throw new Exception("The method or operation is not implemented.");
59: }
60:
61: public override List<DangKiHP> DocDangKiHPs()
62: {
63:     List<DangKiHP> ds = new List<DangKiHP>();
64:     StreamReader sr = new StreamReader("DangKiHp.txt");
65:     string line = "";
66:     while ((line = sr.ReadLine()) != null)
67:     {
68:         string[] s = line.Split('*');
69:         ds.Add(new DangKiHP(s[0].Trim(), s[1].Trim()));
70:     }
71:     return ds;
72: }
73:
74: public override bool XoaDangKiHPs(DangKiHP s)
75: {
76:     throw new Exception("The method or operation is not implemented.");
77: }
78:
79: public override bool ThemDangKiHPs(DangKiHP s)
80: {
81:     throw new Exception("The method or operation is not implemented.");
82: }
83:
84: public override bool SuaDangKiHPs(DangKiHP s)
```

```
85:         {
86:             throw new Exception("The method or operation is not implemented.");
87:         }
88:     }
89: }
```

Xây dựng lớp kết nối dữ liệu thủ công:

```
1: using System;
2: using System.Collections.Generic;
3: using System.Text;
4:
5: namespace QuanLiHocPhan
6: {
7:     class KetNoiDuLieuTinh:KetNoiDuLieu
8:     {
9:
10:         public override List<SinhVien> DocSinhViens()
11:         {
12:             List<SinhVien> ds= new List<SinhVien>();
13:             ds.Add(new SinhVien("001", "A", "CTK32"));
14:             ds.Add(new SinhVien("002", "B", "CTK31"));
15:             ds.Add(new SinhVien("004", "C", "CTK31"));
16:             return ds;
17:         }
18:
19:         public override bool XoaSinhViens(SinhVien s)
20:         {
21:             return true;
22:         }
23:
24:         public override bool ThemSinhViens(SinhVien s)
25:         {
26:             return true;
27:         }
28:
29:         public override bool SuaSinhViens(SinhVien s)
30:         {
31:             return true;
32:         }
33:
34:         public override List<HocPhan> DocHocPhans()
35:         {
36:             throw new Exception("The method or operation is not implemented.");
37:         }
38:     }
```

```
39:         public override bool XoaHocPhans(HocPhan s)
40:         {
41:             throw new Exception("The method or operation is not implemented.");
42:         }
43:
44:         public override bool ThemHocPhans(HocPhan s)
45:         {
46:             throw new Exception("The method or operation is not implemented.");
47:         }
48:
49:         public override bool SuaHocPhans(HocPhan s)
50:         {
51:             throw new Exception("The method or operation is not implemented.");
52:         }
53:
54:         public override List<DangKiHP> DocDangKiHPs()
55:         {
56:             throw new Exception("The method or operation is not implemented.");
57:         }
58:
59:         public override bool XoaDangKiHPs(DangKiHP s)
60:         {
61:             throw new Exception("The method or operation is not implemented.");
62:         }
63:
64:         public override bool ThemDangKiHPs(DangKiHP s)
65:         {
66:             throw new Exception("The method or operation is not implemented.");
67:         }
68:
69:         public override bool SuaDangKiHPs(DangKiHP s)
70:         {
71:             throw new Exception("The method or operation is not implemented.");
72:         }
73:     }
74: }
```

Xây dựng lớp thư viện để đọc nội dung tập tin văn bản bất kì:

```
1: using System;
2: using System.Collections.Generic;
3: using System.Text;
4: using System.IO;
5: namespace QuanLiHocPhan
6: {
7:     public class ThuVien
```

```
8:      {
9:          public static string GetFromFile(string path)
10:         {
11:             StreamReader sr = new StreamReader(path);
12:             string kq = "";
13:             string line = "";
14:             while ((line = sr.ReadLine()) != null)
15:             {
16:                 kq += line;
17:                 kq += "\r\n";
18:             }
19:             return kq;
20:         }
21:     }
22: }
23: }
```

Biên dịch dự án trên thành tập tin dll và tạo mới một dự án quản lý học phần để sử dụng thư viện này:

Nội dung tập tin SinhVien.txt như sau:

```
001 * Nguyen Van A * CTK32
002 * Nguyen Van B * CTK31
004 * Nguyen Van C * CTK31
005 * Nguyen Van D * CTK30
006 * Nguyen Van E * CTK30
```

Nội dung tập tin hocphan.txt như sau:

```
001 * Lap trinh huong doi tuong* 4
002 * Cau truc du lieu * 4
003 * Mang May Tinh * 2
004 * Co so du lieu * 3
```

Nội dung tập tin dangkihp.txt như sau:

```
001 * 001
001 * 002
002 * 001
004 * 001
001 * 003
001 * 004
002 * 003
004 * 004
```

Nội dung tập tin config.txt như sau:

```
0
```

Nội dung tập tin `template.txt` như sau:

```
*****
{2}
* -----
*          KET QUA DANG KI HOC PHAN
*   Ma sinh vien:{0}
*   Ten sinh vien:{1}
*   Danh sach dang ki
* -----
*          Tong so so tin chi dang ki: {3}
*****
```

Nội dung chương trình chính:

```
1: using System;
2: using System.Collections.Generic;
3: using System.Text;
4:
5: namespace QuanLiHocPhan
6: {
7:     class Program
8:     {
9:         static void Main(string[] args)
10:        {
11:            HocPhan.Xoa += new EventHandler(HocPhan_Xoa);
12:            Console.WriteLine("Danh sach dang ki hoc phan");
13:            DangKiHP.Xuat();
14:            Console.WriteLine("Danh sach dang ki hoc phan sau
15:                               khi xoa 001");
16:            HocPhan.XoaHocPhanTheoMa("001");
17:            DangKiHP.Xuat();
18:            Console.ReadKey();
19:        }
20:        static void HocPhan_Xoa(object sender, EventArgs e)
21:        {
22:            DangKiHP.XoaHocPhanDangKiTheoMaHp((string)sender);
23:        }
24:    }
25: }
```

Kết quả thực hiện chương trình:

Danh sach dang ki hoc phan

001 | Nguyen Van A | CTK32 001 | Lap trinh huong doi tuong | 4

001	Nguyen Van A	CTK32	002	Cau truc du lieu	4
002	Nguyen Van B	CTK31	001	Lap trinh huong doi tuong	4
004	Nguyen Van C	CTK31	001	Lap trinh huong doi tuong	4
001	Nguyen Van A	CTK32	003	Mang May Tinh	2
001	Nguyen Van A	CTK32	004	Co so du lieu	3
002	Nguyen Van B	CTK31	003	Mang May Tinh	2
004	Nguyen Van C	CTK31	004	Co so du lieu	3
Danh sach dang ki hoc phan sau khi xoa 001					
001	Nguyen Van A	CTK32	002	Cau truc du lieu	4
001	Nguyen Van A	CTK32	003	Mang May Tinh	2
001	Nguyen Van A	CTK32	004	Co so du lieu	3
002	Nguyen Van B	CTK31	003	Mang May Tinh	2
004	Nguyen Van C	CTK31	004	Co so du lieu	3

Tài liệu tham khảo

- 1) Bài giảng lập trình hướng đối tượng.
- 2) Trang chiếu về lập trình C# của Microsoft.
- 3) Professional C#, 3rd Edition. NXB: Wrox, 2005.
- 4) C# How to Program, NXB: Prentice Hall, 2003.
- 5) Tài nguyên học tập tại <http://192.168.6.1/>