

ĐẠI HỌC DUY TÂN



• Hệ đào tạo:

- Thạc sĩ (Khoa học máy tính, Quản trị kinh doanh, Tài chính - Ngân hàng, Kế toán)
- Cử nhân đại học
- Cử nhân cao đẳng
- Kỹ sư
- Cao đẳng nghề (Công nghệ thông tin, Kế toán doanh nghiệp, Quản trị khách sạn)

• Ngành nghề đào tạo:

- Điện tử - Viễn thông
- Công nghệ thông tin
- Quản trị kinh doanh
- Hệ thống thông tin quản lý
- Tài chính - ngân hàng
- Quản trị dịch vụ du lịch & lữ hành
- Ngôn ngữ Anh
- Kế toán - Kiểm toán
- Văn học
- Kiến trúc
- Công nghệ kỹ thuật môi trường
- Điều dưỡng (Y tế)
- Xây dựng
- Đồ họa máy tính và multimedia
- Dược học
- Quan hệ quốc tế
- Việt nam học



Giá: 85.000đ

TS. LÊ TRUNG HIẾU
ThS. NGUYỄN THỊ MINH THI

LẬP TRÌNH WINDOWS FORM VỚI C#.NET

GIÁO TRÌNH

NHÀ XUẤT BẢN
THÔNG TIN VÀ TRUYỀN THÔNG



ĐẠI HỌC DUY TÂN
TS. LÊ TRUNG HIẾU
ThS. NGUYỄN THỊ MINH THI

GIÁO TRÌNH
LẬP TRÌNH WINDOWS FORM
C#.Net
với **TẬP 1**



NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

ĐẠI HỌC DUY TÂN
TS. LÊ TRUNG HIẾU
ThS. NGUYỄN THỊ MINH THI

GIÁO TRÌNH
LẬP TRÌNH WINDOWS FORM
với **C#.Net**

TẬP 1

NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

MÃ SỐ:

LỜI NÓI ĐẦU

.NET là nền tảng lập trình ứng dụng tiện lợi, cung cấp cho người lập trình môi trường làm việc trực quan, dễ dàng trong việc phát triển các ứng dụng, thuận lợi trong việc kết nối và làm việc với cơ sở dữ liệu. Winform là một kiểu ứng dụng được xây dựng trên môi trường .NET, được các lập trình viên và các nhà nghiên cứu ứng dụng rộng rãi vào công việc. Hiện nay Lập trình Windows Form với C#.NET là một trong những học phần quan trọng đối với các bạn sinh viên chuyên ngành Công nghệ thông tin trong các trường đại học và cao đẳng.

Với mục đích cung cấp cho các bạn sinh viên những kiến thức khá toàn diện về lập trình ứng dụng Windows Form với C# 5.0, phiên bản mới nhất và nền tảng của C# về lập trình cơ sở, lập trình hướng đối tượng, cần thiết cho lập trình ứng dụng Windows Form C#; nhóm tác giả giảng viên, Trường Đại học Duy Tân phối hợp với Nhà xuất bản Thông tin và Truyền thông xuất bản cuốn giáo trình "*Lập trình Windows Form với C#.NET – Tập 1*".

Nội dung giáo trình gồm 3 chương, cụ thể như sau:

Chương 1. Microsoft .Net và C#

Chương 2. Cơ bản về C#

Chương 3. Lập trình hướng đối tượng với C#

Cuốn giáo trình sẽ mang lại cho các bạn sinh viên, những người yêu thích lập trình những kiến thức nền tảng về dịch vụ ADO.NET truy cập nhiều nguồn dữ liệu thông dụng hiện nay

như Microsoft SQL, Microsoft Access, Oracle, MySQL, DB2 và XML; Kiến thức về dự án Excel Workbook mở rộng chức năng cho bảng tính Microsoft Excel sử dụng dịch vụ Excel Interop. Ngoài ra, giáo trình còn cung cấp kiến thức lập trình song song sử dụng tuyến đoạn, lập trình ứng dụng sử dụng mô hình đối tượng thành phần COM và thành phần phân tán mới nhất của Microsoft như Web service, WCF.

Mặc dù đã cố gắng trong công tác biên soạn, tham khảo nhiều tài liệu và diễn đạt trong sáng để giúp người đọc dễ tiếp cận hơn, nhưng giáo trình được xuất bản lần đầu sẽ khó tránh khỏi thiếu sót. Vì vậy rất mong nhận được sự đóng góp ý kiến của bạn đọc để giáo trình hoàn thiện hơn trong lần xuất bản tiếp theo. Mọi ý kiến góp ý xin gửi về địa chỉ Email: thimtnguyen2005@yahoo.com.

Xin trân trọng giới thiệu cùng bạn đọc./.

Đà Nẵng, tháng 8 năm 2012

NHÓM BIÊN SOẠN

MỤC LỤC

<i>Lời nói đầu</i>	3
Chương 1: MICROSOFT .NET VÀ C#	9
1.1. Tổng quan về Microsoft .Net	9
1.1.1. Lịch sử phát triển của Microsoft .NET	9
1.1.2. Đặc điểm và kiến trúc .NET framework.....	10
1.2. Ngôn ngữ c#	13
1.2.1. Giới thiệu C#	13
1.2.2. Đặc trưng của ngôn ngữ C#.....	14
1.2.3. Biên dịch và thực hiện ứng dụng Console C#.NET đơn giản	18
1.3. Câu hỏi chương 1	21
Chương 2: CƠ BẢN VỀ C#	23
2.1. Các thành phần cơ bản của ngôn ngữ C#	24
2.1.1. Bộ ký tự dùng trong C# (Character set)	24
2.1.2. Từ khoá (Keyword)	24
2.1.3. Định danh (Identifier)	25
2.1.3. Lời chú thích (Comment)	25
2.2. Kiểu dữ liệu, biến, hằng	33
2.2.1. Kiểu dữ liệu (Data type)	33
2.2.2. Biến (Variable)	62
2.2.3. Hằng (Constant và literal).....	63
2.3. Biểu thức và toán tử	64
2.3.1. Biểu thức (Expression)	64
2.3.2. Toán tử (Operator).....	65
2.3.3. Các quy tắc thực hiện phép toán và chuyển kiểu	71

2.4. Lệnh (Statement)	73
2.4.1. Lệnh đơn (Simple statement).....	73
2.4.2. Khối lệnh (Compound statement hay block).....	74
2.4.3. Cấu trúc rẽ nhánh có điều kiện (Conditional structure)	74
2.4.4. Cấu trúc lặp (Repeat structure hay loop)	79
2.4.5. Các lệnh điều khiển rẽ nhánh không điều kiện.....	85
2.5. Ngoại lệ (Exception) và xử lý ngoại lệ	85
2.5.1. Lệnh try...catch...finally.....	86
2.5.2. Lệnh throw	88
2.5.3. Các lớp ngoại lệ	89
2.6. Lớp System.Console.....	92
2.6.1. Định dạng kết xuất.....	92
2.6.2. Nhập và xuất với lớp Console	93
2.6.3. Định dạng và thiết lập vị trí kết xuất cho phương thức Write.....	94
Câu hỏi chương 2	97
Bài tập chương 2	98
Chương 3: LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG C#.....	101
3.1. Lớp và đối tượng	104
3.1.1. Xây dựng lớp	104
3.1.2. Tạo đối tượng.....	106
3.1.3. Truy xuất thành viên của lớp	107
3.2. Phương thức	111
3.2.1. Định nghĩa phương thức (Method definition)	111
3.2.2. Phạm vi truy xuất thành phần của lớp	116
3.2.3. Phương thức Main().....	118
3.2.4. Phương thức khởi tạo (Constructor)	118

3.2.5. Phương thức hủy (Destructor)	122
3.2.6. Từ khoá this	123
3.2.7. Nạp chồng phương thức (Overloading method).....	126
3.2.8. Truyền tham đối cho phương thức	127
3.2.9. Nạp chồng toán tử (Overloading operator).....	132
3.3. Thuộc tính (Properties)	135
3.4. Tham chiếu phương thức (Delegate)	138
3.5. Sự kiện (Event)	142
3.6. Chỉ mục (Indexer)	145
3.7. Kiểu cấu trúc (Struct)	147
3.8. Kiểu tổng quát (Generic type)	149
3.8.1. Lớp (generic class), giao tiếp (generic interface) và cấu trúc tổng quát (generic struct)	149
3.8.2. Phương thức tổng quát (generic method)	150
3.9. Cây biểu thức (Expression Tree)	151
3.9.1. Tạo cây biểu thức từ biểu thức lambda	151
3.9.2. Tạo cây biểu thức sử dụng API (application programming interface).....	151
3.10. Kế thừa lớp (Classical Inheritance).....	153
3.10.1. Định nghĩa lớp kế thừa	153
3.10.2. Viết chồng phương thức (Overriding method) hay che khuất phương thức (Hiding method).....	154
3.10.3. Từ khóa base	157
3.11. Không gian tên (NameSpace) và câu lệnh Using	159
3.11.1. Khái niệm namespace	159
3.11.2. Định nghĩa namespace	159
3.11.3. Sử dụng namespace	161
3.11.4. Lệnh using	161

3.12. Lớp, phương thức trừu tượng (Abstract class, method).....	162
3.13. Lớp phương thức hằng (Sealed class, sealed method) ...	164
3.14. Giao tiếp (Interface)	165
3.14.1. Khai báo giao tiếp.....	165
3.14.2. Hiện thực (cài đặt) giao tiếp	165
Câu hỏi chương 3	168
Bài tập chương 3	169
Tài liệu tham khảo	176

Chương 1

MICROSOFT .NET VÀ C#

Chương này giới thiệu tổng quan về Microsoft .NET, ngôn ngữ C#, ý nghĩa và sức mạnh của Microsoft .NET và ưu điểm của ngôn ngữ C# so với các ngôn ngữ khác. Ngoài ra chương này còn trình bày về cách thức bắt đầu lập trình với C# trong môi trường Visual Studio .NET như thế nào và cách thức biên dịch và thực hiện ứng dụng Console C# đơn giản.

Tổng quan về Microsoft .NET

- Lịch sử phát triển của Microsoft .NET
- Đặc điểm và kiến trúc .NET framework: Common language runtime và Thư viện .NET framework

Microsoft Visual C#

- Giới thiệu ngôn ngữ C#
- Các đặc trưng của C#
- Biên dịch và thực hiện một chương trình C# đơn giản

1.1. TỔNG QUAN VỀ MICROSOFT .NET

1.1.1. Lịch sử phát triển của Microsoft .NET

Vào đầu năm 1998, sau khi hoàn thành phiên bản Version 4 của IIS (Internet Information Server), công ty Microsoft bắt đầu xây dựng một kiến trúc mới trên nền tảng IIS và đặt tên là NGWS (Next Generation Windows Services).

Sau khi Visual Basic ra đời vào cuối 1998, dự án kế tiếp là Visual Studio 7 được xác nhập vào NGWS. Đến tháng 11/2000 thì Microsoft đã phát hành phiên bản Beta của .NET 1.0.

Tháng 2/2002, phiên bản 1.0 của .NET framework ra đời tích hợp cùng với bộ Visual Studio .NET, được mặc định cài đặt trong Windows XP, tiếp theo phiên bản 1.1 phát hành với Visual Studio .NET 2003 và Windows Server 2003.

Sau đó, phiên bản 2.0 của .NET framework tích hợp cùng với bộ Visual Studio .NET 2005, và Windows Server 2003.

Phiên bản 3.0 của .NET framework bao gồm với Visual Studio.NET 2008 và Windows Server 2008, Windows Vista, tiếp theo, phiên bản 3.5 với Windows 7 và Windows Server 2008.

Ngày 12 tháng 4 năm 2010, NET framework 4.0 được phát hành cùng với Visual Studio 2010.

Gia đình .NET framework cũng bao gồm hai phiên bản sử dụng cho điện thoại di động và hệ thống nhúng. Một phiên bản .NET compact framework, bao gồm trong Windows CE, một hệ điều hành nguồn mở 32 bit, sử dụng cho các thiết bị thông minh như các vi điều khiển dùng trong công nghiệp, các thiết bị đầu cuối như camera, điện thoại di động, các thiết bị giải trí gia đình. Phiên bản thứ hai là .NET micro framework nhằm phát triển các hệ thống nhúng trên các thiết bị nhỏ, hạn chế tài nguyên.

1.1.2. Đặc điểm và kiến trúc .NET framework

Microsoft .NET là nền tảng cho việc xây dựng và thực thi các ứng dụng từ các ứng dụng truyền thống giao diện người dùng ký tự CUI (Character User Interface), các ứng dụng giao diện người dùng đồ họa GUI (Graphics User Interface) Windows Form đến ứng dụng web, ứng dụng thiết bị di động và hệ thống nhúng, ứng dụng mạng và ứng dụng hướng dịch vụ trong môi trường phân tán Internet.

.NET framework là một nền tảng mới làm đơn giản việc phát triển ứng dụng trong môi trường phân tán của Internet với các đặc điểm:

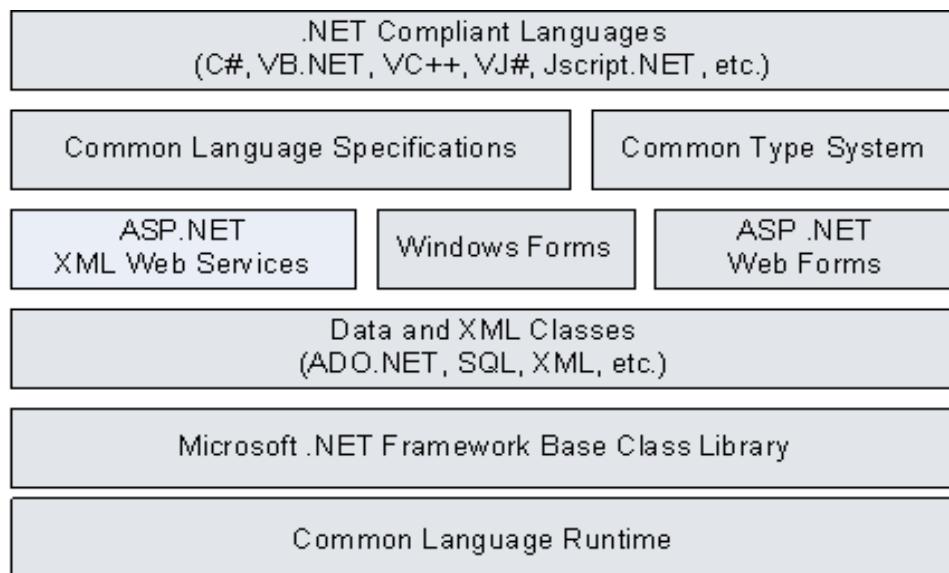
- Cung cấp môi trường lập trình hướng đối tượng thuận tủy và mạnh mẽ.
- Quản lý cài đặt phần mềm đảm bảo không tranh chấp phiên bản, thực thi an toàn mã nguồn
- Cung cấp mô hình bảo mật chung cho các ứng dụng
- .NET độc lập ngôn ngữ nghĩa là bạn có thể sử dụng nhiều ngôn ngữ lập trình được hỗ trợ bởi .NET để xây dựng và tích hợp các ứng dụng .NET. .NET framework cung cấp hệ thống kiểu chung CTS (Common Type System), định nghĩa các kiểu dữ liệu và các cấu trúc hỗ trợ bởi môi trường thực hiện và quản lý mã nguồn CLR (Common Language Runtime), và làm thế nào chúng có thể tương tác với nhau tuân theo đặc tả hạ tầng ngôn ngữ chung CLI (Common Language Infrastructure). Vì vậy, .NET hỗ trợ trao đổi các kiểu giữa các thư viện và ứng dụng sử dụng bất kỳ ngôn ngữ .NET. Có đến hơn hai mươi ngôn ngữ lập trình hiện nay được hỗ trợ trên nền tảng .NET như là Fortran, Pascal, Cobol, Visual Basic, C#, C++, Java, JScript, Python, Eiffel, Perl, Small Talk, Scheme, APL, Mercury... Trong đó, các ngôn ngữ lập trình phổ biến nhất để phát triển các ứng dụng trên nền tảng .NET hiện nay là C# và VB.NET.

.NET framework có hai thành phần chính:

- CLR (Common Language Runtime): là nền tảng của .NET framework, môi trường thực hiện và quản lý mã nguồn mà được biên dịch thành một ngôn ngữ trung gian của Microsoft gọi là MSIL (Microsoft Intermediate Language) và lưu trữ trong tập tin gọi là assembly. Ngôn ngữ trung gian MSIL này là ngôn ngữ chung cho tất cả các ngôn ngữ .NET hiện có. Trong khi biên dịch như vậy, các ứng dụng cũng tạo ra

những thông tin cần thiết để giới thiệu ứng dụng, ta gọi những thông tin này là metadata. CLR sử dụng trình biên dịch JIT (Just-In-Time) để biên dịch mã IL một lần nữa thành ngôn ngữ gốc của máy tính (mã nhị phân) trước khi thực hiện. Các đặc điểm chính khác của CLR là quản lý phiên bản, quản lý bộ nhớ, tích hợp hệ thống độc lập ngôn ngữ và cung cấp hệ thống kiểu dữ liệu chung.

- .NET framework class library: thư viện lớp .NET, là tập hợp hướng đối tượng của các kiểu dữ liệu (lớp, giao tiếp, kiểu liệt kê...) tái sử dụng. Các kiểu dữ liệu đó được tổ chức lại thành từng nhóm riêng biệt như trong một thư viện để ta dễ dàng sử dụng. Ta gọi các nhóm như vậy là không gian tên (namespaces), và ta sẽ dùng những namespace này để gọi hay nhập (import) các kiểu dữ liệu cần thiết cho ứng dụng của mình.



Hình 1.1. Kiến trúc .NET framework

1.2. NGÔN NGỮ C#

1.2.1. Giới thiệu C#

Ngôn ngữ C# (C sharp) do đội ngũ kỹ sư của Microsoft thiết kế, trong đó người dẫn đầu là Anders Hejlsberg và Scott Wiltamuth. Anders Hejlsberg là tác giả của Turbo Pascal và là người lãnh đạo nhóm thiết kế Borland Delphi. Visual C# là một cài đặt của ngôn ngữ C# bởi Microsoft.

C# là một ngôn ngữ lập trình hướng đối tượng đầy năng lực, được dẫn xuất từ Java và C++, và thêm vào những đặc tính mới để làm cho ngôn ngữ này dễ sử dụng hơn. Nhiều trong số những đặc tính này khá giống với những đặc tính có trong ngôn ngữ Java. Nếu bạn đã sử dụng ngôn ngữ C++ hay Java, bạn sẽ thấy rằng cú pháp của C# rõ ràng tương thích.

C# đóng vai trò quan trọng trong kiến trúc của Microsoft .NET framework, và nhiều người đã so sánh với vai trò của C trong việc phát triển UNIX.

Ngôn ngữ C# được thiết kế cho việc xây dựng các ứng dụng dựa trên nền tảng .NET như ứng dụng Windows Form, ứng dụng web truy xuất cơ sở dữ liệu sử dụng công nghệ ADO.NET (ActiveX Data Objects for .NET) hay LINQ (Language Integrated Query), ứng dụng mạng, ứng dụng phân tán và hướng dịch vụ sử dụng các dịch vụ COM+, .NET remoting, Web service, WCF (Windows Communication Foundation), ứng dụng xử lý song song đa tuyến đoạn (multi-threading), ứng dụng cho các thiết bị kỹ thuật số cá nhân PDA (Personal Digital Assistant) như pocket PC, SmartPhone, iPhone...

Lịch sử phát triển của C# và các đặc điểm chính của từng phiên bản mô tả trong bảng sau:

Bảng 1.1. Lịch sử phát triển của C#

Phiên bản	Ngày ra đời	Visual Studio	Đặc điểm mới
C# 1.0	01/2002	2002	
C# 2.0	11/2005	2005	Kiểu tổng quát (generic type), phương thức nặc danh (anonymous method), kiểu Nullable (nullable type)
C# 3.0	11/2007	2008	Định nghĩa biến kiểu không tường minh (implicitly typed local variable), kiểu nặc danh (anonymous type), phương thức mở rộng (extension method), khởi tạo đối tượng và danh sách (object and collection initializer), thuộc tính được tự động cài đặt (auto-implemented properties), biểu thức lambda (lambda expression), cây biểu thức (expression tree), biểu thức truy vấn (query expression) LINQ (Language-Integrated Query)
C# 4.0	04/2010	2010	Tham đối đặt tên (named argument), tham đối tùy chọn (optional argument), cải tiến hơn giao diện thành phần COM (more COM interface), kiểu dynamic và ràng buộc trễ (dynamic type and late binding)
C# 5.0	02/2012	2012	Lập trình bắt đồng bộ (async programming), thông tin lời gọi phương thức (caller information)

1.2.2. Đặc trưng của ngôn ngữ C#

Mục tiêu của C# là cung cấp một ngôn ngữ hướng đối tượng, đơn giản, an toàn và hiện đại, với nhiều tính năng mạnh mẽ và mềm dẻo.

1.2.2.1. Đơn giản

Ngôn ngữ C# đơn giản, chỉ khoảng 80 từ khóa và hơn mươi mấy kiểu dữ liệu được xây dựng sẵn. C# khá giống về cú pháp, biểu thức, toán tử và những chức năng khác được lấy trực tiếp từ ngôn ngữ

Java và C++, nhưng nó đã được cải tiến để làm cho ngôn ngữ đơn giản hơn.

Cũng như Java, C# loại bỏ một vài sự phức tạp của C++, bao gồm đa kế thừa, và lớp cơ sở ảo (virtual base class), chúng là những nguyên nhân gây ra những vấn đề cho người phát triển C++.

C# chỉ sử dụng dấu chấm để truy xuất các thành viên của lớp như Java, thay vì sử dụng dấu ::, . và → như C++. Đối với người mới tìm hiểu về công nghệ này thì các cải tiến này làm bớt nhầm lẫn và đơn giản hơn.

C# cũng đưa ra khái niệm thuộc tính (property) đơn giản thay cho phương thức truy cập và thiết lập giá trị biến thành viên. Thuộc tính cung cấp khả năng bảo vệ các biến dữ liệu bên trong một lớp, bằng việc đọc và ghi chúng thông qua thuộc tính.

Các phương thức nhập xuất dữ liệu là thành viên của lớp Console giúp người mới học dễ dàng nhớ các thao tác nhập xuất dữ liệu.

Các danh sách (Collection) phong phú thuộc không gian tên System.Collections và System.Collections.Generic, các kiểu liệt kê (enum) xây dựng sẵn giúp người dùng thuận tiện hơn trong xử lý các danh sách dữ liệu.

1.2.2.2. An toàn và hiện đại

C# chứa tất cả những đặc tính như là xử lý ngoại lệ, thu gom bộ nhớ tự động, những kiểu dữ liệu mở rộng, bảo mật mã nguồn, kiểu tổng quát (generic type), hỗ trợ cho cấu trúc, thành phần (component), lập trình hướng đối tượng là những đặc tính được mong đợi trong một ngôn ngữ lập trình hiện đại.

Trong C#, bộ thu gom bộ nhớ tự động và kiểu dữ liệu an toàn được tích hợp vào ngôn ngữ, sẽ loại bỏ những vấn đề rắc rối của C++.

1.2.2.3. Hướng đối tượng

Những đặc điểm chính của ngôn ngữ lập trình hướng đối tượng (object-oriented programming language) là đóng gói (encapsulation), kế thừa (inheritance), và đa hình (polymorphism) được hỗ trợ trong C#.

1.2.2.4. Mạnh mẽ và cung mềm dẻo

C# được sử dụng cho nhiều ứng dụng từ client đến server trong môi trường phân tán internet như các ứng dụng Windows Form, ứng dụng web truy xuất cơ sở dữ liệu sử dụng công nghệ ADO.NET hay LINQ, ứng dụng mạng, ứng dụng phân tán và dịch vụ web sử dụng các dịch vụ COM+, .NET remoting, web service, WCF service, ứng dụng xử lý song song đa tuyến đoạn, ứng dụng cho các thiết bị kỹ thuật số cá nhân PDA như pocket PC, SmartPhone, IPhone...

C# là một trong những ngôn ngữ lập trình mới nhất và đang được sử dụng phổ biến. Nhiều sản phẩm của công ty Microsoft đã được chuyển đổi và viết lại bằng C#. Bằng cách sử dụng ngôn ngữ này, Microsoft đã xác nhận khả năng của C#.

1.2.2.5. Ngôn ngữ C# và những ngôn ngữ khác

Có nhiều ngôn ngữ lập trình phổ biến trước đây như Visual Basic, C++ và Java. Chúng ta hãy tìm hiểu sự khác nhau giữa ngôn ngữ C# và những ngôn ngữ này.

Nếu chúng ta đã học Java, chúng ta sẽ tìm thấy nhiều tương thích trong C#. Microsoft nói rằng C# mang đến sức mạnh của ngôn ngữ C++ với sự dễ dàng của ngôn ngữ Visual Basic.

C# loại bỏ một vài đặc tính phức tạp của C++ bao gồm đa kế thừa và lớp cơ sở ảo, chúng là những nguyên nhân gây ra những vấn đề cho người phát triển C++.

Trong ngôn ngữ C# mọi thứ liên quan đến khai báo lớp đều được tìm thấy trong phần khai báo của nó. Định nghĩa một lớp trong ngôn ngữ C# không yêu cầu phải chia ra tập tin header và tập tin nguồn giống như trong ngôn ngữ C++. Hơn thế nữa, ngôn ngữ C# hỗ trợ kiểu XML (eXtensible Markup Language), cho phép chèn các thẻ XML để phát sinh tự động các tài liệu cho lớp.

Như đã nói ở bên trên .NET runtime trong C# thực hiện việc thu gom bộ nhớ (garbage collection) tự động như Java. Do điều này nên việc sử dụng con trỏ trong C# ít quan trọng hơn trong C++. Nhưng con trỏ và phép toán con trỏ cũng có thể được sử dụng trong C#, khi đó những đoạn mã nguồn này sẽ được đánh dấu là không an toàn (unsafe code). Và bộ giải phóng bộ nhớ tự động của CLR sẽ không thực hiện việc giải phóng những đối tượng được tham chiếu bằng sử dụng con trỏ cho đến khi chúng được giải phóng.

Một điểm giống nhau giữa C# và Java là cả hai cùng biên dịch ra mã trung gian: C# biên dịch ra MSIL còn Java biên dịch ra bytecode. Sau đó chúng được thực hiện bằng cách thông dịch hoặc biên dịch trong từng máy ảo tương ứng. Tuy nhiên, trong ngôn ngữ C# nhiều hỗ trợ được đưa ra để biên dịch mã ngôn ngữ trung gian sang mã máy.

C# chứa nhiều kiểu dữ liệu cơ sở hơn Java và cũng cho phép nhiều mở rộng với kiểu dữ liệu giá trị. Ví dụ, ngôn ngữ C# hỗ trợ kiểu liệt kê (enumeration), kiểu này được giới hạn đến một tập hằng định nghĩa trước, và kiểu dữ liệu cấu trúc là kiểu dữ liệu giá trị do người dùng định nghĩa.

C# hỗ trợ ngoại lệ và xử lý ngoại lệ giống Java và C++. Khái niệm không gian tên (namespace) của C# tương tự C++ và khái niệm gói (package) trong Java.

Ngoài việc truyền tham đối phụ thuộc vào kiểu dữ liệu tham đối như Java và C++, C# còn hỗ trợ từ khóa ref, out, cho phép truyền dữ liệu kiểu giá trị bằng tham chiếu. C# cũng hỗ trợ kiểu tổng quát (generic type) cho lớp và phương thức như C++ và Java.

Trong C#, nạp chồng toán tử được hỗ trợ tương tự C++.

Những thành viên của lớp được gọi duy nhất bằng toán tử “.” như Java, và khác với C++ có nhiều cách gọi trong các tình huống khác nhau.

C# cũng hỗ trợ giao tiếp (interface). Trong ngôn ngữ C#, một lớp chỉ có thể kế thừa từ duy nhất một lớp cha, tức là không cho đa kế thừa như trong ngôn ngữ C++, tuy nhiên một lớp có thể cài đặt nhiều giao tiếp như Java và C++.

Trong ngôn ngữ C#, cấu trúc cũng được hỗ trợ, nhưng khái niệm và ngữ nghĩa của nó thay đổi khác với C++. Trong C#, một cấu trúc được giới hạn, là kiểu dữ liệu nhỏ gọn, và khi tạo thẻ hiện yêu cầu ít hơn về hệ điều hành và bộ nhớ so với một lớp. Một cấu trúc thì không thể kế thừa từ một lớp hay được kế thừa nhưng một cấu trúc có thể hiện thực một giao tiếp.

Và nhiều đặc tính mới mạnh mẽ khác được bổ sung trong C# như các khái niệm lệnh lặp foreach, property, event, delegate, indexer...

1.2.3. Biên dịch và thực hiện ứng dụng Console C#.NET đơn giản

Để bắt đầu cho việc tìm hiểu ngôn ngữ C#, chương đầu tiên trình bày cách soạn thảo, biên dịch và thực thi một chương trình C# đơn giản nhất.

Ví dụ: Viết chương trình hiển thị ra màn hình dòng chữ “Welcome to C#.NET”

```
using System;
namespace Hello
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Welcome to C#.NET");
        }
    }
}
```

Đây là một định nghĩa lớp Hello gồm có một phương thức tên là Main(). Tất cả các chương trình C# phải chứa một lớp có phương thức là Main() là phương thức được gọi thực hiện đầu tiên mỗi khi thực hiện chương trình. Phương thức Console.WriteLine() sử dụng để xuất một chuỗi trên màn hình console và kết thúc dòng.

C# là một ngôn ngữ phân biệt hoa-thường. Bạn cần phải chú ý là các từ khóa trong C# đều được viết thường ví dụ public, class, static... trong khi các tên namespace, tên lớp, phương thức... sẽ được viết hoa đầu từ, ví dụ System, Console.WriteLine, Main...

Có hai cách để soạn thảo, biên dịch và thực thi chương trình:

1.2.3.1. Sử dụng trình biên dịch dòng lệnh C#

Sử dụng chương trình soạn thảo văn bản bất kỳ như Notepad soạn thảo mã nguồn chương trình, rồi lưu vào tập tin có phần mở rộng là *.cs, trong ví dụ này là Hello.cs.

Bước tiếp theo là biên dịch tập tin nguồn vừa tạo ra, sử dụng trình biên dịch dòng lệnh C# (C# command-line compiler) csc.exe :

Chọn Start/ Programs/ Microsoft Visual Studio/ Visual Studio Tools/ Visual Studio Command Prompt.

Chuyển đến thư mục chứa tập tin nguồn, rồi gõ lệnh sau:

csc.exe [/out: TậpTinThựcThi] TậpTinNguồn

Ví dụ: csc /out:Hello.exe Hello.cs

hay csc Hello.cs

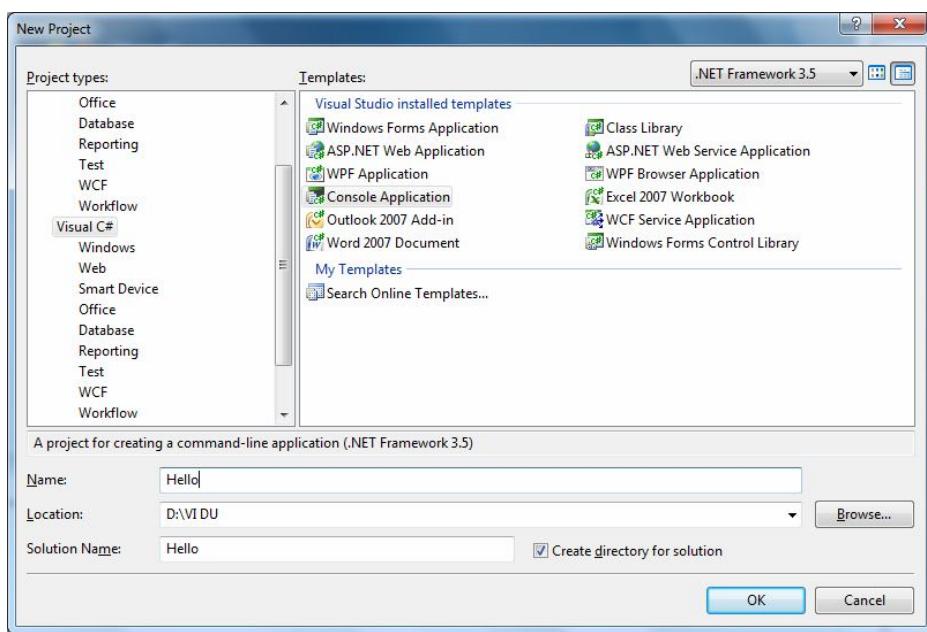
Kết quả tập tin Hello.exe sẽ xuất hiện trong cùng thư mục chứa tập tin nguồn.

Có thể sử dụng tùy chọn /out, theo sau là tên của tập tin chương trình thực thi hay chính là kết quả biên dịch tập tin nguồn.

Các tham số tùy chọn có rất nhiều nếu muốn tìm hiểu chúng ta có thể dùng lệnh: csc.exe /? Lệnh này hiển thị toàn bộ các tùy chọn biên dịch và các hướng dẫn sử dụng.

Cuối cùng, thực hiện tập tin Hello.exe bằng cách gõ: Hello

1.2.3.2. Sử dụng môi trường phát triển tích hợp IDE (Integrated Development Environment) Visual Studio .NET



Hình 1.2. Hộp thoại New project

Để tạo dự án Hello.cs trong Microsoft Visual Studio, khởi động Visual Studio, chọn menu File/ New/ Project. Chức năng này sẽ mở cửa sổ New Project. Chọn mục Visual C# trong vùng Project types bên trái, ở vùng Templates bên phải, chọn Console Application. Lúc này chúng ta có thể nhập tên cho ứng dụng ở mục Name, lựa chọn thư mục lưu trữ các tập tin này ở mục Location, và nhập tên solution chứa dự án ở mục Solution Name. Visual Studio .NET sẽ tạo ra một solution Hello chứa project Hello. Một không gian tên (namespace) phát sinh dựa trên tên của project Hello để chứa project. Một lớp tên là Program.cs phát sinh, có thể tùy ý đổi tên của chúng trong cửa sổ Solution Explorer. Khi đổi tên tập tin chứa lớp là Hello.cs, tên lớp cũng thay đổi thành Hello.

Để biên dịch chương trình, chọn menu Build/ Build Solution.

Để chạy chương trình có hay không sử dụng chế độ debug, chọn Debug/ Start Debugging hay Start Without Debugging.

Sau khi biên dịch và chạy chương trình, kết quả dòng chữ “Welcome to C#.NET” hiển thị ra màn hình.

Chúng ta cần tìm hiểu các khái niệm solution, project và assembly.

Dự án (project) sử dụng để quản lý, xây dựng, biên dịch và thực hiện hiệu quả các thành viên cần thiết để tạo nên ứng dụng như các tham chiếu (references), các kết nối cơ sở dữ liệu (data connections), các thư mục (folders) và các tập tin (files). Tập tin dự án sau khi biên dịch là một tập tin khả thi .exe (executable file) hay một thư viện liên kết động .dll (dynamic link library).

Solution có thể chứa một hay nhiều project. Visual Studio .NET lưu trữ định nghĩa solution trong hai tập tin: .sln và .suo. Tập tin .sln lưu trữ dữ liệu định nghĩa solution như các thành viên và cấu hình ở cấp solution. Tập tin .suo lưu trữ dữ liệu thiết lập tùy chọn IDE. Để một dự án trong solution trở thành dự án khởi động, kích phải tên dự án trong cửa sổ Solution Explorer, chọn Set as StartUp Project.

Một cửa sổ giao diện cho việc xem và quản lý các solution, project và các thành viên của project là Solution Explorer, được cung cấp bởi Visual Studio .NET bằng cách chọn View/ Solution Explorer.

Assembly là thành phần cơ bản nhất của bất kỳ ứng dụng .NET. Khi biên dịch một ứng dụng .NET, Visual Studio sẽ tạo ra một assembly được lưu trữ dạng tập tin khả thi .exe hay tập tin thư viện liên kết động .dll. Assembly chứa mã biên dịch MSIL và các thành phần khác như tập các kiểu và tài nguyên, với mục đích bảo mật, định danh kiểu, chia sẻ, phiên bản.

CÂU HỎI CHƯƠNG 1

Câu 1. Hai thành phần chính của .NET framework là gì?

Câu 2. CLR là gì?

Câu 3. JIT thực hiện công việc gì?

Câu 4. MSIL được tạo ra khi nào?

Câu 5. C# độc lập phần cứng hệ điều hành không?

Câu 6. Kể tên một số khác nhau cơ bản giữa C# với C++, Java

Câu 7. Tại sao ngôn ngữ C# là một ngôn ngữ lập trình tốt?

Câu 8. Tập tin chương trình mã nguồn C# có phần mở rộng gì?

Câu 9. Khái niệm solution và project?

Câu 10. Thế nào là một assembly?

Câu 11. Soạn thảo và biên dịch, thực thi chương trình xuất ra màn hình một bài thơ nào đó theo hai cách:

- Soạn thảo bằng trình soạn thảo văn bản Notepad, biên dịch bởi trình biên dịch dòng lệnh C# và thực thi chương trình.
- Soạn thảo và biên dịch, thực thi chương trình trong môi trường Visual Studio .NET.

Chương 2

CƠ BẢN VỀ C#

Chương này trình bày cơ bản về C# và so sánh ngôn ngữ C# với Java và C++. Bạn đọc sẽ khám phá hệ thống kiểu dữ liệu, phân biệt giữa kiểu dữ liệu giá trị và tham chiếu, kiểu dữ liệu định nghĩa sẵn và kiểu dữ liệu do người dùng định nghĩa. Bên cạnh đó bạn đọc sẽ tìm hiểu cách khai báo biến, cách sử dụng các phép toán, câu lệnh điều kiện, lệnh lặp. Cuối cùng, bạn đọc sẽ tìm hiểu C# sử dụng ngoại lệ để xử lý lỗi dễ dàng như thế nào. Tất cả những kiến thức cơ bản này là nền tảng cho lập trình ứng dụng Windows Form.

Các thành phần cơ bản của ngôn ngữ C#

Kiểu dữ liệu

- Kiểu dữ liệu giá trị (Value data type)
 - Kiểu dữ liệu cơ sở (Primitive data type)
 - Kiểu cấu trúc (Struct)
 - Kiểu liệt kê (Enumeration)
 - Kiểu Nullable
- Kiểu dữ liệu tham chiếu (Reference data type)
 - Chuỗi (String)
 - Kiểu con trỏ (Pointer)
 - Mảng (Array)
 - Kiểu tham chiếu phương thức (Delegate)
 - Lớp (Class)

- Các lớp danh sách (Collection) xây dựng sẵn
- Biến (Variable) và hằng (constant và literal)
- Phép toán (Operator) và các quy tắc trên các phép toán
- Cấu trúc điều khiển (Control structures)
- Ngoại lệ (Exception) và xử lý ngoại lệ
- Nhập/ xuất dữ liệu từ bàn phím

2.1. CÁC THÀNH PHẦN CƠ BẢN CỦA NGÔN NGỮ C#

2.1.1. Bộ ký tự dùng trong C# (Character set)

Ngôn ngữ C# được xây dựng trên bộ ký tự gồm 26 chữ cái hoa A...Z và 26 chữ cái thường a...z, 10 chữ số 0...9, các ký hiệu toán học + - * / % = ()..., dấu nối _, các ký hiệu đặc biệt khác ; : {} [] ? \ & | # \$...

C# sử dụng bộ ký tự chuẩn quốc tế Unicode. Khác với ký tự ASCII dài 8 bit, ký tự Unicode dài 16 bit hay 32 bit. Nó không chỉ bao gồm những ký tự trong bộ ký tự ASCII mà còn có vài triệu ký tự khác tương ứng với hầu hết các bảng chữ cái trên thế giới.

2.1.2. Từ khoá (Keyword)

Là những từ định nghĩa trước trong C#, có ý nghĩa xác định, phải dùng đúng cú pháp, đều viết bằng chữ thường, không dùng vào việc khác hay đặt tên mới trùng từ khoá.

Từ khoá gồm có từ khoá khai báo, điều khiển, kiểu dữ liệu, toán tử:

- Từ khoá khai báo: namespace, public, private, static, const, class, new...
- Từ khoá điều khiển: switch, case, break, if, return, for, while, continue, try, catch...
- Từ khoá toán tử: is
- Từ khoá kiểu dữ liệu: bool, byte, sbyte, short, ushort, float, double, null, void...

2.1.3. Định danh (Identifier)

Định danh là tên được đặt cho kiểu dữ liệu, phương thức, biến, hằng, đối tượng, không gian tên... Định danh là một dãy ký tự có phân biệt chữ hoa, thường. Tên bắt đầu bằng ký tự chữ cái hoặc dấu gạch nối dưới _, các ký tự còn lại phải là ký tự chữ cái, chữ số, dấu gạch nối dưới. Các định danh không được đặt tên trùng với từ khóa của C#.

Khi đặt tên nên theo quy tắc đặt tên định nghĩa sẵn của C# cho dễ nhớ là tất cả các tên của lớp, phương thức, biến, giao tiếp, không gian tên... đều viết chữ hoa đầu từ, ngoại trừ tên hằng viết chữ hoa.

2.1.4. Lời chú thích (Comment)

Lời chú thích thêm vào chương trình với mục đích giải thích, giúp cho người lập trình dễ dàng bổ sung, sửa chữa, nâng cấp chương trình. Khi chạy chương trình, trình dịch sẽ bỏ qua chú thích.

Các lời chú thích có thể được thêm vào mã nguồn C# giống như trong Java, C++ và có thể đặt tại bất kỳ vị trí nào:

- Chú thích gồm một hay nhiều dòng: bắt đầu chú thích với dấu /* và kết thúc bởi */
- Chú thích một dòng: bắt đầu dòng chú thích bởi dấu //
- Ngoài hai kiểu chú thích trên giống trong Java, C++ thì C# còn hỗ trợ thêm kiểu thứ ba, kiểu này chứa các định dạng XML nhằm xuất ra tập tin XML khi biên dịch để tạo tài liệu chú thích cho mã nguồn. Các chú thích XML rất đơn giản, được tạo bằng cách đặt dấu /// trước các lớp, giao tiếp, phương thức hoặc trước phần khai báo thuộc tính... Có hơn 20 loại thẻ khác nhau có thể được sử dụng trong các chú thích XML, và được chia làm 2 loại chính: thẻ chính và thẻ phụ.

Sau khi tạo chú thích, mở cửa sổ Properties của ứng dụng, chọn trang Build, rồi kích chọn mục XML documentation file. Biên dịch ứng dụng, sẽ phát sinh tập tin tài liệu chú thích .XML.

Hay sử dụng trình biên dịch dòng lệnh với tùy chọn
 /doc:TậpTin.xml

csc /doc:Hello.xml Hello.cs

Ví dụ: Chú thích cho một hay nhiều dòng

```
using System;
namespace Hello
{
    class Hello
    {
        /* Xuất ra màn hình lời chào
        Sử dụng phương thức WriteLine của lớp System.Console
        */
        static void Main()
        {
            //Xuất chuỗi "Welcome to C#"
            Console.WriteLine("Welcome to C#");
        }
    }
}
```

Chú thích tài liệu XML

Các thẻ chính:

Là những thẻ mà bạn sử dụng một cách độc lập với những thẻ khác.

a1. <summary>

Thẻ <summary> mô tả kiểu và thành viên của kiểu như mô tả lớp, phương thức, thuộc tính...

a2. <remarks>

Thẻ <remarks> sử dụng để mô tả một kiểu, bổ sung thông tin cho thẻ <summary>.

a3. <value>

Thẻ <value> mô tả giá trị của một thuộc tính.

```
/// <value>
/// Họ và tên nhân viên
/// </value>
private string fullname;
```

a4. <param>

Thẻ *<param>* được sử dụng để chú thích từng tham đổi của một phương thức.

a5. <returns>

Thẻ này được sử dụng để chú thích giá trị trả về của một phương thức.

```
/// <summary>
/// Lớp thư viện chứa các phương thức xử lý chuỗi
/// </summary>
public class StringProcess
{
    /// <remarks>
    /// Phương thức trả về chuỗi con của một chuỗi
    /// </remarks>
    /// <param name="st">Chuỗi cần lấy chuỗi con</param>
    /// <param name="startIndex">Vị trí bắt đầu lấy chuỗi con
    /// </param>
    /// <param name=" endIndex">Vị trí kết thúc chuỗi con
    /// </param>
    /// <returns>Trả về chuỗi con từ vị trí startIndex đến vị trí
    /// endIndex</returns>
    public static string SubString(string st, int startIndex, int
endIndex) {}
}
```

a6. <exception>

Thẻ <exception> chú thích ngoại lệ mà phương thức phát sinh bởi thuộc tính cref và giải thích khi nào ngoại lệ phát sinh.

a7. <example>

Thẻ này có thể được sử dụng để cung cấp một ví dụ về cách sử dụng phương thức, thuộc tính. Các ví dụ là một phần chính của việc ghi chép tài liệu chất lượng cao, là hướng dẫn tốt cho các nhà phát triển. Bằng cách sử dụng thẻ <example> cùng với thẻ <code> (một trong những thẻ phụ), bạn có thể cung cấp trực tiếp các ví dụ.

```
/// <summary>
/// Phương thức xóa và trả về phần tử ở đỉnh của ngăn xếp stack
/// </summary>
/// <returns>Trả về giá trị phần tử ở đỉnh stack</returns>
/// <exception cref="EmptyStackException">
/// Ngoại lệ phát sinh khi stack rỗng
/// </exception>
/// <example>
/// <code>
/// // Lấy phần tử ở đỉnh của stack
/// string st = s.Pop();
/// </code>
/// </example>
public string Pop() {
```

a8. <permission>

Thẻ <permission> cho phép bạn xác định quyền truy cập một thành viên.

```
using namespace System;
```

```
public class TestClass
{
    /// <permission cref="System.Security.PermissionSet">
    Mọi người có thể truy cập phương thức này
    </permission>
    void Test() {}
}
```

a9. <seealso>

Có thể được sử dụng để tham chiếu các lớp, thành viên của lớp hoặc tài liệu bổ sung khác.

```
public class TestClass
{
    /// <summary>DoWork là một phương thức của lớp
    /// TestClass
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1) {}
    static void Main() {}
}
```

a10. <include>

Thẻ <include> được sử dụng để tham chiếu đến các chú giải trong tập tin khác. Thay vì chú thích trong mã nguồn, bạn có thể sử dụng một tập tin chú giải bên ngoài. Để sử dụng thẻ này, bạn cần xác định tên tập tin chứa chú giải và biểu thức XPath chỉ đến chú giải:

```
<include file = 'TậpTin' path = 'ĐườngDẫnThẻ [@name =
    "TênThẻ"]'/>
```

Tập tin: tên tập tin chứa tài liệu chú giải

Đường dẫn thẻ: biểu thức XPath hợp lệ chọn thẻ chứa chú giải trong tập tin

Tên thẻ: tên của thẻ chứa chú giải

```
/// <include file='xml_include_tag.doc'
/// path='MyDocs/MyMembers[@name="test"]' />
public class Test
{
    void TestMethod() {}
}
```

Các thẻ phụ

Là thẻ được sử dụng bên trong các thẻ chính. Những thẻ này được sử dụng để đánh dấu và định dạng Text để đưa vào các thẻ chính.

b1. <c> và <code>

Thẻ <code> được sử dụng định nghĩa một hay nhiều dòng là mã. Và thẻ <c> định nghĩa một phần chú thích là mã.

```
/// <summary>
/// Thay thế các ký tự đặc biệt bằng dấu cách <c>ReplaceSpecial
Char("a")</c>
/// </summary>
/// <param name="s"></param>
/// <returns>Trả về chuỗi sau khi đã thay thế ký tự đặc biệt bằng
dấu cách</returns>
/// <example>
/// <code>
/// Chuỗi ký tự đặc biệt
/// string specialChar = "-%*";
```

```

/// // gán các ký tự đặc biệt thành ký tự rỗng.
/// s = s.ReplaceSpecialChar("%---**");
/// </code>
/// </example>
private static string ReplaceSpecialChar(string s){}

```

b2. <para>

Được sử dụng để chỉ định một đoạn trong các chú thích. Nếu các chú thích dài, bạn nên ngắt nó thành các đoạn để làm cho việc đọc trở nên dễ dàng hơn.

b3. <see>

Thẻ <see> có thể sử dụng giống như thẻ <seealso> nhưng thẻ <see> sử dụng bên trong một thẻ khác. Khi bạn muốn liệt kê một số phương thức của một lớp và sử dụng thẻ <see> để tham chiếu đến những phương thức đó.

```

public class TestClass
{
    /// <summary>DoWork là phương thức của lớp TestClass
    /// <para> Ngắt đoạn chú thích <see
    cref="System.Console.WriteLine"/> cho thông tin về lệnh
    xuất </para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1) {}
    static void Main() {}
}

```

b4. <paramref>

Thẻ này xác định tham đối nào mà bạn tham chiếu đến

/// <remarks>Phương thức MyMethod thuộc lớp MyClass

```
/// <paramref name="i"/> tham số là một số nguyên  
/// </remarks>  
public static void MyMethod(int i) {}
```

b5. Các thẻ List

Các thẻ List để tạo các danh sách. Thẻ <list> có một thuộc tính Type, định nghĩa loại danh sách nào mà bạn đang tạo có thẻ có giá trị bullet, number hoặc table.

Thẻ con của thẻ <list> là thẻ <listheader> có thể được sử dụng để định nghĩa tiêu đề cho danh sách. Nó có thể chứa các thẻ <term> và <description>.

Thẻ <list> có thể chứa các thẻ <item>. Mỗi thẻ <item> là một mục trong danh sách và có thể bao gồm các thẻ <term> và thẻ <description>.

```
public class MyClass  
{  
    /// <remarks> Đây là ví dụ thẻ tạo danh sách  
    /// <list type="bullet">  
    /// <item>  
    /// <description>Item 1.</description>  
    /// </item>  
    /// <item>  
    /// <description>Item 2.</description>  
    /// </item>  
    /// </list>  
    /// </remarks>  
    public static void Main (){}  
}
```

2.2. KIỂU DỮ LIỆU, BIẾN, HÀNG

2.2.1. Kiểu dữ liệu (Data type)

C# là một ngôn ngữ định kiểu mạnh mẽ. Kiểu dữ liệu là tập hợp các giá trị mà một biến thuộc kiểu đó có thể nhận được. Mỗi biến phải khai báo thuộc một kiểu dữ liệu.

Các kiểu dữ liệu trong ngôn ngữ C# chia làm hai loại, kiểu dữ liệu xây dựng sẵn và kiểu dữ liệu do người dùng định nghĩa. C# cũng phân loại các kiểu dữ liệu thành hai loại: kiểu dữ liệu giá trị (value data type), và kiểu dữ liệu tham chiếu (reference data type). Việc phân chia này do sự khác nhau khi lưu kiểu dữ liệu giá trị và kiểu dữ liệu tham chiếu trong bộ nhớ. Biến kiểu dữ liệu giá trị lưu giữ giá trị của nó trong bộ nhớ tĩnh (stack). Trong khi đó biến kiểu tham chiếu, không chứa đối tượng thuộc kiểu này, mà lưu một địa chỉ hay tham chiếu đến đối tượng trong bộ nhớ stack, còn đối tượng thật sự lưu trong bộ nhớ động (heap). Nếu chúng ta có một đối tượng có kích thước rất lớn thì việc lưu giữ chúng trên bộ nhớ heap rất có ích. Tất cả các kiểu dữ liệu cơ sở xây dựng sẵn là kiểu dữ liệu giá trị. Kiểu chuỗi và các lớp định nghĩa sẵn là kiểu dữ liệu tham chiếu. Và tất cả các kiểu do người dùng định nghĩa ngoại trừ kiểu cấu trúc, liệt kê đều là kiểu dữ liệu tham chiếu. Ngoài ra C# cũng hỗ trợ kiểu con trỏ như C++, nhưng ít khi được sử dụng, khi làm việc với những đoạn mã lệnh có sử dụng kiểu con trỏ, sẽ được đánh dấu không an toàn (unsafe).

2.2.1.1. Kiểu dữ liệu giá trị

Bao gồm các kiểu dữ liệu cơ sở định nghĩa sẵn, kiểu liệt kê và kiểu cấu trúc. Kiểu dữ liệu cấu trúc người dùng định nghĩa sẽ được tìm hiểu trong chương kế tiếp cùng với kiểu lớp và kiểu tham chiếu phương thức (delegate).

Kiểu cơ sở

Ngôn ngữ C# đưa ra các kiểu dữ liệu cơ sở rất hữu dụng, mỗi kiểu dữ liệu được ánh xạ đến một kiểu dữ liệu hỗ trợ bởi .NET. Việc

ánh xạ các kiểu dữ liệu cơ sở của C# đến các kiểu dữ liệu của .NET sẽ đảm bảo các đối tượng được tạo ra trong C# có thể được sử dụng đồng thời với các đối tượng được tạo bởi bất kỳ ngôn ngữ khác được biên dịch bởi .NET.

Các kiểu dữ liệu cơ sở là kiểu dữ liệu struct xây dựng sẵn chứa nhiều thuộc tính và phương thức cần thiết, trong đó phương thức thường dùng là: Parse(string s): chuyển chuỗi s thành kiểu struct tương ứng

Ví dụ: float a = float.Parse(s);

Bảng 2.1. Kiểu dữ liệu cơ sở

Kiểu	Mô tả	Độ lớn	Kiểu .Net	Phạm vi
bool	Kiểu logic	1 bit	Boolean	true, false
char	Ký tự Unicode	16 bits	Char	0 ... 65535
byte	Số nguyên	8 bits	Byte	0 ... 255
sbyte	Số nguyên	8 bits	SByte	-128 ... 127
short	Số nguyên	16 bits	Int16	-32768 ... 32767
ushort	Số nguyên	16 bits	UInt16	0 ... 65535
int	Số nguyên	32 bits	Int32	-2147483648 ... 2147483647
uint	Số nguyên	32 bits	UInt32	0 ... 4294967295
long	Số nguyên	64 bits	Int64	-9223372036854775808 ... 9223372036854775807
ulong	Số nguyên	64 bits	UInt64	0 ... 18446744073709551615
float	Số thực	32 bits	Single	-3.402823E38 ... 3.402823E38
double	Số thực	64 bits	Double	-1.79769313486232E308 ... 1.79769313486232E308
decimal	Số thực	64 bits	Decimal	-79228162514264337593543950335 ... 79228162514264337593543950335

Kiểu liệt kê

Kiểu liệt kê đơn giản là tập hợp các tên hằng có giá trị không thay đổi, giúp bạn tổ chức dữ liệu khoa học hơn, mã trong sáng dễ hiểu hơn. C# cung cấp rất nhiều kiểu liệt kê định nghĩa sẵn, sẽ được đề cập đến trong các chương kế tiếp.

Để định nghĩa một kiểu liệt kê ta thực hiện theo cú pháp sau:

```
enum TênKiểuLiệtKê [KiểuCơSở]
{
    //Danh sách các thành phần liệt kê
};
```

Một kiểu liệt kê bắt đầu với từ khóa enum, tiếp sau là một định danh cho kiểu liệt kê.

Danh sách liệt kê là các hằng được gán giá trị và mỗi thành phần phải phân cách nhau dấu phẩy. Nếu chúng ta không khởi tạo giá trị cho các thành phần này thì chúng sẽ nhận các giá trị liên tiếp bắt đầu từ 0. Mỗi tên hằng trong kiểu liệt kê thuộc bất kỳ một kiểu dữ liệu cơ sở nguyên nào như int, short, long..., ngoại trừ kiểu ký tự.

Kiểu cơ sở chính là kiểu khai báo cho các thành phần liệt kê. Nếu bỏ qua thành phần này thì trình biên dịch sẽ gán mặc định là kiểu nguyên int. Khi đó, mỗi thành phần trong kiểu liệt kê tương ứng với một giá trị số nguyên.

Khi truy xuất giá trị của danh sách liệt kê, cần phải chuyển kiểu một cách tường minh.

Ví dụ:

```
using System;
class Example
{
    enum VisibleType {None = 0, Hidden = 2, Visible = 4};
    // In ra tên và giá trị thành phần của enum
    static void Main(string[] args)
    {
```

```

VisibleType visEnum = VisibleType.Hidden;
Console.WriteLine(visEnum.ToString());
Console.WriteLine((int)VisibleType.Hidden);
}
}

```

Kiểu Nullable (Nullable type)

C# 2.0 cung cấp kiểu struct System.Nullable<T> thuộc không gian tên System cho phép kiểu giá trị có giá trị null. Tất cả kiểu dữ liệu có giá trị null tương ứng. Một kiểu Nullable có thể biểu diễn một dãy giá trị của kiểu tương ứng và giá trị null. Chẳng hạn như biến kiểu Nullable<Int32> có thể gán giá trị từ -2147483648 đến 2147483647 hay giá trị null.

Kiểu Nullable khai báo cho các biến kiểu giá trị mà có thể gán giá trị null. Bạn không thể tạo một kiểu Nullable dựa trên kiểu tham chiếu. Kiểu tham chiếu đã hỗ trợ giá trị null là tham chiếu rỗng.

Cú pháp T? sử dụng để khai báo kiểu Nullable<T>, trong đó T là kiểu giá trị. Phép toán ?? để gán giá trị mặc định của biến kiểu Nullable có giá trị là null, cho biến không có kiểu Nullable.

Ví dụ:

int? x = null;	//x không chứa bất kỳ giá trị
int y = x ?? -1;	//x chứa giá trị 0
int? a = default(int);	//a không chứa giá trị 0
int? b = default(int?);	//b không chứa bất kỳ giá trị

Kiểu Nullable cung cấp thuộc tính và phương thức sau:

- Value: Trả về giá trị của biến và phát sinh ngoại lệ InvalidOperationException nếu biến kiểu Nullable không chứa bất kỳ giá trị

Ví dụ:

int c = a.Value;	//trả về 0
int d = b.value;	//phát sinh ngoại lệ

- HasValue: Trả về true nếu biến kiểu Nullable có giá trị, ngược lại trả về false

Ví dụ:

```
bool e = a.HasValue;    //trả về true
bool f = b.HasValue;    //trả về false
if (y.HasValue) Console.WriteLine(y.Value);
```

- GetValueOrDefault(): nếu biến kiểu Nullable có giá trị, trả về giá trị, ngược lại trả về giá trị mặc định của kiểu tương ứng

Ví dụ:

```
int g = a.GetValueOrDefault();    //trả về 0
int h = b.GetValueOrDefault();    //trả về 0
```

2.2.1.2. Kiểu dữ liệu tham chiếu

Kiểu chuỗi

Kiểu dữ liệu chuỗi (string hay String) là kiểu tham chiếu xây dựng sẵn, lưu giữ một dãy ký tự. Chuỗi trong C# là không thay đổi, mỗi khi chuỗi thay đổi, một chuỗi mới được tạo ra để chứa chuỗi kết quả.

Tạo một đối tượng chuỗi: Nhiều string được tạo từ các hằng chuỗi. Khi trình dịch bắt gặp một chuỗi ký tự bao giữa cặp nháy kép, nó tạo ra một đối tượng chuỗi có giá trị là chuỗi này. Bạn có thể dùng hằng chuỗi ở bất kỳ đâu bạn dùng đối tượng string

```
String s = "Hello World";
Hoặc string s = "Hello World";
```

Bạn có thể tạo đối tượng chuỗi như bất kỳ đối tượng khác, dùng từ khoá new.

Khởi tạo chuỗi chứa 2 ký tự ‘a’
 string s = new string('a', 2); //s = “aa”
 Khởi tạo chuỗi từ mảng ký tự

```
char[] ch = { 'a', 'b', 'c' };
string s = new string(ch); //s = "abc"
```

Khởi tạo chuỗi từ 2 phần tử trong mảng ch, bắt đầu từ phần tử thứ 1:

```
string s = new string(ch, 1, 2); //s = "bc"
```

Chuyển kiểu thành kiểu String: Để chuyển 1 kiểu dữ liệu nào đó về String, sử dụng phương thức đổi tượng ToString() của đối tượng cần chuyển kiểu, hay phương thức tĩnh ToString của lớp Convert.

Ví dụ:

```
int Number = 10;
String Str = Number.ToString();
Str = Convert.ToString(Number);
```

Truy cập từng ký tự của chuỗi:

Truy cập các ký tự của chuỗi bằng chỉ số và sử dụng dấu [] như mảng, chỉ số có giá trị bắt đầu từ 0, ví dụ s[0].

Các thuộc tính và phương thức thường dùng của string:

- int Length : Cho chiều dài chuỗi
int len = Str.Length;
- int len = "Hello World".Length;
- bool Equals(string value)
- bool Equals(string value, StringComparison comparisionType): kiểm tra hai chuỗi có giá trị bằng nhau không, trả về true nếu 2 chuỗi có giá trị bằng nhau, ngược lại trả về false, hoặc có thể sử dụng dấu = để so sánh chuỗi

Ví dụ:

```
string s1 = "Hello";
string s2 = s1;
Console.WriteLine("s1 equals s2 : "+s1.Equals(s2));
```

```
//So sánh chuỗi không phân biệt hoa thường  
Console.WriteLine(s1.Equals(s2, StringComparison.OrdinalIgnoreCase));  
Console.WriteLine("s1 == s2 : "+(s1==s2));  
Kết quả là :  
s1 equals s2 : True  
True  
s1 == s2 : True
```

- int CompareTo(string value): So sánh 2 chuỗi, trả về giá trị
nếu < 0 : chuỗi nhỏ hơn value
nếu > 0 : chuỗi lớn hơn value
nếu = 0 : chuỗi bằng value
- int IndexOf(char value), int IndexOf(string value): Trả về vị trí tìm thấy đầu tiên của ký tự hay chuỗi value
- int IndexOf(char value, int startIndex), int IndexOf(string value, int startIndex): Trả về vị trí tìm thấy đầu tiên của ký tự hay chuỗi value kể từ vị trí startIndex đến cuối chuỗi
- int IndexOf(string value, StringComparison comparisionType): Trả về vị trí tìm thấy đầu tiên của chuỗi value với kiểu so sánh phân biệt hoa thường hay không
- int IndexOf(char value, int startIndex, int count), int IndexOf(string value, int startIndex, int count): Trả về vị trí tìm thấy đầu tiên của ký tự hay chuỗi value trong count ký tự của chuỗi kể từ vị trí startIndex
- int IndexOf(string value, int startIndex, StringComparison comparisionType)
- int IndexOf(string value, int startIndex, int count, StringComparison comparisionType)

Tương tự phương thức IndexOf, có 9 phương thức LastIndexOf sử dụng để trả về vị trí tìm thấy cuối cùng:

- string Substring(int startIndex): Trả về chuỗi con của một chuỗi bắt đầu từ vị trí startIndex đến cuối chuỗi
- string Substring(int startIndex, int length): Trả về chuỗi con gồm length ký tự của một chuỗi bắt đầu từ vị trí startIndex

```
string org = "This is a test";
string result1 = org.Substring(10); //result1 = "test"
string result2 = org.Substring(8,1); //result2 = "a"
```
- string Replace(string oldValue, string newValue): Trả về chuỗi sau khi thay thế tất cả các lần xuất hiện của chuỗi oldValue bằng chuỗi newValue
- string Trim(): Trả về chuỗi sau khi cắt bỏ khoảng trắng đầu và cuối chuỗi
- string TrimEnd(): Trả về chuỗi sau khi cắt bỏ khoảng trắng cuối chuỗi
- string TrimStart(): Trả về chuỗi sau khi cắt bỏ khoảng trắng đầu chuỗi
- string ToLower(): Trả về chuỗi thường của chuỗi
- string ToUpper(): Trả về chuỗi chữ hoa của chuỗi

```
string s = "This is a test";
string upper = s.ToUpper();
```
- bool EndsWith(string s): Kiểm tra chuỗi kết thúc bởi 1 chuỗi xác định s hay không
- bool StartsWith(string s): Kiểm tra chuỗi bắt đầu bởi 1 chuỗi xác định s hay không
- string Insert(int index, string value): Trả về chuỗi sau khi chèn chuỗi value vào chuỗi tại vị trí index
- string Remove(int index): Xóa chuỗi kể từ vị trí index đến cuối chuỗi, và trả về chuỗi mới sau khi xóa
- bool StartsWith(string value): bool StartsWith(string value, StringComparison comparisonType), bool StartsWith(string

value, bool ignoreCase, CultureInfo culture): Trả về true nếu chuỗi bắt đầu bởi chuỗi value, ngược lại trả về false. ComparisonType chỉ kiểu so sánh chuỗi, là các hằng liệt kê của enum StringComparison như OrdinalIgnoreCase: không phân biệt hoa thường, Ordinal, CurrentCulture: so sánh theo ngôn ngữ hiện hành.

- string[] Split(params char[] separator): Trả về một mảng các chuỗi con được phân tách bởi một trong các ký tự trong mảng separator
- string[] Split(char[] separator, StringSplitOptions options): Trả về một mảng các chuỗi con được phân tách bởi một trong các ký tự trong mảng separator. Tham số StringSplitOptions.RemoveEmptyEntries để xóa các chuỗi con rỗng, ngược lại không xóa chuỗi con rỗng sử dụng tùy chọn StringSplitOptions.None.

Ví dụ:

```
using System;
class Program
{
    static void Main()
    {
        string s = "there, is a cat";
        string[] words = s.Split(' ', ',');
        foreach (string word in words)
            Console.WriteLine(word);
        char[] delimiters = new char[] { ',', ',' };
        words = s.Split(delimiters, StringSplitOptions.RemoveEmptyEntries);
        foreach (string word in words)
            Console.WriteLine(word);
    }
}
```

- Toán tử +: Để nối hai đối tượng string, hay một đối tượng string và một giá trị khác thành đối tượng string

Ví dụ: Console.WriteLine("Nghiệm là " + x1 + " và " + x2);

- static string Concat(params object[]): Nối nhiều đối tượng thành một chuỗi

Console.WriteLine(string.Concat("Nghiệm là ", x1, " và ", x2));

- static string Format(string format, params object[]): Trả về chuỗi bằng cách thay thế các ký hiệu đánh dấu trong chuỗi format bởi các tham đối theo sau chuỗi. Có thể định dạng các đối tượng object tương tự định dạng kết xuất của phương thức Write trình bày cuối chương.

string.Format("Nghiệm là {0} và {1}", x1, x2);

string.Format("{0:dd/MM/yyyy}", DateTime.Today);

string.Format("{0:n2}", 12345678);

string.Format("{0:c0}", 12345678);

string.Format("{0:0.##}", 123.4567);

Kiểu mảng

Mảng là một cấu trúc lưu giữ các thành phần cùng kiểu. Mỗi thành phần của mảng được truy xuất bởi chỉ số của nó trong mảng.

Mảng 1 chiều:

- Khai báo một biến dùng để tham chiếu đến mảng, nhưng không có mảng nào thật sự tồn tại

KiểuPhànTử[] TênMảng;

Kiểu phần tử là kiểu dữ liệu của các thành phần của mảng, có thể là bất kỳ kiểu giá trị hay tham chiếu.

Ví dụ: int[] m;

Tạo một mảng: Sử dụng toán tử new để cấp phát bộ nhớ cho mảng và gán mảng đến biến đã khai báo

TênMảng = **new** KiểuPhànTử[KíchThướcMảng];

Kích thước mảng là số thành phần trong mảng

Ví dụ: `m = new int[10];`

Có thể kết hợp khai báo biến mảng và tạo mảng như sau:

`KiểuPhầnTử[] TênMảng = new KiểuPhầnTử[KíchThướcMảng];`

Ví dụ:

`int[] m = new int[10];`

`string[] myStrings = new string[3];`

- Truy xuất thành phần của mảng: `TênMảng[ChiMục]`

Chi mục: chỉ vị trí của thành phần trong mảng cần truy xuất, có thể là giá trị, biến hay biểu thức và có giá trị từ 0 đến Kích thước mảng - 1

Ví dụ: `m[0] = 20;`

- Lấy kích thước mảng: `TênMảng.Length`
- Khai báo, cấp phát bộ nhớ và khởi tạo giá trị đầu của mảng:
Mảng có thể khởi tạo khi khai báo sử dụng toán tử `new` hay không. Mảng khởi tạo là danh sách các biểu thức cách nhau dấu phẩy, đặt trong ngoặc mộc. Chiều dài mảng là số thành phần giữa hai dấu { và }

Ví dụ: `int[] numbers = new int[5] {1, 2, 3, 4, 5};`

Có thể bỏ qua kích thước của mảng như sau:

`int[] numbers = new int[] {1, 2, 3, 4, 5};`

`string[] names = new string[] {"Xuan", "Ha", "Thu"};`

Cũng có thể bỏ qua toán tử `new` như sau:

`int[] numbers = {1, 2, 3, 4, 5};`

`string[] names = {"Xuan", "Ha", "Thu"};`

`int[] monthDays = {31,28,31,30,31,30,31,31,31,30,31,31};`

C# cũng hỗ trợ mảng không kiểu bằng cách tạo mảng các đối tượng như sau:

`object[] arr = {1,"Hi there", 5.5};`

- Biến kiểu không tường minh (Implicitly typed local variable): C# 3.0 bổ sung từ khóa `var` cho phép khai báo và

khởi tạo giá trị đầu cho mảng mà không cần khai báo kiểu tường minh (implicitly typed local variable) như sau:

```
var arr = new int[] {1, 2, 3, 4, 5};
```

Ví dụ:

```
using System;
namespace ThamDoi
{
    class ThamDoi
    {
        static void Main(string[] args)
        {
            for (int i=0; i < args.Length; i++)
                Console.WriteLine("Tham doi thu "+i+": "+args[i]);
        }
    }
}
```

Biên dịch và chạy chương trình:

```
csc Thamdoi.cs ↴
```

```
ThamDoi Thu tham doi dong lenh ↴
```

Cho kết quả: Tham doi thu 0 : Thu Tham doi thu 1 : tham ...

Mảng đa chiều (Array of arrays):

Mảng có thể chứa các thành phần là mảng:

- Để khai báo một biến mảng đa chiều cần xác định mỗi chiều của mảng bằng cách sử dụng các cặp dấu ngoặc vuông.

Ví dụ: Khai báo mảng 2 chiều m là một mảng 4 dòng, và có thành phần là các số nguyên.

```
int[][] m = new int[4][];
m[0] = new int[2];           //dòng 1 có 2 cột
m[1] = new int[5];           //dòng 2 có 5 cột
Console.WriteLine(m.Length); //số dòng của mảng là 4
```

Hay có thể khai báo như sau:

```
double[,] myDoubles = new double[2, 2];
int[,] myInts = new int[4,5,3];
```

- Truy xuất thành phần của mảng $m[i][j]$ (phần tử tại dòng i , cột j). Với i là chỉ số dòng, có giá trị từ 0 đến số dòng -1, j là chỉ số cột, có giá trị từ 0 đến số cột - 1. Hay $m[i,j]$.
- Lấy kích thước mảng:

Với mảng khai báo như sau: `int[][] m = new int[4][];`

Kích thước của chiều thứ nhất, nghĩa là số dòng: `m.Length`.

Kích thước của chiều thứ hai, nghĩa là số cột.

Số phần tử trên dòng có thể khác nhau trên mỗi dòng. Ta có kích thước của dòng thứ i `m[i].Length`

Với mảng khai báo như sau: `int[] m = new int[4,5];`

Số tất cả các phần tử mảng ở tất cả các chiều: `m.Length`

Kích thước của chiều thứ nhất, nghĩa là số dòng: `m.GetLength(0)`;

Kích thước của chiều thứ hai, nghĩa là số cột: `m.GetLength(1)`;

Khai báo, cấp phát bộ nhớ và khởi tạo giá trị đầu của mảng

Ví dụ:

```
int[,] numbers = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
int[][] numbers = new int[2][] { new int[] {2,3,4}, new int[] {5,6,7,8,9} };
string[,] nhom = new string[2, 2] { {"Hoa", "Thu"}, {"Hue", "Cuc"} };
```

Có thể bỏ qua kích thước của mảng:

```
int[,] numbers = new int[,] { {1, 2}, {3, 4}, {5, 6} };
int[][] numbers = new int[][] { new int[] {2,3,4}, new int[] {5,6,7,8,9} };
int[][] numbers = { new int[] {2,3,4}, new int[] {5,6,7,8,9} };
string[,] nhom = new string[,] { {"Hoa", "Thu"}, {"Hue", "Cuc"} };
```

Cũng có thể bỏ qua toán tử `new` như sau:

```
int[,] numbers = { {1, 2}, {3, 4}, {5, 6} };
string[,] nhom = { {"Hoa", "Thu"}, {"Hue", "Cuc"} };
```

Ví dụ: Tạo ma trận các phần tử số nguyên có giá trị bằng chỉ số hàng + chỉ số cột của phần tử, và in ma trận

```
using System;
class MaTran {
    static void Main(string[] args) {
        int[,] a = new int[4][];
        for (int i = 0; i < a.Length; i++)
            a[i] = new int[5];
        for (int i = 0; i < a.Length; i++)
            for (int j = 0; j < a[i].Length; j++)
                a[i][j] = i + j;
        for (int i = 0; i < a.Length; i++) {
            for (int j = 0; j < a[i].Length; j++)
                Console.Write(a[i][j] + "\t");
            Console.WriteLine();
        }
    }
}
```

Ví dụ: Một cách viết khác để tạo ma trận các phần tử số nguyên có giá trị bằng chỉ số hàng + chỉ số cột của phần tử, và in ma trận

```
using System;
class Program {
    static void Main(string[] args){
        int[,] a = new int[4,5];
        for (int i = 0; i < a.GetLength(0); i++)
            for (int j = 0; j < a.GetLength(1); j++)
                a[i,j] = i + j;
        for (int i = 0; i < a.GetLength(0); i++) {
            for (int j = 0; j < a.GetLength(1); j++)
                Console.Write(a[i,j] + "\t");
            Console.WriteLine();
        }
    }
}
```

Kiểu con trỏ

C# cũng hỗ trợ kiểu con trỏ như C++, nhưng ít khi được sử dụng, khi đoạn mã lệnh sử dụng kiểu con trỏ, sẽ được đánh dấu không an toàn (unsafe). Khai báo biến kiểu con trỏ như sau:

Kiểu* TênBiến;

void* TênBiến; //cho phép nhưng không nên dùng

Biến kiểu con trỏ không chứa giá trị mà chứa tham chiếu đến biến hay nói cách khác giá trị của biến con trỏ là địa chỉ của biến

Ví dụ	Mô tả
int* p	p là một con trỏ tham chiếu đến biến nguyên
int** p	p là một con trỏ tham chiếu đến con trỏ đến biến nguyên
int*[] p	p là một mảng của các con trỏ tham chiếu đến biến nguyên
char* p	p là con trỏ tham chiếu đến biến char
void* p	p là con trỏ tham chiếu đến biến kiểu void

Khi bạn khai báo nhiều con trỏ cùng kiểu, chỉ sử dụng một dấu * cho nhiều biến của cùng một khai báo, không sử dụng mỗi dấu * cho mỗi biến:

int* p1, p2, p3;

Toán tử * có thể được sử dụng để truy cập giá trị của biến được trỏ đến bởi biến con trỏ. Bạn không thể sử dụng toán tử * cho con trỏ kiểu void*, tuy nhiên có thể chuyển kiểu con trỏ void* đến bất kỳ kiểu con trỏ khác.

Bảng 2.2 Các phép toán thực hiện trên biến con trỏ

Phép toán	Ý nghĩa
*	Lấy giá trị của biến mà con trỏ tham chiếu đến
->	Truy cập thành phần của cấu trúc struct được tham chiếu bởi con trỏ
[]	Truy cập thành phần của mảng tham chiếu bởi con trỏ
&	Lấy địa chỉ của biến
++ and --	Phép toán tăng và giảm trên biến con trỏ

+ and -	Thực hiện phép toán số học trên biến con trả
==, !=, <, >, <=, and >=	Phép toán so sánh biến con trả

Ví dụ:

```
unsafe class TestClass
{
    static void Main()
    {
        char theChar = 'Z';
        char* pChar = &theChar;
        void* pVoid = pChar;
        int* pInt = (int*)pVoid;
        int number;
        int* p = &number;
        *p = 0xffff;
        System.Console.WriteLine("Địa chỉ của biến theChar = {0:X2}",
            (int) pChar);
        System.Console.WriteLine("Giá trị của biến pChar = {0}",
            *pChar);
        System.Console.WriteLine("Giá trị của biến pInt = {0}", *pInt);
        System.Console.WriteLine("Giá trị biến trả bởi con trả p:{0:X}",
            *p);
        System.Console.WriteLine("Địa chỉ lưu trong biến p: {0}",
            p->ToString());
        System.Console.WriteLine("Giá trị của biến number: {0:X}",
            number);
        p++;
        System.Console.WriteLine("Giá trị biến trả bởi con trả p:{0:X}",
            *p);
    }
}
```

Ví dụ:

```
using System;
struct Point
{
    public int x, y;
}
class AccessMembers
{
    static void Main()
    {
        Point home;
        unsafe
        {
            Point* p = &home;
            p->x = 25;
            p->y = 12;
            Console.WriteLine("The coordinates are:x={0},y={1}",
                p->x,p->y);
        }
    }
}
```

Ví dụ: Trong ví dụ sau, chú ý rằng biểu thức charPointer[i] tương đương với biểu thức *(charPointer + i)

```
using System;
class Pointers
{
    unsafe static void Main()
    {
        char* charPointer = stackalloc char[123];
        for (int i = 65; i < 123; i++)
            charPointer[i] = (char) i;
        Console.WriteLine("Uppercase letters:");
        for (int i = 65; i < 91; i++)
            Console.Write(charPointer[i]);
        Console.WriteLine();
```

```

Console.WriteLine("Lowercase letters:");
for (int i = 97; i < 123; i++)
    Console.Write(charPointer[i]);
}
}

```

Các danh sách collection xây dựng sẵn

C# cung cấp các lớp collection trong không gian tên System.Collections hay System.Collections.Generic, là các cấu trúc lưu trữ dữ liệu khác nhau.

ArrayList và List<T>

ArrayList là lớp thuộc không gian tên System.Collections, trong khi List<T> thuộc không gian tên System.Collections.Generic. ArrayList và List<T> đều cho phép tạo một dãy thứ tự các phần tử có kiểu dữ liệu bất kỳ, các phần tử này có thể có giá trị trùng nhau. Khác với mảng, ArrayList và List<T> có thể tự động thay đổi kích thước nếu cần thiết. Các phương thức, thuộc tính trên ArrayList tương tự List<T>.

- Khởi tạo một đối tượng kiểu ArrayList hay List<T>:

```

ArrayList list = new ArrayList();
List<KiểuDữLiệu> list = new List<KiểuDữLiệu>();

```

- Truy cập phần tử thứ i: list[i] với chỉ số i bắt đầu từ 0
- int Add(object value): Bộ sung phần tử value, trả về vị trí phần tử thêm vào
- void Insert(int index, object value): Chèn một phần tử value tại vị trí index
- void RemoveAt(int index): Xóa phần tử tại vị trí index
- void Remove(object value): Xóa phần tử value trong ArrayList
- void RemoveRange(int index, int count): Xóa một dãy count phần tử kể từ vị trí index
- void Clear(): Xóa tất cả các phần tử trong ArrayList

- bool Contains(object value): Kiểm tra phần tử có trong ArrayList, nếu có trả về true
- void Sort(): Sắp xếp các phần tử trong ArrayList
- int IndexOf(object value), int IndexOf(object value, int startIndex),
- int IndexOf(object value, int startIndex, int count): Trả về chỉ số của phần tử value trong count phần tử của ArrayList, kể từ vị trí startIndex, nếu không tìm thấy trả về -1
- int BinarySearch(object value): Tìm kiếm nhị phân trong danh sách đã sắp xếp
- int BinarySearch(int startIndex, int count, object value)
- int Count: Trả về số phần tử có trong ArrayList hay List<T>
- int Capacity: Thiết lập hay trả về số phần tử có thể chứa trong ArrayList
- bool IsFixedSize: Trả về giá trị chỉ rõ ArrayList có kích thước cố định hay không

Ví dụ:

```
using System;
using System.Collections;
class ViDu
{
    static void Main()
    {
        ArrayList list = new ArrayList();
        list.Add(3);
        list.Add(2);
        list.Insert(1, 6);
        list.Remove(3);
        for (int i = 0; i < list.Count; i++)
            Console.WriteLine(list[i].ToString());
    }
}
```

Hashtable

Hashtable là lớp thuộc không gian tên System.Collections, biểu diễn một bảng băm, là danh sách các cặp phần tử và khóa liên kết phần tử này.

- Khởi tạo đối tượng Hashtable: Hashtable h = new Hashtable();
- Truy cập phần tử thứ key: h[key]
- void Add(object key, object value): Thêm một cặp phần tử value, khóa key vào Hashtable, mỗi phần tử tương ứng với khóa
- bool ContainsKey(object key) hay bool Contains(object key): Kiểm tra khóa key có trong Hashtable chưa, nếu có phương thức trả về true
- bool ContainsValue(object value): Kiểm tra phần tử có trong Hashtable chưa, nếu có phương thức trả về true
- void Remove(object key): Xóa cặp khóa và phần tử tương ứng trong Hashtable
- ICollection Keys: Trả về một ICollection chứa các khóa trong Hashtable
- ICollection Values: Trả về một ICollection chứa các giá trị trong Hashtable

Ví dụ:

```
using System;
using System.Collections;
class ViDu
{
    static void Main()
    {
        Hashtable weeks = new Hashtable();
        weeks.Add("1", "Sunday");
        weeks.Add("2", "Monday");
        weeks.Add("3", "Tuesday");
        Console.WriteLine(weeks["2"].ToString());
```

```

        if (weeks.ContainsValue("Tuesday"))
            Console.WriteLine( "Tim thay");
        else Console.WriteLine( "Khong tim thay");
        weeks.Remove("3");
        foreach (DictionaryEntry day in weeks)
            Console.WriteLine( (day.Key + " - " + day.Value));
    }
}

```

SortedList và SortedList<T,T>

SortedList là lớp thuộc không gian tên System.Collections, trong khi SortedList<T,T> thuộc không gian tên System.Collections.Generic. SortedList và SortedList<T,T> biểu diễn một danh sách các cặp phần tử và khóa kết hợp phần tử này tương tự Hashtable, nhưng danh sách SortedList được sắp xếp theo khóa.

Mỗi cặp phần tử và khóa trong SortedList có kiểu struct KeyValuePair<TKey, TValue>, chưa thuộc tính Key trả về khóa và Value trả về giá trị của phần tử.

- Khởi tạo một đối tượng kiểu SortedList hay SortedList<T,T>:
`SortedList list = new SortedList();`
`SortedList<TKey, TValue> list = new SortedList<TKey, TValue>();`
- Truy cập phần tử thứ key: `list[key]`
- `void Add(object key, object value)`: Bổ sung phần tử value với khóa key
- `void RemoveAt(int index)`: Xóa phần tử tại vị trí index
- `bool Remove(object key)`: Xóa phần tử với khóa key
- `int IndexOfValue(object value)`: Tìm giá trị value, trả về vị trí tìm thấy, nếu không tìm thấy trả về -1
- `int IndexOfKey(object key)`: Tìm khóa key, trả về vị trí tìm thấy, nếu không tìm thấy trả về -1
- `bool ContainsKey(object key)`: Kiểm tra danh sách có chứa khóa key không, nếu có trả về true

- bool ContainsValue(object value): Kiểm tra danh sách có chứa giá trị value không, nếu có trả về true
- bool TryGetValue(object key, out object value): Trả về giá trị kết hợp với khóa key trong biến value, nếu tìm thấy trả về true

Ví dụ:

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main(string[] args)
    {
        SortedList<string, int> sorted = new SortedList<string, int>();
        sorted.Add("Hoa", 9);
        sorted.Add("Xuan", 7);
        sorted.Add("Dong", 8);
        bool contain1 = sorted.ContainsKey("Xuan");
        Console.WriteLine("Danh sach co Xuan: " + contain1);
        int value;
        if (sorted.TryGetValue("Dong", out value))
            Console.WriteLine("Gia tri co khoa Dong la " + value);
        foreach (var pair in sorted)
            Console.WriteLine(pair.Key + "-" + pair.Value);
        int index = sorted.IndexOfValue(8);
        Console.WriteLine("Vi tri cua diem 8: " + index);
    }
}
```

Dictionary<T, T>

Dictionary là lớp thuộc không gian tên System.Collections.Generic, biểu diễn một danh sách các cặp phần tử và khóa liên kết với phần tử này tương tự Hashtable. Kiểu dữ liệu của key và value có thể là bất kỳ kiểu cơ sở, string, mảng, kiểu liệt kê, hay struct. Mỗi cặp phần tử và khóa trong Dictionary có kiểu struct KeyValuePair<TKey, TValue>, chứa thuộc tính Key trả về khóa và Value trả về giá trị của phần tử.

- Khởi tạo đối tượng Dictionary: Dictionary< TKey, TValue>
dict = new Dictionary< TKey, TValue>();
- Truy cập phần tử thứ key: dict[key]
- void Add(object key, object value): Thêm một cặp phần tử value, khóa key vào Dictionary, mỗi phần tử tương ứng với khóa
- bool ContainsKey(object key): Kiểm tra khóa có trong Dictionary chưa, nếu có phuong thức trả về true
- bool ContainsValue(object value): Kiểm tra phần tử có trong Dictionary chưa, nếu có phuong thức trả về true
- bool Remove(object key): Xóa cặp khóa và phần tử tương ứng, trả về true nếu tồn tại cặp khóa và phần tử, ngược lại trả về false

Ví dụ:

```
using System;
using System.Collections.Generic;
class ViDu
{
    static void Main()
    {
        Dictionary<Color, string> colorStrings = new Dictionary<Color,
string>();
        colorStrings.Add(Color.Blue, Color.Blue.ToString());
        colorStrings.Add(Color.Red, Color.Red.ToString());
        foreach (KeyValuePair<Color, string> color in colorStrings)
            Console.WriteLine(color.Key.GetBrightness().ToString()+
" " + color.Value);
    }
}
```

NameValueCollection

NameValueCollection là lớp thuộc không gian tên System.Collections.Specialized, biểu diễn một danh sách các cặp chuỗi và tên liên kết với chuỗi này tương tự Hashtable. Các chuỗi có thể truy cập bởi tên hay chỉ số. NameValueCollection có thể lưu nhiều chuỗi kết hợp với một tên.

- Khởi tạo đối tượng NameValueCollection: NameValueCollection pair = new NameValueCollection();
- Truy cập phần tử với tên: pair[name] hay truy cập phần tử thứ index: pair[index]
- void Add(string name, string value): Thêm một cặp chuỗi value và tên name vào NameValueCollection
- void Remove(string name): Xóa chuỗi tương ứng tên name
- string[] GetValues(int index): Trả về một mảng các chuỗi tại vị trí index
- string Get(int index): Trả về chuỗi chứa danh sách các chuỗi tại vị trí index cách nhau dấu phẩy
- string GetKey(int index): Trả về tên liên kết với chuỗi tại vị trí index
- string[] AllKeys: Trả về mảng tất cả các khóa trong danh sách

Ví dụ:

```
using System;
using System.Collections.Specialized;
class ViDu
{
    static void Main()
    {
        NameValueCollection markStatus = new NameValueCollection();
        markStatus.Add("High", "80");
        markStatus.Add("High", "60");
        markStatus.Add("Pass", "40");
        foreach (string key in markStatus.Keys)
        {
            string[] values = markStatus.GetValues(key);
            foreach (string value in values)
                Console.WriteLine (key + " - " + value);
        }
    }
}
```

Stack và Stack<T>

Stack là lớp thuộc không gian tên System.Collections, trong khi Stack<T> thuộc không gian tên System.Collections.Generic. Lớp Stack và Stack<T> đều biểu diễn một ngăn xếp hoạt động theo nguyên tắc LIFO (Last In First Out), một phần tử thêm vào ngăn xếp cuối cùng, sẽ được lấy ra đầu tiên.

- Khởi tạo một đối tượng Stack: Stack stack = new Stack();
- Stack<KiểuPhầnTử> stack = new Stack<KiểuPhầnTử>();
- void Push(object value): Thêm một phần tử vào đỉnh ngăn xếp
- object Pop(): Lấy một phần tử từ đỉnh ngăn xếp
- bool Contains(object value): Kiểm tra giá trị có trong ngăn xếp không, nếu có trả về true
- object Peek(): Trả về một phần tử ở đỉnh của ngăn xếp mà không xóa nó

Ví dụ:

```
using System;
using System.Collections.Generic;
class ViDu
{
    static void Main()
    {
        Stack<string> days = new Stack<string>();
        days.Push("Sunday");
        days.Push("Monday");
        days.Push("Tuesday");
        days.Push("Wednesday");
        days.Push("Thursday");
        days.Push("Friday");
        days.Push("Saturday");
        if (days.Count == 7)
            Console.WriteLine(days.Pop().ToString());
        else Console.WriteLine("Saturday does not exist");
    }
}
```

Queue và Queue<T>

Queue là lớp thuộc không gian tên System.Collections, trong khi Queue<T> thuộc không gian tên System.Collections.Generic. Lớp Queue và Queue<T> đều biểu diễn một hàng đợi hoạt động theo nguyên tắc FIFO (First In First Out), một phần tử thêm vào hàng đợi đầu tiên, sẽ được lấy ra đầu tiên.

- Khởi tạo một đối tượng Queue: Queue<KiểuPhầnTử> queue = new Queue<KiểuPhầnTử>();
- void Enqueue(object value): Thêm một phần tử vào cuối hàng đợi
- object Dequeue(object value): Lấy một phần tử ở đầu hàng đợi
- object Peek(): Trả về phần tử ở đầu hàng đợi
- bool Contains(object value): Kiểm tra giá trị có trong hàng đợi không, nếu có trả về true

LinkedList<T>

LinkedList là lớp thuộc không gian tên System.Collections.Generic, biểu diễn một danh sách liên kết đôi, gồm danh sách các node liên kết với nhau bởi thành phần tham chiếu đến node kế tiếp và kế trước của mỗi node.

- Khởi tạo một đối tượng LinkedList: LinkedList<KiểuPhầnTử> list = new LinkedList<KiểuPhầnTử>();
- void AddAfter(LinkedListNode<T> node, int value): Chèn node có giá trị value sau node chỉ rõ
- void AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode): Chèn node mới newNode sau node chỉ rõ
- void AddBefore(LinkedListNode<T> node, int value): Chèn node có giá trị value trước node chỉ rõ
- void AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode): Chèn node mới newNode trước node chỉ rõ

- void AddFirst(int value): Thêm node có giá trị value vào đầu danh sách
- void AddFirst(LinkedListNode<T> node): Thêm node chỉ rõ vào đầu danh sách
- void AddLast(int value): Thêm node có giá trị value vào cuối danh sách
- void AddLast(LinkedListNode<T> node): Thêm node chỉ rõ vào cuối danh sách
- bool Contains(int value): Kiểm tra giá trị có trong danh sách không
- LinkedListNode<T> Find(int value): Tìm node đầu tiên trong danh sách có giá trị value
- LinkedListNode<T> FindLast(int value): Tìm node cuối cùng trong danh sách có giá trị value
- bool Remove(int value): Xóa node có giá trị value trong danh sách
- void Remove(LinkedListNode<T> node): Xóa node chỉ rõ trong danh sách
- void RemoveFirst(): Xóa node đầu tiên trong danh sách
- void RemoveLast(): Xóa node cuối cùng trong danh sách
- LinkedListNode<T> Last: Trả về node cuối danh sách
- LinkedListNode<T> First: Trả về node đầu danh sách

Mỗi node trong danh sách có kiểu `LinkedListNode<T>`, có các thuộc tính:

- `LinkedListNode<T> Next`: Trả về tham chiếu đến node kế tiếp
- `LinkedListNode<T> Previous`: Trả về tham chiếu đến node kế trước
- `T Value`: Trả về giá trị của node

Ví dụ:

```
using System;
using System.Collections.Generic;
class ViDu
{
    static void Main()
    {
        LinkedList<int> list = new LinkedList<int>();
        list.AddLast(2);
        list.AddFirst(1);
        LinkedListNode<int> node = list.First;
        while (node != null)
        {
            Console.WriteLine(node.Value.ToString());
            node = node.Next;
        }
    }
}
```

HashSet<T> và SortedSet<T>

HashSet là lớp thuộc không gian tên System.Collections.Generic, biểu diễn một tập hợp các phần tử không trùng nhau. .Net 4 bổ sung SortedSet<T> tương tự HashSet<T>, nhưng tập hợp được sắp xếp theo trật tự quy định bởi giao tiếp IComparer<T>.

- Khởi tạo một đối tượng HashSet : HashSet<KiểuPhầnTử> set = new HashSet<KiểuPhầnTử>();
- Có thể khởi tạo một HashSet từ một Collection khác:

```
string[] arr = { "cat", "dog", "cat", "leopard", "tiger", "cat" };
```

```
HashSet<string> hash = new HashSet<string>(arr);
```

- Có thể khi khởi tạo HashSet, quy định cách thức so sánh phần tử trùng nhau: HashSet<string> hash = new HashSet<string>(new Compare());

Với Compare là lớp cài đặt giao tiếp IEqualityComparer<T>

- bool Add(object item): Thêm một phần tử vào HashSet
- bool Remove(object item): Xóa phần tử item từ HashSet
- bool Contains(object item): Kiểm tra HashSet có chứa giá trị item, nếu có trả về true
- void CopyTo(object[] array): Sao chép HashSet đến mảng array

Ví dụ:

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main(string[] args)
    {
        HashSet<string> hash = new HashSet<string>(new Compare());
        hash.Add("1.Hoa");
        hash.Add("2.Lan");
        hash.Add("3.Thu");
        string[] arr = new string[3];
        hash.CopyTo(arr);
        Console.WriteLine(string.Join(", ", arr));
    }
}
class Compare : IEqualityComparer<string>
{
    public bool Equals(string a, string b)
    {
        if (GetHashCode(a) == GetHashCode(b))
            return true;
        return false;
    }
    public int GetHashCode(string a)
    {
        return int.Parse(a.Substring(0, a.IndexOf('.') ));
    }
}
```

2.2.2. Biến (Variable)

Là một vùng nhớ được định danh để lưu trữ dữ liệu hay địa chỉ dữ liệu. Mọi biến cần được định nghĩa trước là khai báo tên và kiểu dữ liệu trước khi sử dụng. Biến phải được khởi tạo trước khi sử dụng.

Khai báo biến thuộc kiểu dữ liệu cơ bản:

Kiểu TênBiến;

hay có thể khai báo và khởi tạo giá trị đầu cho biến:

Kiểu TênBiến = GiáTrị;

Nhiều biến cùng kiểu có thể khai báo cách nhau dấu phẩy, bạn có thể định nghĩa biến kiểu tham chiếu string như trên

Ví dụ:

```
string s = "Hello";           // String s = "Hello";
int x = 5, y = 7;            // Int32 x = 5, y=7;
bool b = true;               // Boolean b = true;
```

Biến kiểu không tường minh (Implicitly typed local variable):

C# 3.0 bổ sung từ khóa var cho phép định nghĩa biến kiểu không tường minh (implicitly typed local variable). Chú ý giá trị biến phải được gán khi khai báo biến, khi thực hiện lệnh gán, biến đã được ngầm định khai báo kiểu tương ứng. Hơn nữa, giá trị gán cho biến phải khác null.

Ví dụ: var i = 1;

Kiểu dynamic và ràng buộc trễ (dynamic type and late binding):

C# 4.0 giới thiệu một kiểu mới, kiểu dynamic. Không như kiểu ràng buộc tĩnh (static binding), các biến khai báo kiểu dynamic ràng buộc trễ vào lúc chạy chương trình, nghĩa là khi biên dịch, CLR không biết kiểu thực sự của biến, vì vậy nó hỗ trợ bất kỳ thao tác vào lúc biên dịch, có thể gán giá trị, đổi tượng kiểu bất kỳ cho biến dynamic. Không giống như var, có thể khai báo biến dynamic, sau đó gán giá trị cho biến.

Xét ví dụ sau:

```
dynamic d;
d = 7;
//Biên dịch không lỗi
//Báo lỗi vào lúc chạy do d là một int, không định nghĩa thuộc tính Length
Console.WriteLine(d.Length);
d = "a string";
Console.WriteLine(d.Length);
d = System.DateTime.Today;
```

2.2.3. Hằng (Constant và literal)

Hằng có hai loại: hằng giá trị (literal) và hằng được đặt tên (constant). Hằng đặt tên là một vùng nhớ để lưu trữ dữ liệu hay địa chỉ dữ liệu không thay đổi giá trị trong chương trình.

const Kiểu TênHằng = GiáTrị;

Ví dụ: const int max = 10;

Hằng giá trị là các giá trị thuộc kiểu nào đó.

C# xem hằng nguyên thuộc kiểu int, trừ khi trị nguyên vượt quá phạm vi kiểu int, khi đó hằng nguyên thuộc kiểu long. Muốn có hằng nguyên nhỏ thuộc kiểu long, bạn thêm L hay L vào cuối trị số. Hằng nguyên có thể được gán cho biến thuộc kiểu byte hay short miễn là trị nguyên được gán không vượt quá phạm vi của các kiểu này.

Ví dụ: 25L

Hằng số thực được C# mặc nhiên hiểu là trị thuộc kiểu double, bạn có thể ghi thêm D hay F (d hay f) ở cuối trị số để khẳng định rõ trị số thuộc kiểu double hay float. Hằng số thực có thể chứa ký tự E hay e.

Ví dụ: 6.5, 6.5F, 12.36E-2

Hằng ký tự được ghi trong cặp dấu nháy đơn.

Ví dụ: 'a', '5'

Ký tự đặc biệt được thêm dấu \ gọi là ký tự thoát (escape character) phía trước

Bảng 2.3. Các ký tự đặc biệt

Ký tự đặc biệt	Ý nghĩa
\n	Ký tự xuống dòng
\t	Ký tự tab ngang
\v	Ký tự tab dọc
\b	Ký tự xoá trái (backspace)
\r	Ký tự về đầu dòng (carriage return)
\f	Ký tự qua trang (formfeed)
\\"	Ký tự xô trái (backslash)
\'	Ký tự nháy đơn (single quote)
\"	Ký tự nháy kép (double quote)
\xddd	Ký tự ứng với mã ASCII dạng bát phân ddd
\udddd	Ký tự ứng với mã Unicode dddd dạng thập lục phân
\0	Ký tự null
\a	Ký tự chuông

Hằng kiểu logic có giá trị true, false. Hằng chuỗi diễn đạt một chuỗi ký tự kiểu string, được ghi trong cặp dấu nháy kép.

Ví dụ:

```
Console.WriteLine("Phương trình có 2 nghiệm:\n\tx1=" + x1 + "\n\tx2=" + x2);
```

C# phiên bản trước chỉ cho phép kiểu tham chiếu có giá trị null là tham chiếu rỗng. C# 2.0 bổ sung hằng null cho kiểu giá trị có kiểu Nullable nghĩa là không có giá trị.

2.3. BIỂU THỨC VÀ TOÁN TỬ

2.3.1. Biểu thức (Expression)

Biểu thức là một công thức tính toán một giá trị theo quy tắc toán học, cấu tạo từ các toán hạng (operand): hằng, phương thức, biến và nối kết với nhau bởi các toán tử (operator). Các toán hạng trong

mọi biểu thức phải tương thích với nhau về kiểu. Một biểu thức có thể là biểu thức số học, logic, ký tự, chuỗi khi kết quả của biểu thức có kiểu tương ứng.

Ví dụ: Biểu thức đơn giản nhất là biến hay hằng

22	a	
"Hello"	true	

2.3.2. Toán tử (Operator)

Toán tử là phép toán dùng để kết hợp các toán hạng, dùng trong các biểu thức.

2.3.2.1. Các toán tử số học (Arithmetic operator)

Phép toán số học sử dụng trên các toán hạng có kiểu số và trả về kết quả kiểu số:

Toán tử	Ý nghĩa	Cách dùng
+	Cộng hai toán hạng	op1 + op2
-	Trừ	op1 – op2
*	Nhân	op1 * op2
/	Chia	op1 / op2
%	Chia lấy số dư	op1 % op2

(op1, op2 là các toán hạng kiểu số)

Với phép chia op1/op2, nếu op1, op2 đều là số nguyên thì phép chia này cho kết quả là số nguyên, nếu một trong hai số op1, op2 hay cả hai số đều là số thực sẽ cho kết quả số thực.

Ví dụ:

```
using System;
class ArithmeticDemo
{
    static void Main(string[] args)
    {
        int i = 67, j = 22;
        double x = 27.475, y = 7.22;
```

```

        Console.WriteLine("\ti / j = " + i / j);
        Console.WriteLine("\ti + j = " + (i + j));
        Console.WriteLine("\tx - y = " + (x - y));
        Console.WriteLine("\tx / y = {0}, \n\ti % j = {1}", x / y, i % j);
    }
}

```

2.3.2.2. Các toán tử tăng, giảm (Increment, decrement operator)

Phép toán tăng giảm sử dụng cho các biến kiểu số nguyên và kiểu ký tự:

- op ++ hay ++ op: Tăng biến số nguyên lên một đơn vị, hay cho ký tự Unicode kế tiếp
- op -- hay -- op: Giảm biến số nguyên xuống một đơn vị, hay cho ký tự Unicode kế trước.

Toán tử tăng, giảm có thể đặt ở trước hay sau biến để diễn đạt ý nghĩa khác nhau.

Ví dụ:

```

int x = 4, y;
y = x++; //Giá trị x gán cho y trước, sau đó tăng lên một đơn vị, x=5, y=4
y = ++x; //Giá trị x tăng lên một đơn vị, sau đó gán cho y, x=6, y=6
x++;     //x = 7
Console.WriteLine(++x); //Tăng x lên một đơn vị sau đó in ra 8
Console.WriteLine(x--); //In ra 8 sau đó giảm x đi 1

```

Ví dụ:

```

char c = 'a';
c++;      //c='b'

```

2.3.2.3. Toán tử quan hệ hay so sánh (Comparison operator)

Phép toán so sánh áp dụng trên các kiểu dữ liệu cơ bản và luôn cho kết quả có kiểu logic.

Toán tử	Ý nghĩa	Cách dùng
<code>==</code>	Bằng	<code>op1 == op2</code>
<code>!=</code>	Không bằng	<code>op1 != op2</code>
<code><</code>	Nhỏ hơn	<code>op1 < op2</code>
<code>></code>	Lớn hơn	<code>op1 > op2</code>
<code><=</code>	Nhỏ hơn hoặc bằng	<code>op1 <= op2</code>
<code>>=</code>	Lớn hơn hoặc bằng	<code>op1 >= op2</code>

Ví dụ: Phép so sánh bằng có thể được sử dụng để so sánh số, ký tự hay chuỗi

`155 > 25`

`4.5 < 8`

`'B' == 'A'`

`'B' < 'A'`

2.3.2.4. Toán tử logic (Logical operator)

Phép toán logic chỉ áp dụng cho các toán hạng kiểu logic, cho kết quả kiểu logic

Toán tử	Ý nghĩa	Cách dùng
<code>& hay &&</code>	Và (And)	<code>op1 && op2</code>
<code> hay </code>	Hoặc (Or)	<code>op1 op2</code>
<code>^</code>	Hoặc loại trừ (Xor)	<code>op1 ^ op2</code>
<code>!</code>	Phủ định (Not)	<code>! op</code>

- Phép `&` hay `&&` cho kết quả true chỉ khi 2 toán hạng `op1`, `op2` có giá trị true
- Phép `|` hay `||` cho kết quả là false chỉ khi 2 toán hạng `op1`, `op2` đều có giá trị false
- Phép `^` cho kết quả là true chỉ khi 2 toán hạng `op1`, `op2` khác giá trị nhau
- Phép `!` cho kết quả là true nếu `op` có trị false, và ngược lại

Ví dụ:

```
100 > 55 && 'B' < 'A'  
Diem >= 5 && Diem < 7  
C == 'Y' || C == 'y'
```

Ví dụ:

```
x > 3 && x < ++y
```

Phép & khác với phép &&: với phép && chỉ cần một toán hạng bên trái: $x > 3$ trả về false, lập tức kết quả phép toán trả về false, C# không tính đến toán hạng bên phải $x < ++y$, nghĩa là y không được tăng lên một đơn vị.

Tương tự với phép | và phép ||, ở phép || chỉ cần một toán hạng bên trái trả về true, toán hạng bên phải sẽ được bỏ qua không tính đến.

2.3.2.5. Các toán tử làm việc với bit

Là các toán tử làm việc trên từng bit của số nguyên:

Toán tử	Ý nghĩa	Cách dùng
&	Và (And)	op1 & op2
	Hoặc (Or)	op1 op2
^	Hoặc loại trừ (Xor)	op1 ^ op2
<<	Dịch trái n bit	op << n
>>	Dịch phải n bit	op >> n
>>>	Dịch phải n bit và điền 0 vào những chỗ vừa dịch	op >>> n
~	Đảo bit (Not)	~ op

Ví dụ:

<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>1</td><td>0</td><td>1</td></tr> </table>	0	1	0	1	5
0	0	0	0							
0	1	0	1							
<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table>	0	0	0	1	5 >> 2 = 1
0	0	0	0							
0	0	0	1							

Phép dịch phải đi n bit cũng tương đương với phép chia cho 2^n

$5 >> 2 = 5 / 2^2 = 1$

$16 >> 3 = 16 / 2^3 = 2$

Khi dịch phải bít với số dương, bít bên trái sẽ điền 0, khi dịch phải bít với số âm, bít dấu bên trái vẫn giữ là 1

<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1	1	1	1	1	<table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>	1	1	0	0	1	1	1	0	-4
1	1	1	1															
1	1	1	1															
1	1	0	0															
1	1	1	0															
		-4 >> 1 = -2																

Phép dịch trái đi n bit cũng tương đương với phép nhân cho 2^n

<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	1	0	0	4
0	0	0	0							
0	1	0	0							
<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	1	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	4 << 2 = 16
0	0	0	1							
0	0	0	0							

$$4 << 2 = 4 * 2^2 = 16$$

Phép dịch phải không dấu đi n bit sẽ điền 0 vào các bít bên trái cho cả số dương và âm:

2 biểu diễn nhị phân là: 0000000 00000000 00000000 00000010

-2 biểu diễn nhị phân là : 11111111 11111111 11111111 11111110

-2 >>> 24 = 255: 00000000 00000000 00000000 11111111

2.3.2.6. Toán tử gán (Assignment operator)

Toán tử gán = dùng để gán giá trị của một biểu thức Exp cho một biến Var.

Var = Exp;

Ví dụ: x = 10;

Có thể dùng toán tử gán nhiều lần liên tiếp trong câu lệnh.

Ví dụ: x = y = z = 0;

C# còn cung cấp các toán tử rút gọn (Shorthand Assignment Operator) như sau:

Toán tử	Ý nghĩa		Cách dùng
<code>+=</code>	Cộng rồi gán	<code>var = var + Exp</code>	<code>var += Exp</code>
<code>-=</code>	Trừ rồi gán	<code>var = var - Exp</code>	<code>var -= Exp</code>
<code>*=</code>	Nhân rồi gán	<code>var = var * Exp</code>	<code>var *= Exp</code>
<code>/=</code>	Chia rồi gán	<code>var = var / Exp</code>	<code>var /= Exp</code>

%=	Chia lấy phần dư rồi gán var = var % Exp	var %= Exp
&=	And từng bit rồi gán var = var & Exp	var &= Exp
=	Or từng bit rồi gán var = Exp	var = Exp
^=	Xor từng bit rồi gán var ^= Exp	var ^= Exp
<<=	Dịch trái rồi gán var <<= Exp	var <<= Exp
>>=	Dịch phải rồi gán var >>= Exp	var >>= Exp
>>>=	Dịch phải, điền 0 vào những chỗ vừa dịch rồi gán var >>>= Exp	var >>>= Exp

Ví dụ:

```
int a = 2; a += 4;           //Tương đương với a = a + 4;
char c = 'A';
Console.WriteLine(c + 4);    //in ra 69
Console.WriteLine((char)(c+4)); //in ra ký tự E
```

2.3.2.7. Một số toán tử khác

Toán tử	Cách dùng	Ý nghĩa
?:	op1 ? op2 : op3	Phép toán điều kiện: nếu toán hạng logic op1 có trị true thì trả về giá trị op2. Ngược lại trả về giá trị op3
[]	op1[op2]	Truy cập đến một phần tử của mảng op1 với chỉ số op2
.	op1.op2	Truy cập thành viên biến và phương thức op2 của đối tượng, struct hay lớp op1
()	op(params)	Định nghĩa hoặc gọi một phương thức có tên là op với các tham đối là params
(type)	(type) op	Chuyển đổi kiểu dữ liệu của toán hạng op sang kiểu type
new		Khởi tạo một đối tượng hay một mảng
is	op1 is op2	Phép toán trả về true nếu đối tượng op1 thuộc kiểu op2
+	op1 + op2 + ...	Nối các chuỗi, nếu có toán hạng nào không phải chuỗi, C# tự động chuyển sang chuỗi

Ví dụ:

```
int a = 2, b = 5;
Console.WriteLine(a>b ? a : b);
```

Ví dụ:

```
Console.WriteLine(" 2 + 2 = " + (2 + 2));
```

2.3.3. Các quy tắc thực hiện phép toán và chuyển kiểu

2.3.3.1. Quy tắc thứ tự ưu tiên (Precedence order rule)

Độ ưu tiên của các phép toán trong biểu thức thực hiện theo thứ tự từ trên xuống như trong bảng sau. Có thể chủ động quy định thứ tự thực hiện phép toán trong biểu thức bằng những cặp ngoặc tròn (), trong ngoặc sẽ thực hiện trước, và ngoặc trong thực hiện trước, ngoặc ngoài thực hiện sau. Các phép toán trên cùng hàng sẽ có cùng độ ưu tiên.

Bảng 2.4. Bảng quy tắc thứ tự ưu tiên

Toán tử	Ghi chú	Quy tắc kết hợp
(), [], .	Dấu . để truy xuất biến, phương thức của đối tượng, lớp, [] truy xuất phần tử của mảng	Trái sang phải
++, --, +op, -op, !, ~	Dấu +op, -op là dấu dương, âm, ! là phép phủ định logic, ~ là đảo bít	Phải sang trái
new, (type) op	(type) op là phép chuyển đổi kiểu	Phải sang trái
*, /, %	Các phép toán số học	Trái sang phải
+, -		Trái sang phải
<<, >>, >>>	Phép toán dịch chuyển bit	Trái sang phải
<, >, <=, >=, is	Các phép toán quan hệ	Trái sang phải
==, !=		Trái sang phải
&	Các phép toán trên từng bit	Trái sang phải
^		Trái sang phải

		Trái sang phải
&&	Các phép toán logic	Trái sang phải
		Trái sang phải
? :	Phép toán điều kiện	Phải sang trái
=, +=, -=, *=, /=, %=, ^=, &=, =, <<=, >>=, >>>=	Các phép toán gán	Phải sang trái

2.3.3.2. Quy tắc kết hợp (Associativity rule)

Quy tắc kết hợp thực hiện trên các phép toán có cùng độ ưu tiên theo thứ tự từ trái qua phải hay từ phải qua trái như trong bảng 2.4

Ví dụ:

```
int x = a + b - c;           //Thứ tự thực hiện phép toán + - từ trái qua phải
s += a += b += c;           //Thứ tự thực hiện phép gán từ phải sang trái
```

2.3.3.3. Quy tắc nâng cấp toán hạng (Operand promotion rule)

C# tự động nâng cấp kiểu cơ sở nào đó thành kiểu cơ sở khác lớn hơn, khi có nhu cầu. Chẳng hạn, nâng cấp kiểu char thành int, short hoặc int thành long, nâng cấp kiểu float thành double, hay kiểu nguyên thành kiểu thực. Các trường hợp chuyển đổi kiểu như vậy không làm mất thông tin.

Ví dụ:

```
short x = 10;
int y = x;
char c = 'A';                //ký tự 'A' có mã unicode 65
Console.WriteLine(c+1); //in ra mã unicode 66
C# tự động nâng cấp kiểu char của biến c thành int
```

Trường hợp chuyển đổi các số thực (float, double, decimal) thành số nguyên (byte, sbyte, short, ushort, int, uint, long, ulong), và việc chuyển đổi các kiểu lớn thành các kiểu nhỏ hơn như long thành int, kết quả tính toán có thể thiếu chính xác, do đó C# không tự động thực hiện.

Ví dụ:

```
byte a, b = 200, c = 100;
a = b + c;
```

Lỗi do kiểu byte của b, c nâng cấp thành int, vì vậy không thể gán giá trị kiểu int cho biến kiểu byte.

2.3.3.4. Quy tắc chuyển kiểu (Casting rule)

Nếu bạn muốn chuyển đổi giữa các kiểu dữ liệu, sử dụng cú pháp như sau:

(KiểuMới) Value

Ví dụ:

```
byte a, b = 200, c = 100;
a = (byte)(b+c);
```

Không lỗi nhưng có thể mất mát thông tin, do kiểu byte: 0-255

```
char c = 'A';
```

```
Console.WriteLine((char)(c+32)) //in ra ký tự 'a'
```

```
int a=5, b=2;
```

```
Console.WriteLine((double)a / b); //in ra 2.5
```

Có thể chuyển giữa 2 kiểu tham chiếu, chẳng hạn ta có một giao tiếp IHinhPhang, và lớp HinChuNhat là lớp cài đặt giao tiếp IHinhPhang:

```
IHinhPhang HCN = new HinChuNhat(2, 3);
Console.WriteLine(((HinChuNhat)HCN).Canh2);
```

2.4. LỆNH (STATEMENT)

Lệnh có thể là lệnh đơn, khối lệnh hay cấu trúc điều khiển.

2.4.1. Lệnh đơn (Simple statement)

Lệnh đơn là thành phần cơ bản nhất trong chương trình C#, diễn đạt một thao tác riêng lẻ nào đó phải làm, được kết thúc bằng dấu chấm phẩy.

Lệnh đơn có thể là lời gọi phương thức, phép gán, phép tăng, giảm giá trị biến, các lệnh định nghĩa, khai báo.

Ví dụ:

```
int i =1;
Console.WriteLine("Đây là một hằng chuỗi");
x++;
```

2.4.2. Khối lệnh (Compound statement hay block)

Nhiều câu lệnh đơn có thể nhóm lại thành một câu lệnh phức hợp gọi là khối lệnh. Khối lệnh được mở đầu bằng dấu { và kết thúc bằng dấu }, có thể đặt trong một khối lệnh khác.

2.4.3. Cấu trúc rẽ nhánh có điều kiện (Conditional structure)

2.4.3.1. Cấu trúc if

if là câu lệnh lựa chọn cho phép chương trình rẽ nhánh thực hiện lệnh theo hai hướng khác nhau căn cứ trên giá trị true, false của biểu thức điều kiện kiểu logic.

Cú pháp:

Dạng 1: **if** (BiểuThứcĐiềuKiện)

Lệnh;

Dạng 2: **if** (BiểuThứcĐiềuKiện)

Lệnh1;

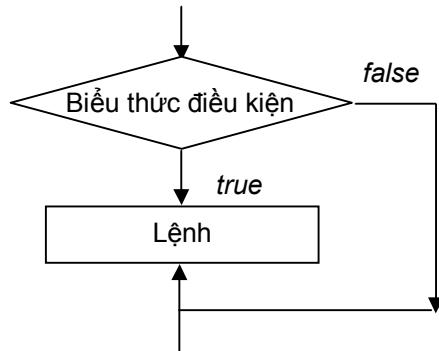
else Lệnh2;

- Biểu thức điều kiện (Conditional expression) có kiểu logic bool.
- Lệnh (Statement) có thể là lệnh đơn, khối lệnh, hay cấu trúc điều khiển

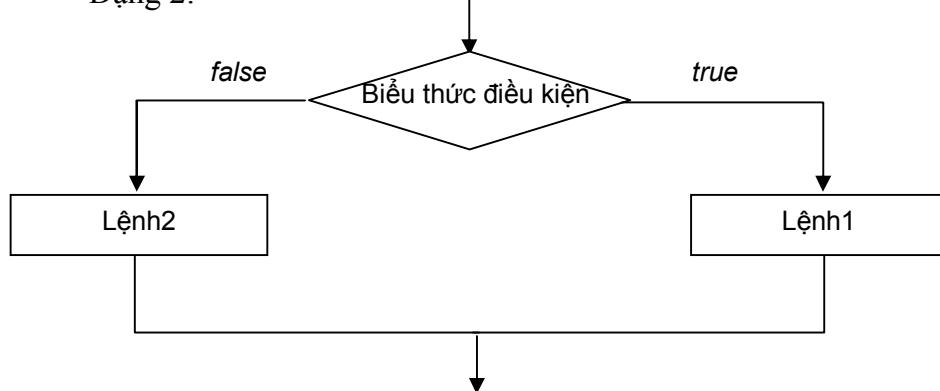
Ý nghĩa: Nếu biểu thức điều kiện thỏa mãn (có giá trị true) thì lệnh 1 thực hiện, ngược lại nếu biểu thức điều kiện không thỏa mãn (có giá trị false) thì không làm gì cả (với dạng 1) hay thực hiện lệnh 2 (với dạng 2).

Sơ đồ khối:

Dạng 1:



Dạng 2:



Ví dụ: Giải phương trình bậc 2 : $ax^2 + bx + c = 0$

```

using System;
class PTB2
{
    static void Main(string[] args)
    {
        Console.Write("Nhập hệ số a:");
        string s=Console.ReadLine(); double a=double.Parse(s);
        Console.Write("Nhập hệ số b:");
        s = Console.ReadLine(); double b=double.Parse(s);
        Console.Write("Nhập hệ số c:");
        s = Console.ReadLine(); double c=double.Parse(s);
        if (a == 0)
  
```

```

        if (b == 0)
            if (c == 0) s="co vo so nghiem";
            else s = "vo nghiem";
            else s = "co mot nghiem la "+(-c/b);
        else
    {
        double delta=b*b-4*a*c;
        if (delta<0) s = "vo nghiem";
        else if (delta==0)
            s = "co nghiem kep la "+(-b/(2*a));
        else s= "co 2 nghiem
            x1= " + ((-b+ Math.Sqrt(delta))/(2*a)) + " va x2= "+((-b-Math.Sqrt(delta))/(2*a));
    }
    Console.WriteLine("Phuong trinh " + s);
}
}

```

2.4.3.2. Cấu trúc switch

switch là câu lệnh lựa chọn, cho phép rẽ nhánh thực hiện lệnh theo nhiều hướng khác nhau căn cứ trên giá trị của một biểu thức.

Cú pháp:

```

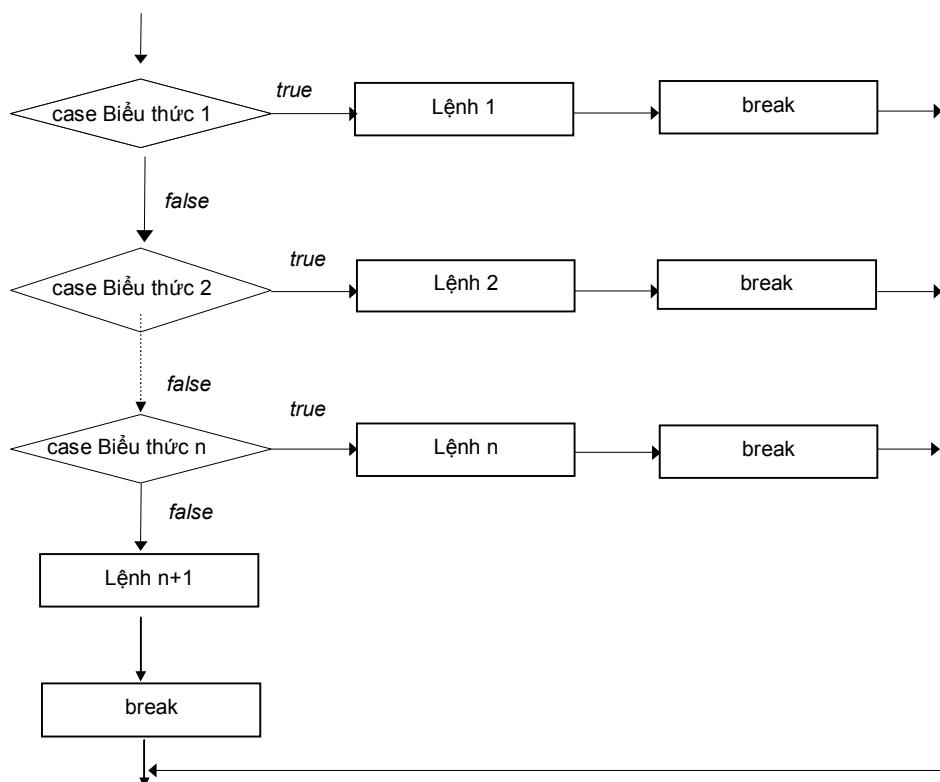
switch (BiểuThứcĐiềuKiện)
{
    case Biểu thức 1:
        Lệnh 1;
        break;
    ...
    case Biểu thức n:
        Lệnh n;
        break;
    default:
        Lệnh n + 1;
        break;
}

```

- Biểu thức điều kiện (Conditional expression): Có kiểu nguyên hay chuỗi
- Lệnh 1,... Lệnh n+1: Có thể là nhiều lệnh đơn, hay cấu trúc điều khiển
- Biểu thức 1,... Biểu thức n: Các biểu thức hằng (toán hạng là các giá trị hằng)

Ý nghĩa: Thực hiện lệnh 1, lệnh 2, ... lệnh n khi biểu thức có giá trị bằng biểu thức 1, biểu thức 2, ... biểu thức n tương ứng, nếu biểu thức không bằng các giá trị đó thì lệnh n+1 theo sau default được thực hiện (nếu có default). Câu lệnh break làm cho chương trình có thể thoát khỏi lệnh switch, nếu không có break, C# sẽ thông báo lỗi, nhưng đặc biệt C# cho phép case rỗng.

Sơ đồ khối:



Ví dụ: Nhập tháng, năm và in ra số ngày trong tháng, biết rằng tháng 1, 3, 5, 7, 8, 10, 12: số ngày trong tháng là 31, tháng 4, 6, 9, 11: số ngày trong tháng là 30, tháng 2: năm nhuận: số ngày là 29, năm không nhuận: số ngày là 28. Năm nhuận là năm chia hết cho 400, hay năm chia hết cho 4 và không chia hết cho 100.

```
using System;
class NumberDay
{
    static void Main(string[] args)
    {
        int month = 7, year = 2004, numDays = 0;
        switch (month)
        {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12: numDays = 31; break;
            case 4:
            case 6:
            case 9:
            case 11: numDays = 30; break;
            case 2: if (year % 4==0 && year % 100 != 0 || year %
                      400==0)
                      numDays = 29;
                    else numDays = 28;
                    break;
        }
        if (numDays >0)
            Console.WriteLine("So ngay trong thang la {0}", numDays);
        else Console.WriteLine("Thang khong hop le");
    }
}
```

2.4.4. Cấu trúc lặp (Repeat structure hay loop)

2.4.4.1. Cấu trúc for

Ngoài for, while và do... while, C# còn bổ sung thêm cấu trúc lặp foreach.

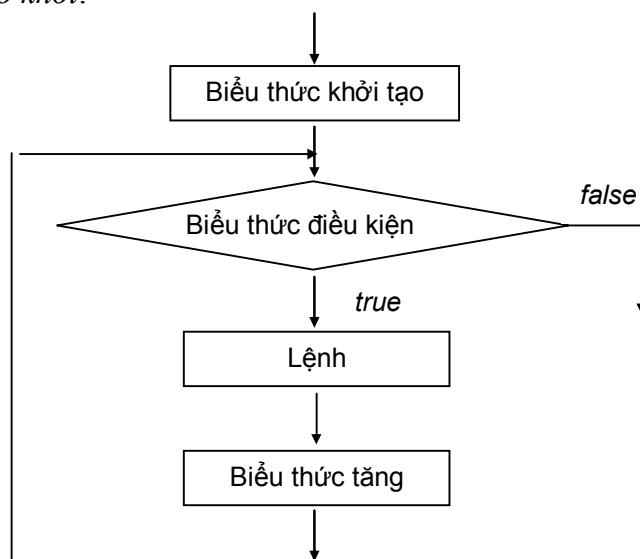
Cú pháp:

for (Biểu Thức Khởi Tạo; Biểu Thức Điều Kiện; Biểu Thức Tăng)

Lệnh;

- Lệnh (Statement) có thể là lệnh đơn, khối lệnh, hay cấu trúc điều khiển.
- Biểu thức khởi tạo (Initialization expressions): là các biểu thức khởi tạo giá trị cho các biến điều khiển vòng lặp, cách nhau dấu phẩy.
- Biểu thức điều kiện (Boolean expression): có kiểu logic bool, thường là biểu thức so sánh giá trị của biến điều khiển với giá trị kết thúc vòng lặp.
- Biểu thức tăng (Increment expressions): là các biểu thức tăng giảm giá trị biến điều khiển vòng lặp, cách nhau dấu phẩy.

Sơ đồ khối:



Ý nghĩa: Khi lệnh lặp for bắt đầu, biểu thức khởi tạo được thực hiện trước, và thực hiện duy nhất một lần, sau đó biểu thức điều kiện được kiểm tra, nếu biểu thức đúng thì lệnh sẽ được thực hiện, tiếp theo thực hiện biểu thức tăng của vòng lặp và quá trình kiểm tra biểu thức điều kiện, thực hiện lệnh, và thực hiện biểu thức tăng sẽ lặp đi lặp lại cho đến khi biểu thức điều kiện sai.

Ví dụ:

```

“Trăm trâu trăm cỗ
Trâu dung ăn năm
Trâu năm ăn ba
Ba trâu già ăn một”

Hỏi số trâu mỗi loại?

using System;
class SoTrau
{
    static void Main(string[] args)
    {
        Console.WriteLine("\tTrau dung\tTrau nam\tTrau gia");
        int td = 100 / 5;
        for (int d = 1; d <= td; d++)
        {
            int tn = (100 - d * 5) / 3;
            for (int n = 1; n <= tn; n++)
            {
                int g = 100 - d - n;
                if (d * 5 + n * 3 + (double)g / 3 == 100)
                    Console.WriteLine("\t"+d+"\t\t"+ n+"\t\t"+ g);
            }
        }
    }
}

```

2.4.4.2. Cấu trúc while

Cú pháp:

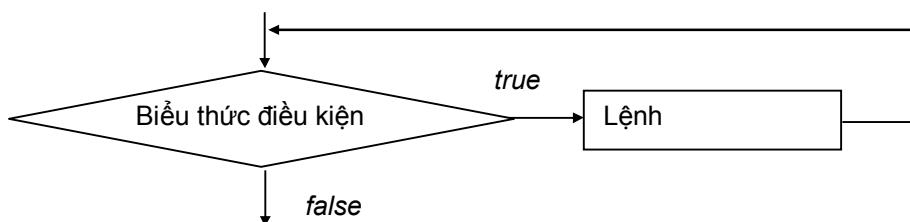
while (BiểuThứcĐiềuKiện)

Lệnh;

- Lệnh (Statement): có thể là lệnh đơn, khối lệnh, hay cấu trúc điều khiển
- Biểu thức điều kiện (Conditional expression): có kiểu logic

Ý nghĩa: Kiểm tra chừng nào biểu thức điều kiện còn thoả mãn thì còn thực hiện lệnh.

Sơ đồ khối:



Ví dụ: Tìm ước số chung lớn nhất của 2 số x, y

Phân tích: nếu $x = y$ thì USCLN của x và y là x,

nếu $x > y$ thì USCLN của x, y cũng là USCLN của $x-y$ và y.

Do đó ta thay số lớn trong hai số x, y bằng hiệu số của chúng cho đến khi hai số bằng nhau, thì đó là ước số chung lớn nhất của hai số x, y

Ví dụ:

```

using System;
class Program
{
    static void Main(string[] args)
    {
        int x = 12, y = 6;
        Console.WriteLine("USCLN cua {0} va {1} la : ", x, y);
        while (x!=y)
    }
  
```

```

        if (x>y) x = x - y;
        else y = y - x;
        Console.WriteLine(x);
    }
}

```

2.4.4.3. Cấu trúc do... while

Cú pháp:

do

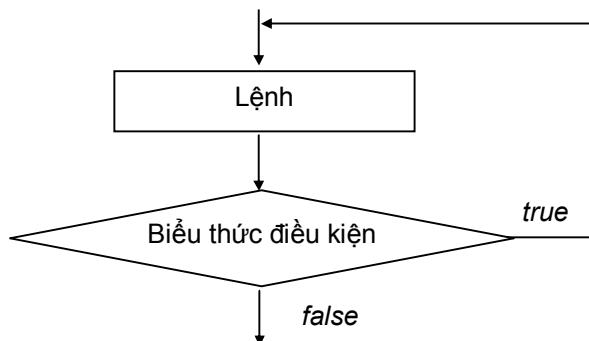
Lệnh;

while (BiểuThứcĐiềuKiện);

- Lệnh (Statement): Có thể là lệnh đơn, khối lệnh, hay cấu trúc điều khiển
- Biểu thức điều kiện (Conditional expression): Có kiểu logic

Ý nghĩa: Đầu tiên sẽ thực hiện lệnh, sau đó kiểm tra biểu thức điều kiện, nếu biểu thức đúng thì vòng lặp tiếp tục, nếu biểu thức sai thì thoát vòng lặp.

Sơ đồ khối:



Ví dụ:

```

using System;
namespace USCLN
{
    class Program

```

```

{
    static void Main(string[] args)
    {
        char f = 'Y';
        do
        {
            Console.Write("Nhập số nguyên thu nhất:");
            int x = int.Parse(Console.ReadLine());
            Console.Write("Nhập số nguyên thu hai:");
            int y = int.Parse(Console.ReadLine());
            Console.WriteLine("USCLN của {0} và {1} là : ",x,y);
            while (y != 0)
            {
                int SoDu = x % y;
                x=y;
                y=SoDu;
            }
            Console.WriteLine(x);
            Console.Write("Bạn có muốn tiếp tục không(Y/N)?");
            f = char.Parse(Console.ReadLine());
        }
        while (f == 'Y' || f == 'y');
    }
}

```

Ví dụ: Cho biết số nguyên dương n có phải là số nguyên tố không?

Phân tích: n là số nguyên tố khi n chia không chẵn cho bất kỳ số nào từ 2 đến căn bậc hai của n, vì vậy lần lượt xét n có chia hết cho các số từ 2 đến căn bậc hai của n, cho đến khi xét hết hay tìm được một số mà n chia hết.

```

using System;
namespace NguyenTo
{
    class Program
    {

```

```

static void Main(string[] args)
{
    Console.Write("Nhập số nguyên:");
    int n = int.Parse(Console.ReadLine());
    double k = Math.Sqrt(n);
    int i = 1;
    do
        i++;
    while (i <= k && n % i != 0);
    if (i > k) Console.WriteLine(n + " là số nguyên tố");
    else Console.WriteLine("{0} không là số nguyên tố", n);
}
}
}

```

2.4.4.4. Cấu trúc foreach

Cấu trúc foreach cho phép tạo vòng lặp duyệt qua tất cả phần tử của một danh sách (collection) hay một mảng. Đây là một câu lệnh lặp mới không có trong ngôn ngữ C/C++. Câu lệnh foreach có cú pháp chung như sau:

foreach (KiểuPhầnTử TênPhầnTử **in** TênTậpHợp)

Lệnh;

- Lệnh (Statement): Có thể là lệnh đơn, khối lệnh, hay cấu trúc điều khiển
- Kiểu phần tử (Item type): Là kiểu phần tử trong danh sách
- Tên phần tử (Item name): Là tên biến dùng để truy cập phần tử
- Tên danh sách (Collection name): Là tên biến danh sách

Ví dụ:

```

using System;
class Program
{
    static void Main(string[] args)
    {

```

```

string[] nhom = {"Lan", "Thu", "Hoa", "Xuan"};
foreach(string ten in nhom)
    Console.WriteLine("{0} ", ten);
}
}

```

2.4.5. Các lệnh điều khiển rẽ nhánh không điều kiện

2.4.5.1. Lệnh break

Cú pháp:

break;

hay **break Nhãn;**

Ý nghĩa: Lệnh break dùng để thoát tức thời ra khỏi vòng lặp while, do... while, for hay lệnh rẽ nhánh switch chứa nó và thực hiện lệnh tiếp theo.

- Lệnh break Nhãn; dùng vòng lặp tức thời và quay về nhãn (Label). Tên nhãn đặt theo quy ước đặt tên của C#, ghi trong chương trình có dạng:

Nhãn:

Lệnh;

2.4.5.2. Lệnh continue

Cú pháp:

continue;

hay **continue Nhãn;**

Ý nghĩa: Lệnh continue để bỏ qua các lệnh sau continue và quay trở về đầu vòng lặp chứa nó. Lệnh continue Nhãn bỏ qua các lệnh sau continue và quay về nhãn.

2.5. NGOẠI LỆ (EXCEPTION) VÀ XỬ LÝ NGOẠI LỆ

Ngoại lệ trong C# là các đối tượng có kiểu lớp định nghĩa sẵn, biểu diễn trạng thái lỗi phát sinh trong trường hợp nào đó khi gọi phương thức hay trong trường hợp chia cho 0, cảnh báo bộ nhớ

thấp... Một số phương thức định nghĩa sẵn trong trường hợp nào đó thì ngoại lệ sẽ phát sinh.

Khi một ngoại lệ phát sinh, phải bắt ngoại lệ bởi lệnh try... catch, nếu không chương trình sẽ kết thúc thực hiện. Lệnh xử lý ngoại lệ thực hiện thông qua các từ khoá: try, catch, finally và throw.

2.5.1. Lệnh try... catch... finally

Cú pháp:

```

try {
    Lệnh;
}

catch (KiểuNgoạiLệ1 [ĐốiTượngNgoạiLệ 1])
{
    Lệnh 1;
}

...
catch (KiểuNgoạiLệ n [ĐốiTượngNgoạiLệ n])
{
    Lệnh n;
}

finally
{
    Lệnh n+1;
}

```

- Lệnh: Có thể là một hay nhiều lệnh đơn, hay cấu trúc điều khiển
- Đối tượng ngoại lệ (Exception instance): Là tên của đối tượng kiểu lớp ngoại lệ, có thể có hoặc không
- Kiểu ngoại lệ (Exception type) : Là kiểu lớp ngoại lệ

Ý nghĩa:

- Try: Định nghĩa một khối lệnh mà ngoại lệ có thể xảy ra.
- catch: Đi kèm với try, để bắt ngoại lệ. Những câu lệnh trong chương trình mà bạn muốn bắt ngoại lệ được đưa vào giữa khối try, nếu một ngoại lệ xảy ra trong khối try, các lệnh còn lại trong khối try sẽ được bỏ qua và thân của mệnh đề catch có kiểu ngoại lệ tương ứng sẽ được thực hiện. Có thể có nhiều mệnh đề catch bắt các kiểu ngoại lệ khác nhau.
- finally: Thân của mệnh đề finally luôn luôn thực hiện ngay trước khi lệnh try kết thúc, cho dù có hay không có ngoại lệ, mệnh đề finally có thể có hay không.

Ví dụ:

```
using System;
class Test
{
    static void Main ()
    {
        int [] arr = new int [5];
        for (int x = 0; x < arr.Length; ++x)
        {
            while (true)
            {
                Console.Write ("Enter a value for element {0}: ", x);
                string str = Console.ReadLine ();
                int val;
                try {
                    val = int.Parse (str);
                }
                catch (FormatException) {
                    Console.WriteLine ("Enter an integer value\r\n");
                    continue;
                }
                arr[x] = val; break;
            }
        }
    }
}
```

```

    }
    int index = 0;
    foreach (int val in arr)
    {
        Console.WriteLine ("arr[{0}] = {1}", index, val);
        ++index;
    }
}

```

2.5.2. Lệnh throw

Cú pháp:

throw Đối tượng ngoại lệ;

- Đối tượng ngoại lệ (Exception instance): Là một đối tượng của kiểu ngoại lệ System.Exception, hay những đối tượng được dẫn xuất từ kiểu dữ liệu này. Namespace System chứa một số các kiểu dữ liệu xử lý ngoại lệ mà chúng ta có thể sử dụng trong chương trình như ArgumentNullException, InvalidCastException, OverflowException...
- Tạo một đối tượng kiểu ngoại lệ bằng toán tử new

Ví dụ: throw new System.Exception();

Ý nghĩa: Câu lệnh throw được đưa vào khôi try, cho phép bạn điều khiển điều kiện phát sinh ngoại lệ. Khi gặp lệnh throw, các câu lệnh sau throw trong khôi try sẽ được bỏ qua, và khôi lệnh của mệnh đề catch có kiểu ngoại lệ tương ứng sẽ được thực hiện.

Ví dụ: Kiểm tra và bắt lỗi nếu phép chia cho 0.

```

using System;
class Test
{
    public static void Main()
    {
        Test t = new Test();
        t.TestFunc();
    }
}
```

```
    }
    public void TestFunc()
    {
        try {
            double a = 5;
            double b = 2;
            Console.WriteLine("{0} / {1} = {2}", a, b, DoDivide(a,b));
        }
        catch (System.DivideByZeroException) {
            Console.WriteLine("Divide By Zero Exception!");
        }
    }
    public double DoDivide(double a, double b)
    {
        if (b == 0) throw new System.DivideByZeroException();
        return a / b;
    }
}
```

2.5.3. Các lớp ngoại lệ

Exception là lớp cha của tất cả các ngoại lệ, cung cấp một số các phương thức và thuộc tính:

- Thuộc tính Message trả về thông tin phát sinh ngoại lệ.
- Thuộc tính HelpLink cung cấp một liên kết để trợ giúp cho các tập tin liên quan đến các ngoại lệ.
- Thuộc tính StackTrace cung cấp thông tin về câu lệnh lỗi.

Có hai kiểu ngoại lệ:

- Ngoại lệ phát sinh bởi ứng dụng dẫn xuất từ lớp ApplicationException
- Ngoại lệ phát sinh bởi hệ thống dẫn xuất từ lớp SystemException.

Hai lớp này kế thừa trực tiếp từ lớp Exception. Lớp ngoại lệ bạn định nghĩa nên dẫn xuất từ lớp ApplicationException.

Cây phân cấp các ngoại lệ xây dựng sẵn kế thừa từ lớp SystemException

```

Object
  Exception
    ApplicationException
    SystemException
      IndexOutOfRangeException
      ArrayTypeMismatchException
      NullReferenceException
      InvalidOperationException
      ArgumentException
        ArgumentNullException
        ArgumentOutOfRangeException
      MemberAccessException
        MethodAccessException
      ArithmeticException
        DivideByZeroException
        NotFiniteNumberException
      FormatException
      InvalidCastException
      OutOfMemoryException
      StackOverflowException
      NotSupportedException
  
```

Bảng 2.5 Các kiểu ngoại lệ

Kiểu ngoại lệ	Mô tả	Ví dụ
Exception	Lớp cha của tất cả ngoại lệ	
SystemException	Lớp cha của tất cả các ngoại lệ xây dựng sẵn	
IndexOutOfRangeException	Phát sinh khi chỉ số truy cập phần tử mảng không hợp lệ	arr[arr.Length+1]

ArrayTypeMismatchException	Phát sinh khi kiểu thành phần mảng không đúng	
NullReferenceException	Phát sinh khi truy cập thành viên của đối tượng null	object o = null; o.ToString();
InvalidOperationException	Phát sinh bởi phương thức khi ở trạng thái không đúng	int? a = default(int?); int b = a.Value; a kiểu Nullable không chứa bất kỳ giá trị
ArgumentException	Lớp cha của tất cả ngoại lệ tham đối	
ArgumentNullException	Phát sinh bởi phương thức không cho phép một tham đối là null	String s = null; "Calculate".IndexOf(s);
ArgumentOutOfRangeException	Phát sinh bởi phương thức kiểm tra các tham đối trong phạm vi đã cho	String s = string"; s.Chars[9];
MemberAccessException	Lớp cha của tất cả ngoại lệ phát sinh khi truy cập thành viên	
MethodAccessException	Phát sinh khi truy cập đến phương thức không được truy cập	
ArithmeticeException	Phát sinh khi có lỗi liên quan đến các phép toán	
DivideByZeroException	Phát sinh khi chia cho 0	
NotFiniteNumberException	Phát sinh khi số 0 là hữu hạn, không hợp lệ	

FormatException	Phát sinh do định dạng không đúng	
InvalidCastException	Phát sinh khi phép gán không hợp lệ	
OutOfMemoryException	Phát sinh khi đầy bộ nhớ	
StackOverflowException	Phát sinh khi tràn stack	
NotSupportedException	Phát sinh khi gọi phương thức không tồn tại trong lớp	

2.6. LỚP SYSTEM.CONSOLE

Lớp Console bao gồm các phương thức nhập, xuất và định dạng kết xuất, các luồng xử lý lỗi cho các ứng dụng dựa trên console. Lớp Console trong phiên bản .NET 2.0 đã được cải tiến thêm các đặc tính mới.

2.6.1. Định dạng kết xuất

Lớp Console cung cấp một số đặc tính định dạng kết xuất sau:

- BackgroundColor, ForegroundColor: hai thuộc tính này cho phép bạn thiết lập màu nền và màu chữ cho kết xuất ra màn hình.
- BufferHeight, BufferWidth: hai thuộc tính này điều chỉnh chiều cao và chiều rộng cho buffer của console.
- Clear(): phương thức xóa buffer và vùng hiển thị của console.
- WindowHeight, WindowWidth, WindowTop, WindowLeft: các thuộc tính này điều khiển các chiều cao, rộng, lề trên, lề trái của console so với buffer.

Ví dụ:

```
using System;
class Program
```

```
{  
    static void Main(string[] args)  
    {  
        ConfigureCUI();  
    }  
    private static void ConfigureCUI()  
    {  
        Console.Title = "My Application";  
        Console.ForegroundColor = ConsoleColor.Yellow;  
        Console.BackgroundColor = ConsoleColor.Blue;  
        Console.WriteLine("*****");  
        Console.WriteLine("**** Welcome to My Application ****");  
        Console.WriteLine("*****");  
    }  
}
```

2.6.2. Nhập và xuất với lớp Console

Lớp Console định nghĩa một tập các phương thức tĩnh để thực hiện việc nhập dữ liệu từ bàn phím và xuất ra màn hình.

- Phương thức Write() xuất một giá trị hay đối tượng ra màn hình. Nếu xuất đối tượng, phương thức ToString() của đối tượng sẽ chuyển đổi thành chuỗi và xuất ra màn hình.
- Phương thức WriteLine() xuất một giá trị hay đối tượng ra màn hình, sau đó xuống dòng.
- Phương thức string ReadLine() trả về chuỗi nhập từ bàn phím, kết thúc bởi phím xuống dòng.
- Phương thức int Read() trả về ký tự nhập từ bàn phím.

Ví dụ: Viết chương trình nhập vào họ tên, và năm sinh với điều kiện họ tên không quá 25 ký tự, và năm sinh từ 1980 đến 1985.

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

```

        ConfigureCUI();
        Console.ReadLine();
    }
    private static void ConfigureCUI()
    {
        string s = null;
        do {
            Console.Write("Nhập Họ và tên không quá 25 ký tự: ");
            s = Console.ReadLine();
        }
        while (s.Length>25 || s.Length==0);
        Console.WriteLine("Họ và tên là : {0}",s);
        //Nhập năm sinh
        while (true)    {
            try {
                Console.Write("Nhập năm sinh : ");
                s = Console.ReadLine();
                int ns = int.Parse(s);
                if (ns<1980 || ns>1985) throw new FormatException();
                Console.WriteLine("Năm sinh là : "+ns);
                break;
            }
            catch (FormatException e1) {
                Console.WriteLine("Nhập năm sinh từ 1980 -1985");
            }
        }
    }
}

```

2.6.3. Định dạng và thiết lập vị trí kết xuất cho phương thức Write

Trong các ví dụ, bạn thường xuyên thấy sự xuất hiện của các ký tự đánh dấu {0}, {1}..., được chèn trong chuỗi kết xuất. .Net giới thiệu một cách định dạng chuỗi mới, cũng khá giống so với hàm printf() của C, nhưng không còn các cờ %d, %s hay %c nữa.

Bạn có thể thiết lập vị trí kết xuất cho phương thức Write trên màn hình bằng cách thiết lập giá trị cho thuộc tính CursorLeft và CursorTop hay phương thức tĩnh SetCursorPosition của lớp Console.

Ví dụ:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int theInt = 90;
        double theDouble = 9.99;
        bool theBool = true;
        Console.CursorLeft = 10;
        Console.CursorTop = 5;
        //Console.SetCursorPosition(10,5);
        Console.WriteLine("Int is: {0}\nDouble is: {1}\nBool is: {2}",
            theInt, theDouble, theBool);
    }
}
```

Tham đối đầu tiên của phương thức WriteLine() là một chuỗi và chứa các ký hiệu đánh dấu {0}, {1}, {2}... (số trong ngoặc được bắt đầu từ số không). Và các tham đối còn lại của WriteLine() chỉ đơn giản là các giá trị được thêm vào lần lượt tương ứng với các ký hiệu đánh dấu (trong trường hợp này là một int, một double và một bool).

Phương thức WriteLine() cũng cho phép bạn truyền vào một mảng các đối tượng, giống như sau:

```
object[] stuff = {"Hello", 20.9, 1, "There", "83", 99.99933};
Console.WriteLine("The Stuff: {0} , {1} , {2} , {3} , {4} , {5} ",
    stuff);
```

Nó cũng cho phép bạn đặt cùng một đánh dấu ở nhiều nơi:

```
Console.WriteLine("{0}, Number {0}, Number {0}", 9);
```

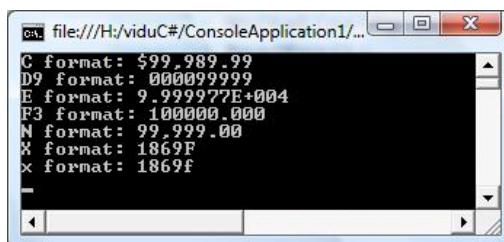
Nếu bạn truyền giá trị không khớp với các đánh dấu, sẽ phát sinh ngoại lệ FormatException

Nếu bạn muốn định dạng thêm nhiều kiểu khác nữa, có thể dùng các cờ sau (viết hoa hay thường đều được), được viết sau dấu hai chấm sau ký hiệu đánh dấu ({0:C}, {1:d}, {2:X},...):

- C hay c: Sử dụng để định dạng tiền tệ.
- D hay d: Sử dụng để định dạng số thập phân.
- E hay e: Sử dụng để thể hiện dạng số mũ.
- F hay f: Sử dụng cho định dạng dấu chấm tĩnh.
- G hay g: Viết tắt của general. Ký tự này dùng để định dạng kiểu chấm tĩnh hay số mũ.
- N hay n: Sử dụng định dạng phần ngàn (với dấu phẩy)
- X hay x: Sử dụng định dạng thập lục phân. Nếu bạn dùng X viết hoa thì ký tự thập lục cũng sẽ được viết hoa.

Ví dụ:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("C format: {0:C}", 99989.987);
        Console.WriteLine("D9 format: {0:D9}", 99999);
        Console.WriteLine("E format: {0:E}", 99999.76543);
        Console.WriteLine("F3 format: {0:F3}", 99999.9999);
        Console.WriteLine("N format: {0:N}", 99999);
        Console.WriteLine("X format: {0:X}", 99999);
        Console.WriteLine("x format: {0:x}", 99999);
    }
}
```



Hoặc có thể sử dụng phương thức tĩnh string.Format() để định dạng chuỗi

Ví dụ:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        string formatStr;
        formatStr = string.Format("Don't you wish you had {0:C} in your
account?", 99989.987);
        Console.WriteLine(formatStr);
    }
}
```

CÂU HỎI CHƯƠNG 2

Câu 1. Sự khác nhau giữa hằng định danh và hằng giá trị?

Câu 2. Sự khác nhau giữa kiểu dữ liệu giá trị và kiểu dữ liệu tham chiếu?

Câu 3. Các kiểu dữ liệu giá trị trong C#?

Câu 4. Các kiểu dữ liệu tham chiếu trong C#?

Câu 5. So sánh khai báo hằng, biến của C# với C++ và Java?

Câu 6. So sánh các phép toán cung cấp bởi C# với C++ và Java?

Câu 7. Sự giống nhau và khác nhau giữa các cấu trúc điều kiện if và switch?

Câu 8. So sánh các cấu trúc lặp for, while và do... while?

Câu 9. Cho biết các lệnh rẽ nhánh có điều kiện và lệnh rẽ nhánh không điều kiện?

Câu 10. Kể tên và ý nghĩa các Collection định nghĩa sẵn của C#?

Câu 11. Các lệnh xử lý ngoại lệ của C#?

BÀI TẬP CHƯƠNG 2

Bài 1. Viết chương trình đổi một ký tự thường thành ký tự in hoa (a - z thành A - Z).

Bài 2. Thời gian hoàn thành một công việc là x giây. Hãy đổi thời gian này ở dạng năm, tháng, ngày và giờ, phút giây.

Bài 3. Tìm số lớn nhất trong 4 số a, b, c, d.

Bài 4. Xếp loại sinh viên xuất sắc, giỏi, khá, trung bình khá, trung bình hay yếu dựa vào điểm trung bình

Bài 5. Giải phương trình bậc hai

$$ax^2 + bx + c = 0$$

Bài 6. Tính tổng sau trên n số nguyên dương đầu tiên

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

Bài 7. Lặp lại chương trình để có thể tính tổng với các giá trị n khác nhau.

Bài 8. Tính tổng sau trên n số nguyên dương đầu tiên

$$S = \frac{1}{2^2} + \frac{3}{4^2} + \frac{5}{6^2} + \dots$$

Bài 9. Tính x^n

Bài 10. Tính n!

Bài 11. Tìm n số Fibonacii đầu tiên. Biết rằng dãy số Fibonacii như sau

1 1 2 3 5 8 13 21 ...

Bài 12. Viết chương trình in ra các số nguyên tố trong n số nguyên dương đầu tiên

Bài 13. Nhập epsilon, và tính tổng sau với độ chính xác epsilon (tính tổng chừng nào số hạng còn lớn hơn hay bằng sai số epsilon)

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$$

Bài 14. Nhập epsilon và tính tổng sau với độ chính xác epsilon (tính tổng chừng nào số hạng còn lớn hơn hay bằng sai số epsilon)

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \dots + (-1)^{n-1} \frac{4}{(2n-1)}$$

Bài 15. Nhập epsilon và tính tổng sau với độ chính xác epsilon (tính tổng chừng nào số hạng còn lớn hơn hay bằng sai số epsilon)

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots + (-1)^n + \frac{x^{2n}}{2n!}$$

Bài 16. Nhập epsilon và tính tổng sau với độ chính xác epsilon (tính tổng chừng nào số hạng còn lớn hơn hay bằng sai số epsilon)

$$\sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + (-1)^{n-1} \frac{x^{(2n-1)}}{(2n-1)!}$$

Bài 17. Nhập epsilon và tính tổng sau với độ chính xác epsilon (tính tổng chừng nào số hạng còn lớn hơn hay bằng sai số epsilon)

$$1 - \frac{x}{1 \times 3} + \frac{x^2}{1 \times 3 \times 5} - \frac{x^3}{1 \times 3 \times 5 \times 7} + \dots + (-1)^n \frac{x^n}{1 \times 3 \times \dots \times (2n-1)(2n+1)}$$

Bài 18. Viết chương trình nhập số dòng và hiển thị tam giác với số dòng (*) như sau:

```
* * * * *
* * * *
* * *
* *
*
```

Bài 19. Viết chương trình nhập số dòng và hiển thị tam giác với số dòng (*) như sau:

```

*
*
*
* * *
* * * * *
* * * * * *
* * * * * * *
```

Bài 20. Viết chương trình in ra bảng cửu chương (từ bảng cửu chương 2 đến bảng cửu chương 9)

“Trăm trâu trăm cỏ

Trâu đírng ăn năm

Trâu năm ăn ba

Ba trâu già ăn một”

Hỏi số trâu mỗi loại?

Chương 3

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VỚI C#

Chương này trình bày khả năng mạnh mẽ của ngôn ngữ lập trình hướng đối tượng C#, so sánh các đặc tính hướng đối tượng của ngôn ngữ C# với Java và C++. Các kiến thức này là nền tảng cho lập trình ứng dụng Windows Form truy xuất cơ sở dữ liệu.

- Định nghĩa lớp và tạo đối tượng
- Truy xuất thành phần của lớp
- Định nghĩa phương thức
- Phương thức khởi tạo (Constructor)
- Phương thức hủy (Destructor)
- Từ khóa this
- Nạp chồng phương thức (Overloading method)
- Truyền tham đối cho phương thức
- Nạp chồng toán tử (Overloading operator)
- Thuộc tính (Property), thuộc tính chỉ đọc (Read only property)
- Kiểu tham chiếu phương thức (Delegate)
- Sự kiện (Event)
- Chỉ mục (Indexer)
- Kiểu cấu trúc (Struct)
- Kiểu tổng quát (Generic type)
- Cây biểu thức (Expression tree)
- Ké thừa lớp (Classical inheritance)

- Viết chồng phương thức (Overriding method) hay che khuất phương thức (Hiding method)
- Từ khóa base
- Không gian tên (Namespace) và lệnh using
- Lớp, phương thức trừu tượng (Abstract class, abstract method)
- Lớp, phương thức hằng (Sealed class, sealed method)
- Khai báo và hiện thực giao tiếp (Interface)

C# là một ngôn ngữ lập trình hướng đối tượng. Kỹ thuật lập trình hướng đối tượng trừu tượng các lớp từ các đối tượng trong thế giới thực, chương trình của bạn sẽ được xây dựng từ các lớp. Lớp là khuôn mẫu của đối tượng, và đối tượng là thể hiện của một lớp.

Tất cả các ngôn ngữ lập trình hướng đối tượng đều có các cơ chế cho phép bạn triển khai các mô hình hướng đối tượng. Đó là tính đóng gói (encapsulation), kế thừa (inheritance), và tính đa hình (polymorphism). C# cung cấp các tính năng thể hiện tính đóng gói, đa hình và kế thừa là các đặc tính chính yếu của bất kỳ một ngôn ngữ lập trình hướng đối tượng nào.

Tính đóng gói là cơ chế dùng một vỏ bọc kết hợp phần dữ liệu và các thao tác trên dữ liệu đó (phần mã lệnh) thành một thể thống nhất, tạo nên sự an toàn, tránh việc sử dụng không đúng thiết kế, bảo vệ cho mã lệnh và dữ liệu chống việc truy xuất từ những đoạn mã lệnh bên ngoài. Trong C#, tính đóng gói thể hiện ở khái niệm lớp (class), không gian tên (namespace), cấu trúc (struct), giao tiếp (interface) và danh sách liệt kê (enumeration).

Tính kế thừa là khả năng xây dựng các lớp mới từ các lớp đã có. Khi lớp đóng gói một số dữ liệu và phương thức, lớp mới sẽ kế thừa cấu trúc dữ liệu và các phương thức của lớp mà nó kế thừa. Ngoài ra nó có thể bổ sung các dữ liệu và các phương thức của riêng mình. Nhờ tính kế thừa mà hệ thống hướng đối tượng có thể thực hiện tái sử dụng mã. Không sử dụng cây phân lớp, mỗi lớp phải định nghĩa tất cả

các dữ liệu và phương thức của mình một cách rõ ràng. Nếu sử dụng sự kế thừa, mỗi lớp chỉ cần định nghĩa thêm những đặc trưng của mình. Cũng như Java, C# là ngôn ngữ không hỗ trợ tính đa kế thừa.

Tính đa hình cho phép một phương thức có các cách thể hiện khác nhau trên nhiều loại đối tượng khác nhau. Các đặc tính của C# thể hiện tính đa hình là nạp chồng phương thức, viết chồng phương thức, lớp trừu tượng, giao tiếp mà ta sẽ tìm hiểu trong chương này

- Khi một lớp được kế thừa từ các lớp tổ tiên thì nó có thể thay đổi cách thức làm việc của lớp tổ tiên trong một số phương thức nào đó (nhưng khai báo phương thức như tên, kiểu trả về, danh sách tham đối của phương thức thì vẫn giữ nguyên). Điều này gọi là viết chồng phương thức (overriding method). Như vậy một thông điệp có thể thực hiện các phương thức khác nhau tùy vào đối tượng nhận thông điệp đó.

Ví dụ: Với phương thức chạy, xe ôtô, xe máy có thể tăng ga, còn xe đạp thì phải đạp...

- Tính đa hình còn thể hiện ở việc cho phép tên của một phương thức được dùng lại trong định nghĩa lớp, có nghĩa là một thông điệp có thể thực hiện các phương thức khác nhau tùy vào tham đối truyền cho phương thức. Điều này gọi là nạp chồng phương thức (overloading method).
- Tính đa hình còn thể hiện ở việc một giao tiếp (interface) hay lớp trừu tượng (abstract class) có thể sử dụng cho các hoạt động của một lớp tổng quát, hay còn gọi là “một giao tiếp, nhiều phương thức”. Có nghĩa là có thể thiết kế một giao tiếp tổng quát cho một nhóm các hành vi liên quan. Vậy giao tiếp hay lớp trừu tượng cho phép các đối tượng khác nhau đáp ứng cùng một thông điệp theo các cách khác nhau.

Sự kết hợp đúng đắn giữa đóng gói, kế thừa và đa hình tạo nên môi trường lập trình có khả năng phát triển tốt hơn rất nhiều so với môi trường không hỗ trợ hướng đối tượng. Chương này sẽ trình

bày cách thức định nghĩa lớp và tạo đối tượng, các thành viên của lớp (class member), đó là các biến thành viên (field), phương thức (method), thuộc tính (property), sự kiện (event), chỉ mục (indexer) và tham chiếu phương thức (delegate). Một đặc tính mới mà ngôn ngữ C# đưa ra để xây dựng lớp là khái niệm thuộc tính (property), thành phần thuộc tính này hoạt động giống như các phương thức để tạo một lớp, bản chất của thuộc tính này là tạo giao tiếp cho bên ngoài tương tác với biến thành viên một cách gián tiếp. Tiếp theo chương này trình bày kiểu dữ liệu cấu trúc, lớp kế thừa, lớp và phương thức trừu tượng, lớp và phương thức hằng và giao tiếp.

3.1. LỚP VÀ ĐỐI TƯỢNG

3.1.1. Xây dựng lớp

Lớp là một cấu trúc dữ liệu nhưng bao gồm dữ liệu và các phương thức thao tác trên dữ liệu đó. Lớp định nghĩa kiểu dữ liệu mới, dùng để tạo đối tượng thuộc kiểu lớp đó.

Khi định nghĩa một lớp, bạn chỉ ra thuộc tính (fields) của lớp được thể hiện bằng biến gọi là biến thành viên (member variable) và hành vi được thể hiện bởi mã lệnh trong phương thức (method), thuộc tính (property), sự kiện (event), chỉ mục (indexer) và tham chiếu phương thức (delegate). Trong hầu hết các lớp, các biến thành viên được truy cập bởi các phương thức của lớp đó. Vì vậy, chính các phương thức quyết định dữ liệu của lớp có thể dùng như thế nào.

Dạng đầy đủ của một định nghĩa lớp như sau:

[public/internal]	Lớp được truy xuất chung cho các gói khác, nếu không có public, mặc định là internal, chỉ được truy xuất trong nội bộ một assembly.
[static]	Lớp tĩnh chỉ chứa thành viên tĩnh. Không thể khởi tạo đối tượng của lớp tĩnh bởi toán tử

	new. Lớp tĩnh là lớp hằng
[abstract]	Lớp trừu tượng, không thể khởi tạo đối tượng
[sealed]	Lớp hằng không có lớp con
[partial]	Phân tách định nghĩa lớp vào nhiều tập tin
class TênLớp	Tên lớp
[: TênLớpCha, GiaoTiếp]	Kế thừa một lớp cha (base class) hay cài đặt các giao tiếp (interfaces)
{	
// Khai báo hằng (constant)	
// Khai báo biến thành viên (field)	
// Khai báo phương thức (method)	
// Khai báo thuộc tính (property)	
// Khai báo sự kiện (event)	
// Khai báo chỉ mục (indexer)	
// Khai báo tham chiếu phương thức (delegate)	
}	

Ngoài ra lớp có thể chứa định nghĩa lớp khác, giao tiếp (interface) và cấu trúc (struct).

Phạm vi nhìn thấy của kiểu (Type visibility): Các kiểu như lớp (class), giao tiếp (interface), cấu trúc (struct), danh sách liệt kê (enumeration), và tham chiếu phương thức (delegate), có thể có điều khiển truy xuất public hay internal. Trong C# internal là tùy chọn mặc định cho các kiểu, nếu bạn không sử dụng từ khóa public

```
// Lớp này có thể dùng cho bất kỳ assembly nào
```

```
public class MyClass{}
```

```
//Những lớp này chỉ dùng trong cùng một assembly
internal class MyHelperClass{}
class FinalHelperClass{} // Mặc định là internal trong C#.
```

Ví dụ: Tạo một lớp các đối tượng nhân viên như sau:

```
class NhanVien
{
    string HoTen;
    int NamSinh;
    float HeSoLuong;
    public static int LuongCB;
}
```

Theo nguyên tắc của tính đóng gói, các biến thành viên nên có điều khiển truy xuất là private (điều khiển mặc định). Khi đó chỉ có phương thức thành viên của lớp truy cập được giá trị của biến.

3.1.2 Tạo đối tượng

Để có được các đối tượng của một lớp phải qua hai giai đoạn:

Khai báo biến kiểu tham chiếu để trỏ đến đối tượng

TênLớp TênĐốiTượng;

Ví dụ : NhanVien A;

Khai báo biến A có kiểu lớp NhanVien, thực ra không cấp phát bộ nhớ đủ chứa đối tượng kiểu lớp NhanVien, mà chỉ tạo ra tham chiếu để trỏ đến đối tượng. Sau câu lệnh này, biến A xuất hiện trên bộ nhớ chưa giá trị null chỉ ra rằng nó chưa trỏ đến một đối tượng thực tế nào.

Sau đó, để thực sự cấp phát bộ nhớ để tạo ra một đối tượng và gán địa chỉ của đối tượng cho biến này, dùng toán tử new

TênĐốiTượng = **new** TênLớp();

Thật ra cú pháp đúng như sau, tuy nhiên ta sẽ tìm hiểu về phương thức khởi tạo sau:

TênĐốiTượng = **new** PhươngThứcKhởiTạo
(DanhSáchThamĐối);

Ví dụ:

```
A = new NhanVien();
```

Có thể kết hợp cả hai bước trên vào một câu lệnh:

```
TênLớp TênĐốiTượng = new TênLớp();
```

Ví dụ:

```
NhanVien A = new NhanVien();
```

Kiểu nặc danh (Anonymous type): C# 3.0 cho phép tạo đối tượng thuộc lớp mà không cần phải định nghĩa lớp trước đó gọi là kiểu nặc danh (anonymous type).

Ví dụ: Tạo đối tượng học viên của một lớp có các thuộc tính mã học viên, họ tên, năm sinh

```
var HocVien = new
{
    mahv = 01,
    hoten = "Nguyen Van A",
    namsinh = 1990
};
```

Khi biên dịch, C# sẽ tạo ra một lớp có 3 thuộc tính như trên. Ngoài ra, nếu có 2 khai báo tạo đối tượng của cùng một lớp, trình biên dịch C# sẽ tạo một lớp cho 2 khai báo tạo đối tượng như vậy.

3.1.3. Truy xuất thành viên của lớp

Các thành viên (members) của một lớp có thể là thành viên tĩnh (static member) hay thành viên đối tượng (instance member). Nói chung, thành viên đối tượng thuộc về đối tượng, và thành viên tĩnh thuộc về lớp.

Khi một biến, phương thức, thuộc tính, chỉ mục, sự kiện, phép toán, phương thức khởi tạo, phương thức hủy là thành viên tĩnh, nó được khai báo với từ khóa static.

Thành viên tĩnh không hoạt động trên một đối tượng cụ thể. Một thành viên tĩnh chia sẻ chung cho mọi đối tượng thuộc lớp, và được truy cập thông qua tên lớp chứa thành viên.

Các thành viên đối tượng hoạt động trên một đối tượng của lớp và được truy cập thông qua tên đối tượng.

Chúng ta xét hai thành viên thông dụng của lớp là biến và phương thức.

3.1.3.1. Truy xuất biến

Ngoài biến thành viên của lớp gồm biến đối tượng và biến tĩnh, còn có biến cục bộ được khai báo bên trong phương thức.

Biến đối tượng (Instance variable): chỉ thuộc tính đối tượng, khi truy xuất phải khởi tạo đối tượng. Biến đối tượng được truy cập thông qua tên đối tượng.

- Khai báo biến đối tượng: Kiểu Biến ĐôiTượng;
- Truy cập biến đối tượng: TênĐôiTượng.BiếnĐôiTượng

Ví dụ:

```
NhanVien nv = new NhanVien();  
System.Console.WriteLine(nv.hoTen);
```

Biến tĩnh (Static variable): về bản chất là biến toàn cục, là biến tĩnh được tạo lập một lần cùng với lớp, dùng chung hay chia sẻ cho mọi đối tượng thuộc lớp, khi truy xuất không cần khởi tạo đối tượng, để trao đổi thông tin của các đối tượng cùng lớp. Biến tĩnh được truy cập thông qua tên lớp.

- Khai báo biến tĩnh: static Kiểu BiếnTĩnh;
- Truy cập biến tĩnh: TênLớp.BiếnTĩnh

Ví dụ:

```
System.Console.WriteLine(NhanVien.luongCB);
```

Biến cục bộ (Local variable): Được khai báo và tồn tại trong phạm vi phương thức. Biến cục bộ bắt buộc phải khởi tạo giá trị trước khi sử dụng.

Ví dụ: Sử dụng biến tĩnh để đếm số đối tượng

```
using System;
public class Bong
{
    private static int soBong = 0;
    public Bong()
    {
        soBong++;
    }
    public static void SoBong()
    {
        Console.WriteLine("{0} So bong bong ", soBong);
    }
}
public class Test
{
    static void Main()
    {
        Bong.SoBong();
        Bong bong1 = new Bong();
        Bong.SoBong();
    }
}
```

3.1.3.2. Lời gọi phương thức (Method caller)

Phương thức có thể là phương thức đối tượng hay phương thức tĩnh. Để gọi phương thức, bạn gọi tên phương thức và truyền vào danh sách tham đối thật sự (actual parameters) tương ứng với số lượng, kiểu và thứ tự các tham đối trong danh sách tham đối hình thức (formal parameters). Khác nhau giữa lời gọi phương thức đối tượng và phương thức tĩnh, đó là phương thức đối tượng được gọi thông qua tên đối tượng, còn phương thức tĩnh được gọi thông qua tên lớp.

Phương thức đối tượng (Instance method): Thuộc về đối tượng và hoạt động trên một đối tượng cụ thể. Gọi phương thức đối tượng như biến đối tượng.

TênĐốiTượng.PhươngThứcĐốiTượng(DanhSáchThamĐối)

Phương thức tĩnh (Static method): Dùng chung cho mọi đối tượng thuộc lớp, khi truy xuất không cần khởi tạo đối tượng. Khai báo phương thức tĩnh bổ sung từ khóa static.

Gọi phương thức tĩnh:

TênLớp.PhươngThứcTĩnh(DanhSáchThamĐối)

Ví dụ:

```
using System;
class NhanVien
{
    private string HoTen; //Biến đối tượng
    private int NamSinh;
    private float HeSoLuong;
    public static int LuongCB; //Biến tĩnh
    public float Luong() //Phương thức đối tượng
    {
        float L = LuongCB * HeSoLuong; //Biến cục bộ
        return L;
    }
}
class Test
{
    public static void Main(string args[])
    {
        NhanVien.LuongCB = 290;
        NhanVien A = new NhanVien();
        Console.WriteLine(A.Luong());
    }
}
```

Các hằng, phương thức chuẩn toán học của lớp Math trong namespace System là hằng, phương thức tĩnh nên khi gọi không cần phải khởi tạo đối tượng.

Hàm, Phương thức	Ý nghĩa	Kiểu giá trị trả về
Math.E	Số E	Số thực
Math.PI	Số Pi	Số thực
Math.Abs(x)	Trị tuyệt đối của số x x có thể là số nguyên hay thực	Cùng kiểu với x
Math.Tan(x)	Tan x	Số thực
Math.Cos(x)	Cos x x tính bằng radian	Số thực
Math.Sin(x)	Sin x	Số thực
Math.Exp(x)	e^x	Số thực
Math.Log(b, a)	Log _a b	Số thực
Math.Round(x, n)	Làm tròn số nguyên với n số lẻ	Số nguyên
Math.Pow(x, y)	Cho x^y với x,y là số thực	Số thực
Math.Sqrt(x)	Khai căn của số x	Số thực

Ví dụ: double s = Math.pow(r,2) * Math.PI;

3.2. PHƯƠNG THỨC

3.2.1. Định nghĩa phương thức (Method definition)

Dạng tổng quát của định nghĩa phương thức như sau:

[ĐiềuKhiểnTruyXuất]	Phạm vi truy cập phương thức
[static]	phương thức tĩnh
[abstract]	phương thức trừu tượng
[sealed]	phương thức hằng
[Kiểu] TênPhươngThức(DanhSáchThamĐôi) { //Thân phương thức }	

Kiểu (Data type): Kiểu dữ liệu do phương thức trả về, có thể là kiểu bất kỳ. Nếu phương thức không trả về giá trị nào, kiểu trả về của nó phải là void.

Các phương thức có kiểu dữ liệu trả về không phải là void, sẽ trả về một giá trị cho phương thức sử dụng câu lệnh return như sau:

return BiểuThức;

Giá trị của biểu thức được tính và trả về cho phương thức.

Danh sách tham đối (Parameter list): Danh sách tham đối truyền cho phương thức, phân cách bởi dấu phẩy, nằm trong hai dấu () ngay sau tên phương thức. Nếu không có tham đối vẫn phải có (). Mỗi tham đối phải khai báo kiểu, có thể là kiểu bất kỳ, có dạng:

Kiểu ThamĐối, Kiểu ThamĐối2 ...

C# cho phép sử dụng từ khóa params để khai báo tham đối cuối cùng trong danh sách tham đối của phương thức là một mảng, do đó một số lượng các tham đối thực sự có thể truyền vào khi gọi phương thức.

Ví dụ: Khai báo phương thức truyền vào mảng số nguyên với từ khóa params

```
public void methodCaller (params int[] a);
```

Và phương thức có thể được gọi với danh sách các số nguyên bất kỳ

```
methodCaller (1);
methodCaller (1, 2, 3, 4);
int[] explicitArray = new int[5]{1,2,3,4,5};
methodCaller(explicitArray);
```

Bên trong phương thức, các tham đối có thể được truy cập thông qua mảng a đã được định nghĩa.

Tham đối đặt tên (Named argument): C# 4.0 giới thiệu tham đối đặt tên (named argument) cho phép bạn không phải nhớ thứ tự tham đối trong danh sách tham đối thực sự của lời gọi phương thức. Các

tham đối thực sự truyền vào phương thức có thể chỉ rõ bởi tên của tham đối hình thức. Tham đối đặt tên cũng làm cho chương trình dễ đọc hơn bởi vì nó chỉ rõ ý nghĩa của tham đối thực sự. Tham đối đặt tên có thể sử dụng cho phương thức, chỉ mục, tham chiếu phương thức.

Ví dụ: Phương thức tính diện tích hình chữ nhật

```
class HCN
{
    static int DienTich(int rong, int dai)
    {
        return dai * rong;
    }
}

class Test
{
    //Lời gọi phương thức
    HCN.DienTich(2, 4);
    //Nếu không nhớ trật tự các tham đối nhưng nhớ tên tham đối hình thức
    //C# 4.0 cho phép gọi phương thức
    HCN.DienTich(dai: 4, rong: 2);
    //Một tham đối đặt tên có thể theo sau một tham đối không đặt tên
    HCN.DienTich(2, dai: 4);
    //Nhưng tham đối không đặt tên không thể theo sau tham đối đặt tên
    HCN.DienTich(dai: 4, 2);
}
```

Tham đối tùy chọn (Optional argument): C# 4.0 cho phép phương thức, chỉ mục hay tham chiếu phương thức có thể chỉ rõ các tham đối là bắt buộc hay tùy chọn. Bất kỳ lời gọi phương thức phải truyền các tham đối bắt buộc nhưng có thể bỏ qua các tham đối tùy chọn.

Mỗi tham đối tùy chọn có một giá trị mặc định, nếu bỏ qua tham đối tùy chọn, tham đối sẽ có giá trị mặc định. Một giá trị mặc

định phải là biểu thức hằng hay kiểu giá trị. Các tham đối tùy chọn được định nghĩa cuối của danh sách tham đối, sau các tham đối bắt buộc. Nếu lời gọi phương thức truyền tham đối cho bất kỳ tham đối tùy chọn, phải truyền tham đối cho bất kỳ tham đối tùy chọn đứng trước trong danh sách tham đối.

Ví dụ: Phương thức đối tượng MyMethod định nghĩa một tham đối bắt buộc và hai tham đối tùy chọn.

```
class MyClass
{
    public void MyMethod(int r, string o1 = "default string", int o2 = 10)
    {
        Console.WriteLine("{0}: {1} và {2}.", r, o1, o2);
    }
    static void Main(string[] args)
    {
        MyClass obj = new MyClass();
        obj.MyMethod(1, "One", 1);
        obj.MyMethod(2, "Two");
        obj.MyMethod(3);
        //Lỗi biên dịch do truyền tham đối tùy chọn thứ ba
        //nhưng không truyền tham đối tùy chọn thứ hai
        //obj.MyMethod(3, ,4);
        //obj.MyMethod(3, 4);
        //Nhưng nếu bạn biết tên tham đối thứ ba, có thể sử dụng tham
        //đối đặt tên
        obj.MyMethod(3, o2: 4);
        //Biên dịch lỗi do không truyền tham đối yêu cầu thứ nhất
        //obj.MyMethod("One", 1);
        //obj.MyMethod();
    }
}
```

Cải tiến hơn giao diện thành phần COM (More COM interface): Các tham đối đặt tên và tùy chọn cùng với các cải tiến khác của C# 3.0, 4.0 đã cải tiến đáng kể tương tác với giao diện thành phần COM.

Chẳng hạn, phương thức AutoFormat trong dịch vụ Excel Interop có 7 tham số, tất cả đều là tùy chọn. Trong C# 3.0 và các phiên bản trước, các tham số của phương thức này đều là tham số bắt buộc.

```
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;
var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFor-
    matAccounting1;
excelApp.get_Range("A1", "B4").AutoFormat(myFormat, Type.Missing,
    Type.Missing,      Type.Missing,      Type.Missing,      Type.Missing,
    Type.Missing);
```

Tuy nhiên, với C# 4.0, bạn có thể đơn giản lời gọi phương thức AutoFormat bằng cách sử dụng tham số đặt tên và tùy chọn. Tham số đặt tên và tùy chọn cho phép bạn bỏ qua tham số truyền vào phương thức nếu không muốn thay đổi giá trị mặc định.

```
excelApp.Range["A1", "B4"].AutoFormat(Format: myFormat);
```

Phương thức mở rộng (Extension method): C# 3.0 cung cấp khả năng định nghĩa phương thức mở rộng (extension method) cho một kiểu tồn tại, mà không phải tạo kiểu kế thừa hay chỉnh sửa kiểu gốc, tái biên dịch.

Phương thức mở rộng phải là phương thức tĩnh (static method), nhưng nó được gọi như một phương thức đối tượng. Kiểu dữ liệu của tham số đầu tiên của phương thức là kiểu lớp cần bổ sung phương thức. Phương thức mở rộng phải đặt trong lớp tĩnh, không lồng nhau (non-nested, non-generic static class). Từ khóa this phải đặt trước tham số đầu tiên.

Ví dụ: Bổ sung phương thức đếm số từ trong một chuỗi cho kiểu string

```

namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}

using ExtensionMethods;
namespace ConsoleApplication1
{
    public class Test
    {
        static void Main(string[] args)
        {
            string s = "Hello Extension Methods";
            int i = s.WordCount();
            Console.WriteLine(i);
        }
    }
}

```

3.2.2. Phạm vi truy xuất thành phần của lớp

Các điều khiển truy xuất (Access modifier) có thể là:

- **public:** Chỉ ra rằng thành phần này có thể được truy xuất bởi bất kỳ dòng lệnh nào dù ở trong hay ngoài lớp, assembly mà nó khai báo.
- **private:** Khi không có điều khiển truy xuất nào được dùng, mặc nhiên là private. Thành phần này chỉ có thể được truy xuất trong lớp của nó, mọi đoạn mã nằm ngoài lớp, kể cả những lớp con đều không có quyền truy xuất.

- protected: Liên quan đến sự kế thừa, thành phần chỉ truy xuất trong lớp và lớp con của nó.
- internal: Thành phần chỉ được truy xuất trong nội bộ một assembly.
- protected internal: Thành phần được truy xuất trong nội bộ một assembly hoặc là trong bất kỳ lớp con của nó từ bất kỳ assembly khác.

Ngoài ra trong định nghĩa phương thức tĩnh của một lớp, không thể truy xuất trực tiếp biến đối tượng và phương thức đối tượng trong lớp đó, chỉ cho phép truy xuất biến tĩnh và phương thức tĩnh.

Ví dụ:

```
class SomeClass
{
    // Truy cập được ở bất kỳ đâu
    public void PublicMethod(){}
    // Chỉ truy cập được trong lớp SomeClass
    private void PrivateMethod(){}
    // Truy cập được trong lớp SomeClass và các lớp kế thừa
    protected void ProtectedMethod(){}
    // Truy cập được trong cùng một assembly
    internal void InternalMethod(){}
    // Truy cập được trong lớp và các lớp kế thừa trong cùng assembly
    protected internal void ProtectedInternalMethod(){}
    // Mặc định là private trong C#
    void SomeMethod(){}
}

class Test
{
    static void Main(string[] args)
    {
        SomeClass c = new SomeClass();
        c.PublicMethod();
        c.InternalMethod();
```

```

        c.ProtectedInternalMethod(); //Lỗi
        c.PrivateMethod();         //Lỗi
        c.ProtectedMethod();       //Lỗi
        c.SomeMethod();           //Lỗi
    }
}

```

3.2.3. Phương thức Main()

Dạng tổng quát của phương thức Main()

```

[public] static void Main(string args[]) {
    // Thân phương thức
}

[public] static int Main(string args[]) {
    // Thân phương thức
}

```

- Ứng dụng chỉ có một lớp có phương thức Main() là phương thức được thực hiện đầu tiên mỗi khi thực hiện ứng dụng.
- Từ khoá static cho phép phương thức Main() được gọi khi không cần khởi tạo đối tượng. Vì phương thức Main() được gọi đầu tiên trước khi bất kỳ đối tượng nào được khởi tạo.
- Từ khoá void cho biết phương thức Main() không trả về giá trị, từ khoá int cho biết phương thức Main() có trả về giá trị số nguyên.
- Tham đối string args[] là một mảng các chuỗi ký tự, chứa các tham đối dòng lệnh khi gọi thực hiện ứng dụng, có thể có hay không.

3.2.4. Phương thức khởi tạo (Constructor)

Có hai loại phương thức khởi tạo: phương thức khởi tạo đối tượng (instance constructor) và phương thức khởi tạo tĩnh (static constructor).

3.2.4.1. Phương thức khởi tạo đối tượng

Có những thao tác cần thực hiện mỗi khi đối tượng được tạo như khởi tạo giá trị cho các biến đối tượng. Các công việc này có thể làm tự động bằng cách dùng phương thức khởi tạo đối tượng.

Phương thức khởi tạo có cùng tên với lớp mà nó thuộc về, chỉ được tự động gọi bởi toán tử new khi đối tượng thuộc lớp được tạo.. Phương thức khởi tạo luôn không có giá trị trả về, khi định nghĩa phương thức không cần ghi void.

Khi bạn không định nghĩa tường minh phương thức khởi tạo cho một lớp, C# sẽ tạo phương thức khởi tạo mặc nhiên cho lớp đó. Phương thức khởi tạo mặc nhiên không có danh sách tham đối, tự động gán tất cả các biến của đối tượng về trị rỗng theo quy ước mặc định của C#, trị 0 cho kiểu số, ký tự ‘\0’ cho kiểu ký tự char, trị false cho kiểu bool, trị null cho các đối tượng.

Ví dụ:

```
using System;
class NhanVien
{
    string HoTen;
    int NamSinh;
    float HeSoLuong;
    public static int LuongCB;
    public NhanVien() {}
    public NhanVien(string ht, int ns, float hsl)
    {
        HoTen = ht;
        NamSinh = ns;
        HeSoLuong = hsl;
    }
    public string GetHoTen()
    {
        return HoTen;
    }
}
```

```

public float Luong()
{
    return LuongCB * HeSoLuong;
}
}
class Test
{
    public static void Main(string[] args)
    {
        NhanVien.LuongCB=290;
        NhanVien nv1 = new NhanVien();
        Console.WriteLine("Luong cua nhan vien {0} la {1}",
                           nv1.GetHoTen(), nv1.Luong());
        NhanVien nv2 = new NhanVien("Nguyen A", 1977, 3.0F);
        Console.WriteLine("Luong cua nhan vien {0} la {1}",
                           nv2.GetHoTen(), nv2.Luong());
    }
}

```

Khởi tạo đối tượng và danh sách (Object and collection initializer): C# 3.0 cũng cung cấp đặc tính khởi tạo đối tượng và danh sách (object and collection initializer) cho phép khởi tạo các đối tượng và danh sách không cần phải có phương thức khởi tạo.

```

class NhanVien
{
    string hoTen;
    int namSinh;
    public string HoTen
    {
        get { return hoTen; }
        set { hoTen = value; }
    }
    public int NamSinh
    {
        get { return namSinh; }
        set { namSinh = value; }
    }
}

```

```

        }
    }
}

class Test
{
    static void Main(string[] args)
    {
        NhanVien v = new NhanVien { HoTen = "Lan", NamSinh = 1983 };
        List<NhanVien> list = new List<NhanVien>
        {
            new NhanVien{HoTen = "Lan", NamSinh = 1980},
            new NhanVien{HoTen = "Hoa", NamSinh = 1982}
        };
        Console.WriteLine(list.Count.ToString());
    }
}

```

3.2.4.2. Phương thức khởi tạo tĩnh (Static constructor)

Phương thức khởi tạo tĩnh sẽ được thực hiện trước khi bất kỳ thê hiện nào của lớp được tạo, thường dùng để khởi tạo giá trị cho các biến tĩnh. Phương thức khởi tạo tĩnh hữu dụng khi chúng ta cần cài đặt một số công việc mà không thể thực hiện khi khởi tạo đối tượng, và chỉ được thực hiện duy nhất một lần. Phương thức khởi tạo tĩnh không có tham đối, không có điều khiển truy cập và không được gọi tường minh mà sẽ được gọi tự động.

Ví dụ: Bổ sung phương thức khởi tạo tĩnh cho lớp NhanVien như sau:

```

class NhanVien
{
    string HoTen;
    int NamSinh;
    float HeSoLuong;
    private static int LuongCB;
    static NhanVien()
    {

```

```

        LuongCB = 290;
    }
}

```

Thay vì sử dụng phương thức khởi tạo tĩnh để khởi tạo biến tĩnh, ta có thể khởi tạo biến tĩnh như sau:

```

private static int LuongCB = 290;
hay

```

```

public static int LuongCB;
và khởi tạo biến tĩnh bất kỳ đâu:

```

```
NhanVien.LuongCB = 290;
```

3.2.5. Phương thức hủy (Destructor)

Mỗi lớp chỉ có một phương thức hủy để hủy một đối tượng của nó. Phương thức hủy chỉ được gọi tự động khi một đối tượng không còn được truy cập nữa, thông qua bộ thu gom rác tự động (garbage collection).

Phương thức hủy cũng có tên trùng tên lớp, nhưng không có tham đối, không có điều khiển truy cập và được bắt đầu bằng dấu ~.

Các đối tượng cấp phát động bằng toán tử new, khi không tồn tại biến nào trả đến đối tượng, đối tượng đó xem như không còn cần đến nữa và bộ nhớ dành cho nó có thể được tự động giải phóng bởi bộ thu gom rác. Trình thu gom rác hoạt động trong một tuyến đoạn (thread) độc lập với chương trình của bạn. Bạn không phải bận tâm gì với công việc này. Tuy nhiên, khi làm việc với các đoạn mã không được quản lý thì cần phải khai báo tường minh phương thức hủy để giải phóng tài nguyên.

Ví dụ:

```

class Library
{
    private int ibooktypes;
    public Library()

```

```

{
    ibooktypes = 5;
}
public Library(int value)
{
    ibooktypes = value;
}
~ Library()
{
    //thực thi câu lệnh
}
}

```

3.2.6. Từ khoá this

Từ khoá this dùng để chỉ đến đối tượng của lớp hiện hành. Bình thường, khi các phương thức truy xuất biến thành viên hay phương thức ngay trong lớp của nó, chỉ cần ghi tên của biến thành viên hay phương thức mà không cần chỉ rõ tên đối tượng hay lớp, C# ngầm hiểu là biến hay phương thức của đối tượng hiện hành.

Tuy nhiên sử dụng từ khoá this để truy xuất phương thức hay biến thành viên ngay trong lớp của nó là cần thiết trong một số trường hợp, hay làm cho chương trình rõ ràng, dễ đọc hơn

this.BiếnThànhViên

this.PhươngThức(DanhSáchTham Đôi)

Chẳng hạn như khi biến cục bộ hay tham đổi truyền cho phương thức được đặt tên trùng với biến thành viên của lớp, nó sẽ che khuất biến thành viên trong thân phương thức, khi đó ta sử dụng từ khoá this để truy xuất biến thành viên.

Không sử dụng từ khóa this để truy xuất thành viên static của lớp.

Ví dụ:

```

using System;
class ViDu
{

```

```
int a;
public void HienThi()
{
    int a = 5;
    Console.WriteLine(a);
}
public ViDu(int a)
{
    this.a = a;
}
public static void Main(string[] args)
{
    ViDu v = new ViDu(2);
    v.HienThi();
}
```

Vi dụ:

```
using System;
namespace NhanVien
{
    class NhanVien
    {
        string HoTen;
        int NamSinh;
        float HeSoLuong;
        public static int LuongCB;
        public NhanVien(string HoTen, int NamSinh, float HeSoLuong)
        {
            this.HoTen = HoTen;
            this.NamSinh = NamSinh;
            this.HeSoLuong = HeSoLuong;
        }
        public float Luong()
        {
            return LuongCB * this.HeSoLuong;
        }
    }
}
```

```
        }
    }
class DSNhanVien
{
    NhanVien[] m = new NhanVien[100];
    int n = 0;
    //Thêm nhân viên vào cuối mảng
    public void Them(NhanVien v)
    {
        m[n] = v;
        n++;
    }
    // Tính tổng lương nhân viên
    public float Tong()
    {
        float TongLuong=0 ;
        for (int i = 0; i <n; i++)
            TongLuong += m[i].Luong();
        return TongLuong;
    }
}
class Test
{
    public static void Main(string[] args)
    {
        DSNhanVien ds = new DSNhanVien();
        NhanVien.LuongCB=290;
        NhanVien nv1 = new NhanVien("Nguyen A", 1975, 1.96F);
        ds.Them(nv1);
        NhanVien nv2 = new NhanVien("Nguyen B", 1977, 3.0F);
        ds.Them(nv2);
        Console.WriteLine("Tong luong la: "+ds.Tong());
    }
}
```

3.2.7. Nạp chồng phương thức (Overloading method)

C# cho phép bạn định nghĩa nhiều phương thức trùng tên trong cùng một lớp, nhưng khác nhau về số lượng hoặc kiểu tham đổi truyền cho phương thức, C# xem đây là các phương thức khác nhau. Khả năng như vậy gọi là nạp chồng phương thức.

Phương thức khởi tạo cũng có thể được nạp chồng như phương thức bình thường nghĩa là ta được phép định nghĩa nhiều phương thức khởi tạo khác nhau về số lượng hay kiểu tham đổi.

Ví dụ:

```
using System;
class NhanVien
{
    string HoTen;
    int NamSinh;
    float HeSoLuong;
    public static int LuongCB;
    public NhanVien() {}
    public NhanVien(string HoTen, int NamSinh, float HeSoLuong)
    {
        this.HoTen = HoTen;
        this.NamSinh = NamSinh;
        this.HeSoLuong = HeSoLuong;
    }
    public float Luong()
    {
        return LuongCB * HeSoLuong;
    }
    public string GetHoTen()
    {
        return HoTen;
    }
    public static void Main(string[] args)
    {
        NhanVien.LuongCB = 290;
```

```

NhanVien nv1 = new NhanVien();
Console.WriteLine("Luong cua {0} la {1}", nv1.GetHoTen(),
nv1.Luong());
NhanVien nv2 = new NhanVien("Nguyen Thi Hoa", 1980, 1.96F);
Console.WriteLine("Luong cua {0} la {1}", nv2.GetHoTen(),
nv2.Luong());
}
}

```

3.2.8. Truyền tham đối cho phương thức

C# cung cấp ba cách truyền tham đối cho phương thức: truyền bằng giá trị (value), truyền bằng tham chiếu (reference) và truyền kết quả (output).

- *Truyền bằng tham chiếu*: truyền tham chiếu, không phải truyền giá trị của tham đối thực sự cho tham đối hình thức, mà sẽ thực hiện trên chính đối tượng truyền vào, vì vậy thay đổi với tham đối hình thức sẽ tác động đến tham đối thực sự, nghĩa là tham đối thực sự sẽ chịu ảnh hưởng bởi sự thay đổi của phương thức. Tham đối này gọi là tham biến. Tham đối có kiểu tham chiếu sẽ được truyền tham chiếu vào phương thức.
- *Truyền bằng giá trị*: tạo bản sao của các tham đối truyền vào phương thức, nghĩa là sao chép giá trị của tham đối thực sự cho tham đối hình thức. Vì vậy thay đổi trên tham đối hình thức không tác động đến tham đối thực sự, nghĩa là tham đối hình thức không thay đổi trước và sau khi truyền cho phương thức. Tham đối này gọi là tham trị. Tham đối có kiểu giá trị sẽ được truyền giá trị vào cho phương thức.

Ví dụ: Truyền tham đối dưới dạng tham trị cho phương thức

```

using System;
class Program
{
    public void Change(int num)

```

```

    {
        num = 5;
    }
}
class Test
{
    static void Main(string[] args)
    {
        int val = 2;
        Program p = new Program();
        p.Change(val);
        Console.WriteLine("Gia tri cua val sau khi goi phuong thuc la
{0}", val);
    }
}

```

Chú ý với kiểu string là kiểu tham chiếu nhưng truyền bởi giá trị, bởi vì các thao tác trên string thật sự tạo ra một đối tượng mới, chúng ta thực hiện trên bản sao để tham chiếu đến đối tượng mới, nhưng thật sự chuỗi gốc không thay đổi.

Ví dụ:

```

using System;
class Program
{
    static void HoanVi(string s1, string s2)
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        Console.WriteLine("Trong phuong thuc: {0} {1}", s1, s2);
    }
    static void Main(string[] args)
    {
        string str1 = "Hello";
        string str2 = "World";
        Console.WriteLine("Truoc khi hoan vi: {0} {1}", str1, str2);
    }
}

```

```

        HoanVi(str1, str2);
        Console.WriteLine("Sau khi hoan vi: {0} {1}", str1, str2);
    }
}

```

Sử dụng từ khóa ref để truyền tham chiếu: C# còn cung cấp khả năng cho phép ta truyền các tham đối có kiểu giá trị dưới hình thức là tham chiếu, bằng cách bổ sung từ khóa ref.

Ví dụ: Truyền tham đối dưới dạng tham chiếu cho phương thức.

```

using System;
class Program
{
    public void Change(ref int num)
    {
        num = 5;
    }
}
class Test
{
    static void Main(string[] args)
    {
        int val = 2;
        Program p = new Program();
        p.Change(ref val);
        Console.WriteLine("Gia tri cua val sau khi goi phuong thuc la
{0}", val);
    }
}

```

Để kiểu tham chiếu string có thể truyền bằng tham chiếu, sử dụng từ khóa ref như sau:

Ví dụ:

```

using System;
class Program
{
    static void HoanVi(ref string s1, ref string s2)

```

```

{
    string temp = s1;
    s1 = s2;
    s2 = temp;
    Console.WriteLine("Trong phương thuc: {0} {1}", s1, s2);
}
static void Main(string[] args)
{
    string str1 = "Hello";
    string str2 = "World";
    Console.WriteLine("Truoc khi hoan vi: {0} {1}", str1, str2);
    HoanVi(ref str1, ref str2);
    Console.WriteLine("Sau khi hoan vi: {0} {1}", str1, str2);
}
}

```

Có thể sử dụng lớp StringBuilder trong namespace System.Text để truyền tham chiếu như sau:

Ví dụ:

```

using System;
using System.Text;
class Program
{
    static void HoanVi(StringBuilder s1, StringBuilder s2)
    {
        StringBuilder temp = new StringBuilder();
        temp.Append(s1);
        s1.Remove(0, s1.Length);
        s1.Append(s2);
        s2.Remove(0, s2.Length);
        s2.Append(temp);
        Console.WriteLine("Trong phương thuc: {0} {1}", s1, s2);
    }
    static void Main(string[] args)
    {
        StringBuilder str1 = new StringBuilder("Hello");

```

```
        StringBuilder str2 = new StringBuilder("World");
        Console.WriteLine("Truoc khi hoan vi: {0} {1}", str1, str2);
        HoanVi(str1, str2);
        Console.WriteLine("Sau khi hoan vi: {0} {1}", str1, str2);
    }
}
```

Sử dụng từ khóa out để truyền tham chiếu kết quả (output):

Ngôn ngữ C# bắt buộc phải thực hiện một phép gán cho biến trước khi sử dụng, do đó khi khai báo một biến cục bộ thì trước khi sử dụng các biến này, phải gán giá trị cho biến. Vì vậy, nếu bạn muốn truyền tham đối kiểu giá trị dưới dạng tham chiếu khi gọi phương thức, nhưng không cần phải khởi tạo giá trị cho tham đối, mà chỉ muốn nhận giá trị của tham đối, phải sử dụng từ khóa out.

Ví dụ: Truyền tham đối dưới dạng kết quả cho phương thức

```
using System;
namespace TruyenThamDoi
{
    class Program
    {
        public static void Sum(int num1, int num2, out int result)
        {
            result = num1 + num2;
        }
    }
    class Test
    {
        static void Main(string[] args)
        {
            int val1 = 2, val2 = 5;
            int output;
            Program.Sum(val1, val2, out output);
            Console.WriteLine("Tong la {0}", output);
        }
    }
}
```

3.2.9. Nạp chồng toán tử (Overloading operator)

Phương thức nạp chồng toán tử là một thành viên của lớp, để định nghĩa một phép toán sử dụng cho đối tượng của lớp. Có ba kiểu phép toán có thể nạp chồng: một ngôi (unary operator), hai ngôi (binary operator) và phép chuyển kiểu. Cú pháp phương thức nạp chồng ba kiểu phép toán này như sau:

```
BởiTừ Kiểu Operator PhépToánMộtNgôi(ThamĐối){  
    //Thân phương thức  
}  
  
BởiTừ Kiểu Operator PhépToánHaiNgôi(Tham Đôi1,  
ThamĐối2){  
    //Thân phương thức  
}  
  
BởiTừ implicit/ explicit Operator  
KiểuChuyểnĐổi(ThamĐối){  
    //Thân phương thức  
}
```

Trong đó:

- Bở từ (Modifier) là các từ khóa public, static hay extern.

Từ khoá extern chỉ ra rằng phương thức được gọi tồn tại trong một thư viện DLL

- Một phương thức nạp chồng toán tử phải là một phương thức tĩnh static
- Kiểu (Data type) là kiểu dữ liệu trả về của phép toán
- Phép toán một ngôi là phép toán có một toán hạng như + - ! ~ ++ --
- Phép toán hai ngôi là phép toán có hai toán hạng như + - * / % & | ^ <<>> == != ><=> <=

- Khi nạp chồng phép toán so sánh, phải nạp chồng cho phép toán trái ngược của nó, ví dụ nạp chồng phép so sánh ==, ta phải nạp chồng cả phép so sánh !=, nếu không chương trình sẽ báo lỗi.
- Kiểu chuyển đổi (Convert type) là kiểu dữ liệu cần chuyển đổi thành.
- implicit/ explicit là từ khóa cho phép chuyển kiểu mặc định hay tường minh.

Ví dụ:

Chuyển kiểu mặc định phân số a thành double:

```
double d = a;
```

Chuyển kiểu tường minh phân số a thành double:

```
double d = (double) a;
```

- Tham đối (Parameter) có dạng: Kiểu ThamĐối
- Thân phương thức phải có lệnh trả về giá trị cho phép toán

Ví dụ: Định nghĩa các phép toán cho lớp PhanSo

```
using System;
class PhanSo
{
    int tu, mau;
    public int Tu
    {
        get {
            return tu;
        }
    }
    public int Mau
    {
        get {
            return mau;
        }
    }
}
```

```
public PhanSo(int tu, int mau)
{
    this.tu = tu;
    this.mau = mau;
}
// Nạp chồng phép cộng
public static PhanSo operator+(PhanSo a, PhanSo b)
{
    PhanSo tong = new PhanSo(a.tu * b.mau + b.tu * a.mau,
                           a.mau * b.mau);
    return tong;
}
// Nạp chồng phép nhân
public static PhanSo operator*(PhanSo a, PhanSo b)
{
    return new PhanSo(a.tu * b.tu, a.mau * b.mau);
}
// Nạp chồng phép chuyển kiểu mặc định (implicit) thành double
// Hay nạp chồng phép chuyển kiểu tường minh (explicit thay cho
// implicit)
public static implicit operator double(PhanSo a)
{
    return (double) a.tu / a.mau;
}
// Nạp chồng phép so sánh ==
public static bool operator==(PhanSo a, PhanSo b)
{
    if(a.tu*b.mau == a.mau*b.tu)
        return true;
    return false;
}
// Nạp chồng phép so sánh !=
public static bool operator!=(PhanSo a, PhanSo b)
{
    if (a.tu*b.mau != a.mau*b.tu)
        return true;
    return false;
```

```

        }
    }
}

class Test
{
    public static void Main()
    {
        PhanSo a = new PhanSo(1, 2);
        PhanSo b = new PhanSo(3, 7);
        PhanSo c = new PhanSo(2, 3);
        // Chuyển kiểu mặc định
        Console.WriteLine(a * b + c);
        // Chuyển kiểu tường minh
        Console.WriteLine((double)(a * b + c));
    }
}

```

3.3. THUỘC TÍNH (PROPERTIES)

Thuộc tính là một đặc tính mới được giới thiệu trong ngôn ngữ C#. Đặc tính này cung cấp khả năng bảo vệ các biến dữ liệu bên trong một lớp bằng việc đọc và ghi chúng thông qua thuộc tính. Trong ngôn ngữ khác, điều này có thể được thực hiện thông qua việc tạo các phương thức truy cập (getter method hay accesser) và phương thức thiết lập (setter method hay mutator).

Ví dụ:

```

class ViDu
{
    int a;
    public int geta ()
    {
        return this.a;
    }
    public void seta ( int a )
    {
        this.a = a;
    }
}

```

Có thể được viết trong C# như sau, và có thể dễ dàng truy cập trực tiếp property bên trong một chương trình C#.

```

class ViDu
{
    int a;
    public int A
    {
        get {
            return this.a;
        }
        set {
            // value là biến được tạo ra bởi trình biên dịch để thay thế
            // tham đối
            this.a = value;
        }
    }
}
class Test
{
    static void Main(string[] args)
    {
        ViDu v = new ViDu();
        v.A = 3;
        Console.WriteLine(v.A);
    }
}

```

Lợi ích của việc sử dụng thuộc tính (property) là cho phép các thành phần bên ngoài có thể truy cập thuộc tính một cách trực tiếp mà vẫn đảm bảo tính đóng gói trong thiết kế hướng đối tượng.

Thuộc tính được tự động cài đặt (Auto-implemented properties): C# 3.0 cho phép các thuộc tính của lớp được tự động cài đặt (auto-implemented properties). Đa số các properties đều thực hiện các thao tác như là ghi và đọc từ 1 biến private nào đó. Trong C# 3.0, nếu như properties của bạn chỉ thực hiện các thao tác đọc và ghi vào biến

private thì chỉ cần khai báo ngắn gọn như sau, trình biên dịch sẽ tự động tạo biến private vô danh tương tự khai báo tường minh. Để một thuộc tính chỉ được đọc hay ghi giá trị trong lớp chứa thuộc tính, thêm từ khóa private trước từ khóa get hay set.

Ví dụ:

```
public class ViDu
{
    public int A { get; set; }
    //public int A { get; private set; }
}
```

Thuộc tính chỉ đọc (Read only property): Giả sử chúng ta muốn tạo lớp Time cung cấp một số thuộc tính static để hiển thị ngày hiện hành.

Ví dụ:

```
using System;
public class RightNow
{
    public static int Year;
    public static int Month;
    public static int Date;
    static RightNow()
    {
        DateTime dt = DateTime.Now;
        Year = dt.Year;
        Month = dt.Month;
        Date = dt.Day;
    }
}
public class Tester
{
    static void Main()
    {
        Console.WriteLine("This year: {0}", RightNow.Year.ToString());
```

```

        RightNow.Year = 2003;
        Console.WriteLine("This year: {0}", RightNow.Year.ToString());
    }
}

```

Biến thành viên chỉ năm hiện hành Year đã được thay đổi thành 2003. Điều này thực sự không như mong muốn, các biến tĩnh này không được thay đổi. Nhưng khai báo hàng cũng không được vì biến tĩnh không được khởi tạo cho đến khi phương thức khởi tạo tĩnh được thi hành. Do vậy C# cung cấp bổ sung từ khóa readonly cho mục đích này. Với ví dụ trên ta khai báo lại như sau:

```

public static readonly int Year;
public static readonly int Month;
public static readonly int Date;

```

Khi đó ta phải bỏ lệnh gán biến thành viên Year, vì nếu không sẽ bị báo lỗi:

```
// RightNow.Year = 2003;
```

3.4. THAM CHIẾU PHƯƠNG THỨC (DELEGATE)

Một tham chiếu phương thức (delegate) trong C# tương tự với con trỏ hàm (function pointer) trong C++. Một delegate cho phép chúng ta định nghĩa một hay nhiều phương thức mà delegate tham chiếu đến. Đối tượng delegate được sử dụng để gọi một hay nhiều phương thức mà nó tham chiếu. Delegate là một kiểu tham chiếu đến phương thức, cư xử như phương thức đó, và có thể được sử dụng như bất kỳ phương thức với các tham đối và giá trị trả về.

Có ba bước định nghĩa và sử dụng delegate: Khai báo (declaration), khởi tạo (initiation) và gọi phương thức tham chiếu bởi delegate (invocation).

Khai báo delegate: Khai báo delegate giống như khai báo phương thức mà delegate tham chiếu đến

```
ĐiềuKhiểnTruyCập delegate KiểuDelegate
(DanhSáchThamĐối);
```

Trong đó:

- Kiểu Delegate: Là tên kiểu delegate
- Kiểu (Data type): Kiểu kết quả trả về của phương thức mà delegate tham chiếu đến
- Danh sách tham đối: Của phương thức mà delegate tham chiếu đến

Ví dụ:

```
public delegate void SimpleDelegate();  
public delegate void SimpleDelegate(int x);
```

Khởi tạo delegate: Khởi tạo một đối tượng Delegate, để chỉ ra các phương thức mà delegate tham chiếu đến

KiểuDelegate TênĐốiTượng = new KiểuDelegate
(TênPhươngThức);

- Tên đối tượng: là tên đối tượng delegate (delegate object)
- Tên phương thức: là phương thức mà delegate tham chiếu đến

Ví dụ:

```
SimpleDelegate simpleDelegate = new SimpleDelegate(MyFunc);
```

Để thêm và xóa phương thức delegate tham chiếu đến, sử dụng phép toán += và -=

Ví dụ:

```
SimpleDelegate simpleDelegate += new SimpleDelegate(MyFunc);  
SimpleDelegate simpleDelegate += new SimpleDelegate(MyFunc1);  
SimpleDelegate simpleDelegate -= new SimpleDelegate(MyFunc1);
```

Gọi phương thức tham chiếu bởi delegate:

TênĐốiTượng(DanhSáchThamĐối);

Tên đối tượng: Là tên đối tượng delegate (delegate object)

Ví dụ:

```
simpleDelegate();  
simpleDelegate(2);
```

Ví dụ: Delegate tham chiếu đến phương thức không truyền tham đối

```
using System;
namespace BasicDelegate
{
    // Khai báo delegate
    public delegate void SimpleDelegate();
    class TestDelegate
    {
        public static void MyFunc()
        {
            Console.WriteLine("Phương thức được gọi bởi delegate");
        }
        public static void Main()
        {
            // Khởi tạo delegate
            SimpleDelegate simpleDelegate = new
                SimpleDelegate(MyFunc);
            // Gọi delegate
            simpleDelegate();
        }
    }
}
```

Ví dụ: Delegate tham chiếu đến phương thức có truyền tham đối

```
using System;
namespace BasicDelegate
{
    // Khai báo delegate
    public delegate void SimpleDelegate(int x);
    class TestDelegate
    {
        public static void MyFunc(int n)
        {
            Console.WriteLine("Phương thức được gọi bởi delegate");
            Console.WriteLine(n);
        }
    }
}
```

```

public static void Main()
{
    // Khởi tạo delegate
    SimpleDelegate simpleDelegate = new SimpleDelegate(MyFunc);
    // Gọi delegate
    simpleDelegate(2);
}
}
}

```

Phương thức nặc danh (Anonymous method): Trong phiên bản trước của C# 2.0, ta chỉ có thể khai báo delegate tham chiếu đến phương thức đặt tên (named method). C# 2.0 giới thiệu phương thức nặc danh (anonymous method) cho phép ta khi khởi tạo đối tượng delegate, không cần tách riêng phương thức mà delegate tham chiếu đến như ví dụ sau:

```

// Khai báo delegate
public delegate void SimpleDelegate(int x);
// Khởi tạo delegate sử dụng phương thức nặc danh
SimpleDelegate simpleDelegate = delegate(int n)
{
    Console.WriteLine("Phương thức được gọi bởi delegate");
    Console.WriteLine(n);
};
// Gọi delegate
simpleDelegate(2);

```

Biểu thức Lambda (Lambda expression): C# 3.0 đưa ra khái niệm biểu thức lambda (lambda expression). Biểu thức lambda là một thao tác nặc danh (anonymous function) chứa biểu thức hay khối lệnh, có thể sử dụng để tạo delegate hay cây biểu thức (expression tree). Biểu thức lambda sử dụng phép toán lambda =>. Vết trái phép toán lambda là các tham đối đầu vào và vết phải là biểu thức hay khối lệnh:

(DanhSáchThamĐối) => BiểuThức/ KhốiLệnh

Nếu không có tham đối thì cú pháp sẽ như sau:

`() => BiểuThức/ KhốiLệnh`

Biểu thức lambda: `x => x * x` có thể được gán đến một đối tượng delegate như sau:

```
// Khai báo delegate
public delegate int SimpleDelegate(int x);
// Khởi tạo delegate sử dụng biểu thức lambda
SimpleDelegate simpleDelegate = x => x * x;
// Gọi delegate
int y = simpleDelegate(2);
Console.WriteLine(y);
```

Hay biểu thức lambda chứa khối lệnh có thể gán đến đối tượng delegate như sau:

```
// Khai báo delegate
public delegate void SimpleDelegate(int x, int y);
// Khởi tạo delegate sử dụng biểu thức lambda
SimpleDelegate simpleDelegate = (x, y) =>
{
    Console.WriteLine(x);
    Console.WriteLine(y);
};
// Gọi delegate
simpleDelegate(2,3);
```

3.5. SỰ KIỆN (EVENT)

C# hỗ trợ lập trình sự kiện bằng cách bổ sung khái niệm sự kiện (event) và tham chiếu phương thức (delegate).

Một event là một thành viên của lớp, cho phép một lớp khai báo sự kiện xảy ra khi chạy chương trình, và xác định một delegate sẽ được gọi khi xảy ra sự kiện. Delegate có thể có một hay nhiều phương thức kết hợp sẽ được gọi khi sự kiện phát sinh. Nhiều sự kiện có thể chia sẻ cùng một delegate.

Các bước để tạo một sự kiện trong C# là khai báo một sự kiện (event declaration), đăng ký sự kiện (event accessor) và phát sinh sự kiện (event raise).

Khai báo event: Để khai báo event trong một lớp, đầu tiên phải khai báo một kiểu delegate tham chiếu đến phương thức được gọi khi sự kiện phát sinh.

Ví dụ:

```
public delegate void Handler();
```

Kế tiếp ta khai báo event, một event được khai báo giống như một biến kiểu delegate, ngoại trừ việc bổ sung từ khóa event, event thường khai báo public, nhưng có thể sử dụng các điều khiển truy cập khác:

[BổTù] **event** KiểuDelegate TênSựKiện;

- Bổ từ (Modifier) là tùy chọn, có thể là new, virtual, override, abstract, sealed, static, extern.
- Kiểu delegate (Delegate type) là kiểu delegate được gọi khi phát sinh sự kiện.

Ví dụ:

```
public event Handler WokenUp;
```

Đăng ký event: Mã đăng ký event để thêm hay xóa delegate từ một event. Đăng ký event có thể thực hiện khi kết hợp và xóa một delegate vào event, sử dụng toán tử += và -=.

```
TênSựKiện += new KiểuDelegate(TênPhươngThức);
```

```
TênSựKiện -= new KiểuDelegate(TênPhươngThức);
```

Tên phương thức: Là phương thức mà delegate tham chiếu đến

Ví dụ:

```
private void HandleEvent(object sender, EventArgs e)
{
    Console.WriteLine("Your animal has woken up. Feed it now!");
}
WokenUp += new Handler(HandleEvent);
```

Khi đã gọi sự kiện, xóa delegate khỏi event bằng cách sử dụng toán tử -=.

```
WokenUp -= new Handler(HandleEvent);
```

Phát sinh một event: Khi đã khai báo event, có thể sử dụng event như là một biến của kiểu delegate. Phát sinh một event như gọi một delegate

TênSựKiện(DanhSáchThamĐôi)

Danh sách tham đối là tham đối của phương thức mà delegate tham chiếu đến.

Ví dụ:

```
if (WokenUp != null)
    WokenUp();
```

Ví dụ:

```
using System;
using System.Threading;
namespace ConsoleApplication1
{
    //Khai báo kiểu delegate
    public delegate void Handler();
    public class Animal
    {
        //Khai báo event
        public event Handler WokenUp;
        public void Sleep()
        {
            Thread.Sleep(3000);
            // Phát sinh event
```

```

        if (WokenUp != null)
            WokenUp();
    }
}
public class Farmer
{
    public Farmer()
    {
        //Thêm delegate vào event
        Animal a = new Animal();
        a.WokenUp += new Handler(HandleEvent);
        a.Sleep();
    }
    private void HandleEvent()
    {
        Console.WriteLine("Your animal has woken up. Feed it now!");
    }
    public static void Main()
    {
        Console.WriteLine("Begin execution");
        Farmer f = new Farmer();
        Console.WriteLine("End execution");
        Console.ReadKey();
    }
}
}

```

3.6. CHỈ MỤC (INDEXER)

Một indexer là một thành viên của lớp cho phép một đối tượng của một lớp hay struct được chỉ số như là một mảng. Các thành phần của mảng được truy cập sử dụng dấu ngoặc vuông. Khai báo indexer giống như thuộc tính, ngoại trừ khi các phương thức truy cập và thiết lập có tham đối. Một indexer được khai báo như sau:

```

[BỎ TÙ] Kiểu this [DanhSáchThamĐối]
{
    //Mã kết hợp với indexer
}

```

Trong đó:

- Bỏ từ là tùy chọn, và phải là điều khiển truy cập hay một trong các từ khóa new, virtual, sealed, override, hay abstract.
- Kiểu (Return type) là kiểu dữ liệu trả về của indexer
- Danh sách tham đối (Parameter list) của indexer. Ít nhất phải có một tham đối truyền cho indexer. Tham đối khai báo ref và out là không được phép
- Mã kết hợp với indexer tương tự các phương thức truy cập và thiết lập sử dụng cho thuộc tính

Ví dụ:

```
using System;
namespace ConsoleApplication1
{
    class SampleCollection<T>
    {
        // Khai báo một mảng
        private T[] arr = new T[100];
        // Định nghĩa Indexer, cho phép đối tượng của lớp hiện hành sử
        // dụng []
        public T this[int i]
        {
            get {
                //Indexer này đơn giản trả về phần tử của mảng
                return arr[i];
            }
            set {
                arr[i] = value;
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            SampleCollection<string> stringCollection =
new SampleCollection<string>();
```

```

        stringCollection[0] = "Hello, World";
        System.Console.WriteLine(stringCollection[0]);
    }
}
}
}

```

3.7. KIỂU CÁU TRÚC (STRUCT)

Cáu trúc là một kiểu giá trị, được sử dụng để đóng gói tập hợp các đặc tính liên quan. Tương tự định nghĩa lớp, cáu trúc chứa biến thành viên, thuộc tính và phương thức, tham chiếu phương thức, sự kiện và chỉ mục. Cáu trúc giúp bạn tổ chức dữ liệu khoa học hơn, mã được trung sáng dễ hiểu hơn và cải tiến tốc độ chương trình và bộ nhớ sử dụng.

Cáu trúc là kiểu giá trị, trong khi lớp là kiểu tham chiếu. Cáu trúc không có phương thức hủy, nhưng lớp có thể có phương thức hủy. Một điểm khác nhau nữa giữa cáu trúc và lớp là cáu trúc không thể kế thừa, nhưng có thể cài đặt giao tiếp như lớp. Vì vậy, các thành viên của cáu trúc không thể khai báo protected.

Nếu có không nhiều thành viên, hay chỉ có thuộc tính trong cáu trúc và không kế thừa hay đa hình, bạn nên sử dụng kiểu cáu trúc thay cho kiểu lớp.

Cáu trúc cũng có phương thức khởi tạo, nhưng cáu trúc không thể định nghĩa một phương thức khởi tạo mà không có tham đổi. Cáu trúc định nghĩa các phương thức khởi tạo với các tham đổi chính xác như tất cả các biến thành viên.

Giống lớp, phương thức khởi tạo mặc định khởi tạo tất cả các thuộc tính về giá trị rỗng, nguyên là 0, thực là 0.0, boolean là false, ký tự là '\0', và kiểu tham chiếu là null.

Cú pháp định nghĩa và sử dụng cáu trúc tương tự lớp.

Ví dụ:

```

using System;
struct Rectangle
{

```

```
int width;
public int Width
{
    get { return width; }
    set { width = value; }
}
int height;
public int Height
{
    get { return height; }
    set { height = value; }
}
public Rectangle(int width, int height)
{
    this.width = width;
    this.height = height;
}
public int Area()
{
    return this.width * this.height;
}
}
class StructExample
{
    static void Main()
    {
        Rectangle rect1 = new Rectangle();
        rect1.Width = 1;
        rect1.Height = 3;
        Console.WriteLine("rect1: {0}, {1}", rect1.Width, rect1.Height);
        Rectangle rect2 = new Rectangle(5, 7);
        Console.WriteLine("rect2: {0}, {1}", rect2.Width, rect2.Height);
        Console.WriteLine("Area of rect2: {0}", rect2.Area());
        Console.ReadKey();
    }
}
```

3.8. KIỀU TỔNG QUÁT (GENERIC TYPE)

C# 2.0 hỗ trợ lập trình kiểu tổng quát với lớp, cấu trúc, giao tiếp, tham chiếu phương thức và phương thức.

Tham đổi kiểu tổng quát cho phép bạn định nghĩa lớp, cấu trúc, giao tiếp, tham chiếu phương thức hay phương thức sử dụng kiểu tổng quát bất kỳ, sẽ được chỉ rõ kiểu khi bạn khởi tạo lớp, cấu trúc, hay sử dụng phương thức. Lập trình kiểu tổng quát hỗ trợ tính tái sử dụng cao, tiết kiệm thời gian công sức lập trình, mà độ hoàn thiện của chương trình không giảm.

3.8.1. Lớp (generic class), giao tiếp (generic interface) và cấu trúc tổng quát (generic struct)

Chúng ta đã quen thuộc với lớp tổng quát List xây dựng sẵn thuộc không gian tên System.Collections.Generic. Lớp List cho phép chúng ta khai báo và khởi tạo một danh sách các phần tử có kiểu bất kỳ bằng cách chỉ rõ một tham đổi kiểu bên trong cặp dấu ngoặc nhọn <>. Tham đổi kiểu có thể là bất kỳ kiểu nào của C# hay .NET. Có thể khai báo và khởi tạo nhiều danh sách khác nhau, mỗi danh sách sử dụng một tham đổi kiểu khác nhau. Chẳng hạn khai báo danh sách các chuỗi như sau:

```
List<string> stringList = new List<string>();
List<float> floatList = new List<float>();
```

Theo phương pháp như vậy, bạn có thể định nghĩa các lớp, giao tiếp và cấu trúc tổng quát khác nhau:

```
public interface IInterface<T1, T2 ... >{ }
public class CClass<T1, T2...> { }
public struct SStruct<T1, T2...> { }
```

Chẳng hạn bạn có thể định nghĩa một bộ đôi, bộ ba, hay mảng chứa các phần tử có kiểu bất kỳ nào đó, sẽ được chỉ rõ khi khởi tạo đối tượng

```
public class Couple<T, E>
{
    public T itemA {get; set;}
```

```

public E itemB {get; set;}
public Couple(T itemA, E itemB)
{
    this.itemA = itemA;
    this.itemB = itemB;
}

```

Để tạo đối tượng gồm hai phần tử string và double, bạn khai báo như sau:

```
Couple<string, double> couple = new Couple<string,
double>("SV01", 7.5);
```

3.8.2. Phương thức tổng quát (generic method)

C# cũng cho phép lập trình tổng quát với phương thức và tham chiếu phương thức.

Ví dụ: Phương thức hoán vị giá trị hai biến có kiểu bất kỳ

```

using System;
class Program
{
    static void HoanVi<T>(ref T v1, ref T v2)
    {
        T temp = v1;
        v1 = v2;
        v2 = temp;
    }
    static void Main(string[] args)
    {
        string str1 = "Hello";
        string str2 = "World";
        Console.WriteLine("Truoc khi hoan vi: {0} {1}", str1, str2);
        HoanVi<string>(ref str1, ref str2);
        Console.WriteLine("Sau khi hoan vi: {0} {1}", str1, str2);
    }
}
```

3.9. CÂY BIỂU THỨC (EXPRESSION TREE)

Cây biểu thức biểu diễn mã dạng cấu trúc cây, trong đó mỗi nốt là một biểu thức, ví dụ một lời gọi phương thức hay một phép toán nhị ngôi như $x < y$.

Cây biểu thức được sử dụng để tạo truy vấn LINQ (Language-Integrated Query) và DLR (Dynamic Language Runtime) cung cấp tương tác giữa các ngôn ngữ động và nền tảng .NET framework.

Bạn có thể tạo cây biểu thức dựa trên biểu thức lambda hay tạo thủ công sử dụng không gian tên System.Linq.Expressions.

3.9.1. Tạo cây biểu thức từ biểu thức lambda

Khi một biểu thức lambda được gán cho biến kiểu Expression<TDelegate>, trình biên dịch sẽ xây dựng cây biểu thức biểu diễn biểu thức lambda.

Ví dụ: Tạo cây biểu thức biểu diễn biểu thức lambda $x \Rightarrow x * x$

```
using System.Linq.Expressions;
public delegate int del(int x);
Expression<del> myET = x => x * x;
```

3.9.2. Tạo cây biểu thức sử dụng API (application programming interface)

Không gian tên System.Linq.Expressions cung cấp lớp Expression chứa các phương thức tĩnh để tạo các nốt của cây biểu thức. Lớp ParameterExpression, biểu diễn một biến hay tham đổi, MethodCallExpression biểu diễn lời gọi phương thức.

Ví dụ: Tạo biểu thức cây biểu diễn biểu thức lambda num \Rightarrow num < 5

```
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 = Expression.Lambda<Func<
    int, bool>>(numLessThanFive, new
    ParameterExpression[] { numParam });
```

Trong .NET Framework 4, cây biểu thức API cũng hỗ trợ biểu thức phép gán và luồng điều khiển như vòng lặp, cấu trúc điều kiện và lệnh try catch. Bằng cách sử dụng API, bạn có thể tạo cây biểu thức phức tạp hơn tạo từ biểu thức lambda.

Ví dụ: Tạo cây biểu thức tính lũy thừa của một số

```
// Tạo một biểu thức chứa tham số
ParameterExpression value = Expression.Parameter(typeof(int), "value");
// Tạo một biểu thức chứa biến cục bộ
ParameterExpression result = Expression.Parameter(typeof(int), "result");
// Tạo một nhãn để nhảy đến từ một vòng lặp
LabelTarget label = Expression.Label(typeof(int));
// Tạo thân phương thức
BlockExpression block = Expression.Block(
(
    // Bổ sung biến cục bộ
    new[] { result },
    // Gán hằng đến biến cục bộ
    Expression.Assign(result, Expression.Constant(1)),
    // Bổ sung vòng lặp
    Expression.Loop(
        (
            // Bổ sung khối điều kiện vào vòng lặp
            Expression.IfThenElse(
                (
                    // Điều kiện: value > 1
                    Expression.GreaterThan(value, Expression.Constant(1)),
                    // Nếu true: result*=value—
                    // ngược lại thoát vòng lặp và nhảy đến nhãn
                    Expression.MultiplyAssign(result,
                        Expression.PostDecrementAssign(value)),
                    Expression.Break(label, result)
                ),
            )
        )
    )
),
    // Bổ sung khía cạnh
    Expression.PostIncrementAssign(result)
);
```

```

label
)
);
//Biên dịch và thực hiện cây biểu thức
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()(5);
Console.WriteLine(factorial);

Duyệt cây biểu thức: Mã sau duyệt cây biểu thức biểu diễn biểu
thức lambda num => num < 5.

using System.Linq.Expressions;
// Tạo cây biểu thức
Expression<Func<int, bool>> exprTree = num => num < 5;
// Duyệt cây biểu thức
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;
Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
    param.Name, left.Name, operation.NodeType, right.Value);

```

3.10. KẾ THỪA LỚP (CLASSICAL INHERITANCE)

3.10.1. Định nghĩa lớp kế thừa

Ta có thể tạo lớp tổng quát có những đặc tính chung đại diện cho nhiều lớp có cùng dữ liệu và hành vi. Sau đó, lớp này có thể được kế thừa bởi một hay nhiều lớp khác, lớp được kế thừa gọi là lớp cha (base class), lớp kế thừa gọi là lớp con (subClass hay derived class). Lớp con kế thừa tất cả các thuộc tính, phương thức định nghĩa trong lớp cha, ngoài trừ các thành phần khai báo private.

```

class LớpCon : LớpCha
{
    //Khai báo thuộc tính
    //Khai báo phương thức
    //....
}

```

Tuy nhiên bạn chỉ có thể định nghĩa một lớp cha cho lớp con của bạn, C# không hỗ trợ đa kế thừa lớp, và thậm chí bỏ qua khai báo kế thừa, lớp của bạn mặc định cũng có một lớp cha là Object hay object thuộc không gian tên System. Trong C#, lớp cao nhất, lớp mà từ đó tất cả các lớp dẫn xuất từ, là lớp Object định nghĩa trong không gian tên System. Lớp Object định nghĩa dữ liệu, hành vi mà mọi lớp trong C# cần đến.

3.10.2. Viết chòng phương thức (Overriding method) hay che khuất phương thức (Hiding method)

Trong phân cấp lớp, khi một phương thức của lớp con có cùng tên, và giống nhau về số lượng và kiểu tham đổi cũng như kiểu trả về với một phương thức ở lớp cha, thì phương thức ở lớp con được gọi là viết chòng (override) hay che khuất (hide) phương thức kế thừa từ lớp cha.

Khác với C++ và Java, trong C# các phương thức trong một đối tượng mặc định không được viết chòng. Bằng cách khai báo virtual cho một phương thức của lớp cơ sở, sẽ cho phép phương thức đó được viết chòng trong bất kỳ lớp con nào. Và các phương thức viết chòng phương thức virtual phải được khai báo override. Bạn không thể khai báo virtual cho phương thức tĩnh. Khai báo override có thể sử dụng cho phương thức ảo (virtual method) và phải sử dụng cho phương thức trừu tượng (abstract method).

```
public class MyBaseClass
{
    public virtual void VirtualMethod() { }

    public class MyDerivedClass : MyBaseClass
    {
        //Viết chòng phương thức
        public override void VirtualMethod () { }
    }
}
```

Nếu một phương thức ở lớp con có cùng khai báo phương thức ở lớp cơ sở nhưng không viết chòng (override) có nghĩa là phương thức lớp con che khuất (hide) phương thức kế thừa từ lớp cơ sở. Trong hầu hết mọi trường hợp viết chòng một phương thức hơn là che khuất nó. Nếu tạo ra hai phương thức có cùng khai báo ở cả lớp con và lớp cơ sở mà không có khai báo override thì sẽ bị cảnh báo trong khi biên dịch. Trong C#, nên sử dụng từ khoá new để đảm bảo muôn che khuất phương thức đó.

```
public class MyBaseClass
{
    public void Method() { }

}
public class MyDerivedClass : MyBaseClass
{
    //Nếu che khuất phương thức ở lớp cơ sở, khai báo new
    public new void Method () { }
}
```

Xét ví dụ sau, bạn sẽ thấy sự khác nhau của khai báo new hay override cho phương thức của lớp con:

Ví dụ:

```
using System;
namespace QLNV
{
    class NhanVien
    {
        protected string HoTen;
        protected int NamSinh;
        public NhanVien(string HoTen, int NamSinh)
        {
            this.HoTen = HoTen;
            this.NamSinh = NamSinh;
        }
    }
}
```

```

public virtual void HienThi()
{
    Console.WriteLine("Ho ten la:{0}", HoTen);
    Console.WriteLine("Nam sinh la:{0}", NamSinh);
}
class NVBienChe:NhanVien
{
    float HeSoLuong;
    public NVBienChe(string HoTen, int NamSinh, float HeSoLuong)
    {
        this.HoTen = HoTen;
        this.NamSinh = NamSinh;
        this.HeSoLuong = HeSoLuong;
    }
    public override void HienThi()
    {
        Console.WriteLine("Ho ten la:{0}", HoTen);
        Console.WriteLine("Nam sinh la:{0}", NamSinh);
        Console.WriteLine("He so luong la:{0}", HeSoLuong);
    }
    public static void Main()
    {
        NhanVien NV;
        NV = new NhanVien("Nguyen Thi Hoa", 1980);
        NV.HienThi(); //Gọi phương thức HienThi() của lớp cơ sở
        NV = new NVBienChe("Nguyen Thi Hoa", 1980, 1.96F);
        NV.HienThi(); //Gọi phương thức HienThi() của lớp con
    }
}

```

Nếu bạn khai báo new cho phương thức HienThi(), cả 2 dòng lệnh gọi phương thức HienThi() trên đều gọi phương thức HienThi() của lớp cơ sở.

Việc khai báo từ khóa override cho phương thức của lớp con, sẽ hữu dụng khi bạn khai báo một danh sách vừa nhân viên biên chế, và

nhân viên hợp đồng. Và tùy thuộc nhân viên là biên chế hay hợp đồng, phương thức HienThi() tương ứng của lớp con NVBienChe hay NVHopDong sẽ được gọi.

3.10.3. Từ khóa base

Đôi khi bạn không muốn thực hiện viết chòng phương thức mà chỉ muốn thêm chức năng vào phương thức thừa kế từ lớp cha. Khi đó, bạn gọi phương thức thừa kế từ lớp cha dùng từ khoá base. Từ khoá base dùng khi lớp con cần tham chiếu lớp cha trực tiếp của nó. base có hai dạng cú pháp:

Dạng 1: Phương thức khởi tạo lớp cha phải được gọi trước phương thức khởi tạo của lớp con. Bạn bổ sung lệnh gọi phương thức khởi tạo ở lớp cha, theo sau dấu hai chấm cuối khai báo phương thức khởi tạo của lớp con

base(DanhSáchThamĐối)

Danh sách tham đối là danh sách các tham đối thực sự cần thiết cho phương thức khởi tạo của lớp cha.

Dạng 2: dùng để phương thức lớp con truy xuất phương thức hay thuộc tính kế thừa từ lớp cha:

base.PhươngThức(DanhSáchThamĐối)

Ví dụ: Lớp NVBienChe có thể viết lại như sau, sử dụng từ khoá base. Phương thức ToString() của lớp NhanVien viết chòng phương thức ToString() kế thừa từ lớp Object thuộc không gian tên System. Phương thức ToString() sẽ được gọi khi chuyển đổi tượng thành một string

Console.WriteLine(NV.ToString());

Có thể viết là: Console.WriteLine(NV);

```
using System;
namespace QLNV
{
    class NhanVien
```

```
{  
    protected string HoTen;  
    protected int NamSinh;  
    public NhanVien(string HoTen, int NamSinh)  
    {  
        this.HoTen = HoTen;  
        this.NamSinh = NamSinh;  
    }  
    public virtual void HienThi()  
    {  
        Console.WriteLine("Ho ten la:{0}", HoTen);  
        Console.WriteLine("Nam sinh la:{0}", NamSinh);  
    }  
    public override String ToString()  
    {  
        return "Ho ten la:" + HoTen + "\nNam sinh la:" + NamSinh;  
    }  
}  
class NVBieuChe:NhanVien  
{  
    float HeSoLuong;  
    public NVBieuChe(string HoTen, int NamSinh, float HeSoLuong) :  
        base(HoTen, NamSinh)  
    {  
        this.HeSoLuong = HeSoLuong;  
    }  
    public override void HienThi()  
    {  
        base.HienThi(); //Gọi phương thức HienThi() của lớp cơ sở  
        Console.WriteLine("He so luong la:{0}", HeSoLuong);  
    }  
    public override String ToString()  
    {  
        return base.ToString() + "\nHe so luong la:" + HeSoLuong;  
    }  
    public static void Main()
```

```

    {
        NhanVien NV;
        NV = new NhanVien("Nguyen Thi Hoa", 1980);
        NV.HienThi(); //Gọi phương thức HienThi() của lớp cơ sở
        Console.WriteLine(NV);
        NV = new NVBienChe("Nguyen Thi Hoa", 1980, 1.96F);
        NV.HienThi(); //Gọi phương thức HienThi() của lớp con
        Console.WriteLine(NV);
    }
}
}

```

3.11. KHÔNG GIAN TÊN (NAMESPACE) VÀ LỆNH USING

3.11.1. Khái niệm namespace

Namespace trong C# là kỹ thuật phân hoạch không gian các định danh, các kiểu dữ liệu thành những vùng dễ quản lý hơn, nhằm tránh sự xung đột giữa việc sử dụng các thư viện khác nhau từ các nhà cung cấp. Ví dụ khi bạn tạo một lớp trong một namespace nào đó, bạn không cần phải kiểm tra xem nó có bị trùng tên với một lớp nào đó trong namespace khác không.

Ngoài thư viện namespace do Microsoft .NET và các hãng thứ ba cung cấp, ta có thể tạo riêng cho mình các namespace.

3.11.2. Định nghĩa namespace

Để tạo một namespace sử dụng cú pháp sau:

```

namespace KhôngGianTên
{
    //Định nghĩa lớp A
    //Định nghĩa lớp B...
}
```

Ví dụ:

```

using System;
namespace MyLib
```

```
{  
    namespace Demo1  
    {  
        class Example1  
        {  
            public static void Show1()  
            {  
                Console.WriteLine("Lop Example1");  
            }  
        }  
    }  
    namespace Demo2  
    {  
        public class Tester  
        {  
            public static void Main()  
            {  
                Demo1.Example1.Show1();  
                Demo1.Example2.Show2();  
            }  
        }  
    }  
}
```

Lớp Example2 có cùng namespace MyLib.Demo1 với lớp Example1 nhưng hai khai báo không cùng một tập tin

```
namespace MyLib.Demo1  
{  
    class Example2  
    {  
        public static void Show2()  
        {  
            Console.WriteLine("Lop Example2");  
        }  
    }  
}
```

3.11.3. Sử dụng namespace

C# đưa ra từ khóa using để khai báo không gian tên cho việc sử dụng các định danh, kiểu dữ liệu định nghĩa thuộc không gian tên trong chương trình:

using KhôngGianTên;

Thay vì sử dụng lệnh using, có thể sử dụng dấu chấm truy cập namespace.

Ví dụ: using System;

Cho phép ta sử dụng Console.WriteLine() thay cho System.Console.WriteLine()

Ví dụ: using Demo1;

Cho phép ta truy cập Example1.Show1() thay cho Demo1.Example1.Show1();

using cũng có thể cung cấp một không gian tên bí danh

Ví dụ: using Utils = Company.Application.Utilities;

3.11.4. Lệnh using

Từ khóa using được sử dụng với hai ý nghĩa hoàn toàn không liên quan. Ngoài việc sử dụng như một khai báo không gian tên cho việc tham chiếu, từ khóa using còn được sử dụng để tự động gọi phương thức Dispose() cho đối tượng cụ thể.

Ví dụ: Sau khi đọc tập tin, đối tượng reader sẽ được tự động hủy

using (StreamReader reader = new StreamReader("vb.txt"))

{

 //Đọc tập tin

}

Ví dụ:

using System;

using System.IO;

class Program

```

{
    static void Main(string[] args)
    {
        using (StreamReader reader = new StreamReader("vb.txt"))
        {
            string line;
            while ((line = reader.ReadLine()) != null)
                Console.WriteLine(line);
        }
    }
}

```

3.12. LỚP, PHƯƠNG THỨC TRÙU TƯỢNG (ABSTRACT CLASS, METHOD)

Trong trường hợp chúng ta muốn định nghĩa một lớp cha theo một cấu trúc trừu tượng cho trước mà không cần định nghĩa đầy đủ các thành phần phương thức. Tức là ta muốn tạo một lớp cha có dạng chung cho tất cả các lớp con và để các lớp con hiện thực chi tiết. Khi đó, chắc chắn lớp con có viết chồng phương thức. Những phương thức phải được viết chồng trong lớp con gọi là phương thức trừu tượng, được khai báo abstract và không có phần thân phương thức.

abstract Kiểu TênPhươngThức(DanhSáchThamĐôi);

Bất kỳ lớp nào chứa một hay nhiều phương thức trừu tượng cũng phải khai báo trừu tượng, sử dụng từ khoá abstract trước từ khoá class. Không thể khởi tạo đối tượng kiểu lớp trừu tượng, vì lớp trừu tượng không được định nghĩa đầy đủ. Bất kỳ lớp con nào cũng phải hoặc là viết chồng tất cả các phương thức trừu tượng hoặc chính nó lại được khai báo abstract.

Ví dụ:

```

using System;
abstract class NhanVien
{

```

```
protected string manv;
protected string hoten;
protected int namsinh;
public string MaNV
{
    get { return this.manv; }
}
public string HoTen
{
    get { return this.hoten; }
}
public int NamSinh
{
    get { return this.namsinh; }
}
public NhanVien(string manv, string hoten, int namsinh)
{
    this.manv = manv;
    this.hoten = hoten;
    this.namsinh = namsinh;
}
public virtual void HienThi()
{
    Console.WriteLine("Ho ten la:{0}", hoten);
    Console.WriteLine("Nam sinh la:{0}", namsinh);
}
public abstract double Luong();
}
class NVBienChe:NhanVien
{
    public float HeSoLuong {get; private set;}
    public static int LuongCB;
    public NVBienChe(string MaNV, string HoTen, int NamSinh, float
        HeSoLuong): base(MaNV, HoTen, NamSinh)
    {
        this.HeSoLuong = HeSoLuong;
    }
}
```

```

public override void HienThi()
{
    base.HienThi();
    Console.WriteLine("He so luong la:{0}", HeSoLuong);
}
public double Luong()
{
    return HeSoLuong * LuongCB;
}
public static void Main()
{
    NhanVien NV;
    NV = new NhanVien("Nguyen Thi Hoa", 1980);
    NV.HienThi();
    NV = new NVBienChe("Nguyen Thi Hoa", 1980, 1.96F);
    NV.HienThi();
}
}

```

3.13. LỚP, PHƯƠNG THỨC HẮNG (SEALED CLASS, SEALED METHOD)

C# cho phép các lớp và phương thức được khai báo sealed. Nếu là lớp có nghĩa là bạn không được quyền thừa kế lớp đó, nếu là phương thức tức là bạn không được phép viết chồng phương thức đó.

Ví dụ:

```

sealed class FinalClass { }

class DerivedClass : FinalClass {}           // Biên dịch lỗi
class MyClass
{
    public sealed void FinalMethod() {}
}

class DerivedClass : MyClass
{
    public override void FinalMethod() {}     // Biên dịch lỗi
}

```

3.14. GIAO TIẾP (INTERFACE)

3.14.1. Khai báo giao tiếp

Với từ khoá interface, có thể trùu tượng hoàn toàn giao tiếp của lớp khỏi sự hiện thực của nó. Interface là giao tiếp của một lớp. Bạn đã biết cách định nghĩa một lớp, đó là khai báo lớp với các đặc tả thuộc tính, phương thức, chỉ mục và sự kiện và cài đặt cụ thể nội dung cho từng phương thức. Giao tiếp của một lớp chính là phần đặc tả của lớp (không bao gồm phần cài đặt cụ thể).

Khai báo giao tiếp có cú pháp tương tự lớp, có thể bao gồm thuộc tính, phương thức, chỉ mục và sự kiện, nhưng những phương thức của chúng là phương thức trùu tượng (dù không có abstract) không có thân, chúng không được hiện thực trong giao tiếp. Giao tiếp chỉ chứa khai báo thuộc tính, phương thức, sự kiện và tham chiếu phương thức. Giao tiếp không chứa các biến thành viên, hằng, phương thức khởi tạo, phương thức hủy và các thành viên static. Các thành phần của giao tiếp tự động ngầm hiểu là public, và không có bất kỳ điều khiển truy cập nào.

Giao tiếp được khai báo sử dụng từ khóa interface, như sau:

```
[public/internal] [partial] interface TênGiaoTiếp
{
    Kiểu TênThuộcTính { get; set; }
    Kiểu TênPhươngThức(DanhSáchThamĐối);
}
```

3.14.2. Hiện thực (cài đặt) giao tiếp

Giao tiếp được thiết kế để hỗ trợ quyết định phương thức động lúc thời gian chạy. Phương thức của giao tiếp sẽ được cài đặt bởi lớp hay struct hiện thực giao tiếp.

Thông thường, để gọi phương thức của một lớp, phương thức cần hiện diện lúc thời gian dịch, gọi là liên kết sớm. Điều này làm cho môi trường lớp trở nên tĩnh và không có khả năng mở rộng. Trong một hệ thống như vậy cây phân cấp càng ngày càng bị đẩy lên cao.

Vì vậy, giao tiếp được định nghĩa để hạn chế việc ngày càng nhiều lớp con. Nó tách định nghĩa các phương thức ra khỏi cây phân cấp kế thừa. Do đó các lớp không có quan hệ trong cây phân cấp cũng có thể hiện thực cùng một giao tiếp. Ta có thể thấy đây thực sự là thế mạnh giao tiếp. Giao tiếp giải quyết tính phức tạp của đa kế thừa, một lớp có thể hiện thực nhiều giao tiếp. Và nhiều lớp có thể hiện thực cùng một giao tiếp, có cài đặt các phương thức khác nhau, thể hiện đặc điểm “một giao tiếp, nhiều phương thức” của tính đa hình. Chỉ khi chạy chương trình, tùy thuộc vào đối tượng gọi phương thức, mới biết phương thức trong lớp hiện thực giao tiếp tương ứng được gọi, đó là liên kết muộn.

Các lớp hay struct hiện thực giao tiếp, phải cài đặt đầy đủ các phương thức của giao tiếp. Nếu chỉ cài đặt một số phương thức, thì lớp hay struct phải khai báo trừu tượng (abstract). Những phương thức hiện thực giao tiếp phải khai báo public và non-static. Hình thức khai báo của phương thức hiện thực phải giống hệt khi nó được đặc tả trong khai báo giao tiếp.

Khai báo lớp và struct hiện thực giao tiếp tương tự như khai báo lớp kế thừa lớp cơ sở:

```
[public/internal] [abstract] class TênLớp :
    TênGiaoDiện1, ... TênGiaoDiệnn
{
    public Kiểu TênThuộcTính
    {
        get { }
        set { }
    }
    public Kiểu TênPhươngThức1(DanhSáchThamĐôi)
    {
        //Thân phương thức
    }
}
```

Một giao tiếp có thể khai báo một thuộc tính với phương thức get, nhưng lớp cài đặt giao tiếp có thể khai báo thuộc tính này với cả hai phương thức get và set.

Một lớp A kế thừa lớp B và hiện thực giao tiếp C và D, khai báo như sau:

```
class A : B, C, D { }
```

Chú ý rằng, lớp B được đặt trước giao tiếp C, D trong danh sách.

Giao tiếp có thể kế thừa một giao tiếp khác. Cú pháp khai báo kế thừa tương tự lớp.

Ví dụ: Hình chữ nhật, hình tam giác, hình vuông... đều là những hình phẳng cơ bản và đều có ít nhất một thuộc tính là một cạnh, các phương thức như diện tích, chu vi... Vì vậy, cần xây dựng một giao tiếp IHinhPhang, và các lớp HinhChuNhat, HinhVuong... sẽ hiện thực các phương thức khai báo trong giao tiếp.

```
using System;
interface IHinhPhang
{
    int Canh1 { get; set; }
    int DienTich();
    int ChuVi();
}
class HinhChuNhat : IHinhPhang
{
    public int Canh1 { get; set; }
    public int Canh2 { get; set; }
    public HinhChuNhat(int ChieuDai, int ChieuRong)
    {
        this.Canh1 = ChieuDai;
        this.Canh2 = ChieuRong;
    }
    public int DienTich()
    {
        return Canh1 * Canh2;
```

```
        }
        public int ChuVi()
        {
            return (Canh1 + Canh2) * 2;
        }
    }
    class MainClass
    {
        public static void Main()
        {
            IHinhPhang HCN = new HinhChuNhat(2, 3);
            Console.WriteLine("Hinh chu nhat co chieu dai la {0} va chieu
                rong la {1}", HCN.Canh1, ((HinhChuNhat)HCN).Canh2);
            Console.WriteLine("Dien tich la {0}", HCN.DienTich());
            Console.WriteLine("Chu vi la {0}", HCN.ChuVi());
        }
    }
}
```

CÂU HỎI CHƯƠNG 3

Câu 1. Mối liên quan giữa đối tượng và lớp?

Câu 2. Sự khác nhau giữa biến tĩnh và biến đối tượng, phương thức tĩnh và phương thức đối tượng?

Câu 3. Sự khác nhau giữa biến cục bộ và biến thành viên?

Câu 4. Phương thức tĩnh có thể truy cập được thành viên nào và không truy cập được thành viên nào trong một lớp?

Câu 5. Các điều khiển truy cập biến thành viên và phương thức trong C#

Câu 6. Thể nào là thuộc tính (property)?

Câu 7. So sánh delegate và event?

Câu 8. Các chức năng nào của C# thể hiện tính đóng gói?

Câu 9. Phương thức khởi tạo là gì? Có thể gọi phương thức khởi tạo như gọi các phương thức khác?

Câu 10. Nạp chồng phương thức là gì? Có thể nạp chồng phương thức khởi tạo?

Câu 11. Sự khác nhau giữa tham đối thực sự và tham đối hình thức của một phương thức?

Câu 12. Các kiểu truyền tham đối trong C#?

Câu 13. Sự khác nhau giữa từ khóa base và từ khóa this?

Câu 14. Khái niệm, định nghĩa và sử dụng không gian tên?

Câu 15. Ý nghĩa của lệnh using?

Câu 16. So sánh lớp trừu tượng và giao tiếp?

Câu 17. Từ khóa sealed sử dụng cho khai báo các thành phần nào?

Câu 18. Từ khóa abstract sử dụng cho khai báo các thành phần nào?

Câu 19. Viết chồng phương thức là gì?

Câu 20. Tham đối kiểu tổng quát sử dụng trong trường hợp nào?

Câu 21. Khai báo giao tiếp gồm các thành viên nào?

Câu 22. So sánh giữa khai báo lớp kế thừa lớp cha, và khai báo lớp cài đặt giao tiếp?

Câu 23. Các tính năng nào của C# thể hiện tính đa hình?

BÀI TẬP CHƯƠNG 3

Bài 1. Xây dựng phương thức cho lớp ngân hàng. Ngân hàng này chỉ chấp nhận tiền lẻ là tờ 500 đồng

- Thuộc tính của lớp ngân hàng: số tờ 500 đồng có trong ngân hàng
- Phương thức trả về số tờ 500 đồng có trong ngân hàng
- Phương thức thay đổi số tờ 500 đồng trong ngân hàng với giá trị mới là tham đối của phương thức

- Phương thức trả về true nếu có đủ tiền trong ngân hàng để mua một mặt hàng. Giá tiền của mặt hàng là tham số truyền vào phương thức, và tính bằng ngàn đồng. Phương thức sẽ trả về false nếu không đủ tiền trả cho mặt hàng. Ví dụ, nếu giá là 50 ngàn đồng, tham số sẽ là 50
- Phương thức xóa tất cả tiền trong ngân hàng và trả về số tiền trong ngân hàng tính bằng ngàn đồng. Ví dụ, nếu ngân hàng chứa 7 tờ 500 đồng, phương thức trả về 3.5
- Phương thức xóa đủ số tiền để trả cho mặt hàng có giá tính bằng ngàn đồng. Ví dụ, nếu mặt hàng có giá 12.3 ngàn đồng, phương thức sẽ xóa 24 tờ 500 đồng trong ngân hàng. Nếu không có đủ số tiền để trả cho mặt hàng, rồi thì không xóa bất kỳ số tiền nào từ ngân hàng và hiển thị một thông báo lỗi
- Cho trước một đối tượng ngân hàng khác và một số tờ 500 đồng nào đó, phương thức này chuyển số tờ 500 đồng này từ ngân hàng khác đến ngân hàng này. Nếu không đủ số tờ 500 đồng trong ngân hàng khác để chuyển, phương thức sẽ chuyển tất cả số tờ trong ngân hàng khác đến ngân hàng này

Bài 2. Xây dựng lớp Tam giác gồm:

- Thuộc tính độ dài cạnh thứ nhất, độ dài cạnh thứ hai, độ dài cạnh thứ ba
- Phương thức nhập độ dài 3 cạnh
- Phương thức tính chu vi tam giác
- Phương thức tính diện tích tam giác
- Phương thức kiểm tra ba cạnh đúng của tam giác không
- Phương thức xác định loại tam giác

Bài 3. Tạo lớp số phức bao gồm:

- Thuộc tính: phần thực, phần ảo

- Phương thức khởi tạo số phức
- Phương thức in số phức
- Phương thức nạp chồng phép cộng 2 số phức
- Phương thức nạp chồng phép trừ 2 số phức
- Phương thức nạp chồng phép nhân 2 số phức
- Phương thức nạp chồng phép chia 2 số phức

Hướng dẫn: Viết lớp SoPhuc gồm các thuộc tính phần thực r, phần ảo i và các phương thức theo yêu cầu, với cách tính như sau:

Cộng, trừ 2 số phức:

$$r = r1 \pm r2$$

$$i = i1 \pm i2$$

Nhân 2 số phức:

$$r = r1 * r2 - i1 * i2$$

$$i = r1 * i2 + r2 * i1$$

Chia 2 số phức:

$$r = (r1 * r2 + i1 * i2) / (r2 * r2 + i2 * i2)$$

$$i = (i1 * r2 - r1 * i2) / (r2 * r2 + i2 * i2)$$

Bài 4. Tạo lớp phân số bao gồm:

- Thuộc tính: tử số, mẫu số
- Phương thức khởi tạo phân số
- Phương thức in phân số
- Phương thức tính ước số chung lớn nhất
- Phương thức rút gọn phân số
- Phương thức nạp chồng phép cộng 2 phân số
- Phương thức nạp chồng phép trừ 2 phân số
- Phương thức nạp chồng phép nhân 2 phân số

- Phương thức nạp chồng phép chia 2 phân số
- Phương thức nạp chồng phép so sánh hai phân số
- Phương thức nạp chồng phép chuyển kiểu phân số thành số thực

Bài 5. Xây dựng ứng dụng quản lý sinh viên. Lớp SinhVien gồm:

- Thuộc tính: mã sinh viên, họ tên sinh viên, năm sinh, lớp, địa chỉ, điểm tổng kết
- Phương thức khởi tạo các thuộc tính của lớp SinhVien
- Các phương thức get và set để nhận và thiết lập các giá trị thuộc tính cần thiết
- Phương thức trả về xếp loại của sinh viên, biết rằng sinh viên xếp loại

<i>Điểm tổng kết</i>	<i>Xếp loại</i>
9 - 10	Xuất sắc
8 - <9	Giỏi
7 - <8	Khá
5 - <7	Trung bình
<5	Không đạt

Xây dựng lớp DSSV bao gồm:

- Thuộc tính: Mảng các đối tượng sinh viên, số phần tử sinh viên hiện có trong mảng
- Phương thức thêm một sinh viên vào mảng
- Phương thức thống kê số lượng sinh viên đạt
- Phương thức tìm sinh viên tương ứng với mã sinh viên chỉ rõ
- Phương thức xóa sinh viên tương ứng với mã sinh viên
- Phương thức xóa tất cả sinh viên thuộc lớp chỉ rõ

- Phương thức kiểm tra sinh viên có thuộc lớp nào đó không
- Phương thức tìm sinh viên có điểm tổng kết lớn nhất

Bài 6. Xây dựng ứng dụng quản lý danh sách các sách trong thư viện.

Xây dựng lớp Sach gồm:

- Thuộc tính: mã sách, tên sách, năm xuất bản, đơn giá, số lượng
- Phương thức khởi tạo các thuộc tính của lớp Sach
- Các phương thức get và set để nhận và thiết lập các giá trị thuộc tính cần thiết
- Phương thức tính giá sách = số lượng x đơn giá
- Phương thức tính giá tiền vận chuyển biết rằng:

Giá vận chuyển

Nếu số lượng ≤ 50	0
Ngược lại, nếu số lượng ≤ 500	2% của giá sách
Ngược lại	5% của giá sách

Xây dựng lớp DMSach, để quản lý danh sách các sách trong thư viện bao gồm:

- Thuộc tính: mảng các đối tượng sách, số lượng sách hiện có trong mảng
- Phương thức thêm một sách vào mảng
- Phương thức tính tổng giá sách (kể cả giá vận chuyển) trong mảng
- Phương thức xóa sách tương ứng với mã sách chỉ rõ
- Phương thức tìm sách có số lượng lớn nhất trong thư viện
- Phương thức tìm sách có mã sách chỉ rõ
- Phương thức đếm các sách có số lượng từ 50 đến 100 trong mảng

Bài 7. Đề quản lý nhân viên trong công ty, các thông tin chung của nhân viên lưu trữ là:

- Mã nhân viên, họ tên, năm sinh
- Có hai loại nhân viên: biên chế và hợp đồng. Nhân viên biên chế, bổ sung thông tin: hệ số lương, lương cơ bản. Nhân viên hợp đồng, bổ sung thông tin: mức lương

Yêu cầu: Tạo lớp trừu tượng NhanVien bao gồm các thuộc tính và phương thức:

- Thuộc tính: Mã nhân viên, Họ tên, Năm sinh
- Phương thức khởi tạo các thuộc tính nhân viên
- Phương thức ảo (virtual) hiển thị thông tin nhân viên
- Phương thức trừu tượng (abstract) tính lương

Tạo lớp NVBC kế thừa lớp NhanVien bao gồm các thuộc tính và phương thức:

- Thuộc tính: Hệ số lương, Lương cơ bản
- Phương thức khởi tạo các thuộc tính nhân viên biên chế
- Viết chòng phương thức hiển thị thông tin nhân viên
- Cài đặt phương thức tính lương

Tạo lớp NVHD kế thừa lớp NhanVien gồm các thuộc tính và phương thức:

- Thuộc tính: Mức lương
- Phương thức khởi tạo các thuộc tính của nhân viên hợp đồng
- Viết chòng phương thức hiển thị thông tin nhân viên
- Cài đặt phương thức tính lương

Tạo lớp DSNV bao gồm các thuộc tính và phương thức:

- Thuộc tính: Mảng các nhân viên, Số phần tử của mảng

- Phương thức khởi tạo danh sách nhân viên vừa biên chế vừa hợp đồng
- Phương thức thêm một nhân viên vào cuối danh sách
- Phương thức in danh sách nhân viên
- Phương thức sắp xếp danh sách nhân viên theo lương giảm dần
- Phương thức đếm số nhân viên biên chế
- Phương thức tìm nhân viên có mã nhân viên chỉ rõ
- Phương thức tìm nhân viên có lương nhỏ nhất đầu tiên trong mảng
- Phương thức tính tổng quỹ lương phải trả
- Phương thức xóa nhân viên có mã nhân viên chỉ rõ

TÀI LIỆU THAM KHẢO

- [1] **Nguyễn Ngọc Bình Phương, Thái Thanh Phong**, *Các giải pháp lập trình C#*, NXB Giao thông vận tải, 2006
- [2] **Erik brown**, *Windows Forms Programming with C#*, Manning Publications Co., 2002
- [3] **Eric Gunnerson**, *A Programmer's Introduction to C#*, Apress publisher, 2000
- [4] **Jack Xu, Ph.D**, *Practical C# Charts and Graphics*, UniCAD Publishing, 2007
- [5] **James W. Cooper**, *C# Design Patterns*, Addison-Wesley publisher, 2006
- [6] **Joseph Albahari**, *C# 4.0 in a Nutshell*, O'Reilly Media Inc., 2006
- [7] **Matthew MacDonald**, *Pro .NET 2.0 Windows Forms and Custom Controls in C#*, Apress publisher, 2006
- [8] Một số tài liệu trên Internet

GIÁO TRÌNH
LẬP TRÌNH WINDOWS FORM
với **C#.Net**

TẬP 1

Chịu trách nhiệm xuất bản

NGUYỄN THỊ THU HÀ

Biên tập : NGUYỄN TIẾN SỸ
Trình bày sách : THANH HƯƠNG
Sửa bản in : NGUYỄN TIẾN SỸ
Thiết kế bìa : TRẦN HỒNG MINH

NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

Trụ sở chính: Số 9, ngõ 90, Ngụy Như Kon Tum, Thanh Xuân, Hà Nội

ĐT Biên tập: 04.35772143

ĐT Phát hành: 04.35772138

E-mail: nxb.tttt@mic.gov.vn

Fax: 04.35772037

Website: www.nxbthongtintruyenthong.vn

Chi nhánh TP. Hồ Chí Minh: 8A đường D2, Phường 25, Q. Bình Thạnh, TP. Hồ Chí Minh

Điện thoại: 08.35127750, 08.35127751 Fax: 08.35127751

E-mail: cnsg.nxbttt@mic.gov.vn

Chi nhánh TP. Đà Nẵng: 42 Trần Quốc Toản, quận Hải Châu, TP. Đà Nẵng

Điện thoại: 0511.3897467 Fax: 0511.3843359

E-mail: cndn.nxbttt@mic.gov.vn

In 1.000 bản, khổ 16×24 cm tại Công ty In Hải Nam
Số đăng ký kế hoạch xuất bản: 247-2012/CXB/1-46/TTTT
Số quyết định xuất bản: 202/QĐ-NXB TTTT ngày 07/8/2012
In xong nộp lưu chiểu tháng 8 năm 2012