**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**

 **KHOA CÔNG NGHỆ PHẦN MỀM**

**ĐỒ ÁN MÔN**

**CÔNG NGHỆ .NET**

**ĐỀ TÀI:**

**TÌM HIỂU VỀ ENTITYFRAMEWORK**

Giảng viên hướng dẫn: ThS. Phạm Thi Vương

Sinh viên thực hiện: Nguyễn Văn Quang 13520675

Bùi Đình Lộc Thọ 13520844

Đoàn Duy Phương 13520

Nguyễn Thanh Hải 13520

***TP. Hồ Chí Minh, ngày 21 tháng 06 năm 2017***

**Lời Cảm Ơn**

Đầu tiên, nhóm chúng em xin gởi lời cảm ơn chân thành đến tập thể quý Thầy Cô Trường Đại học Công nghệ thông tin – Đại học Quốc gia TP.HCM và quý Thầy Cô khoa Công Nghệ Phần Mềm đã giúp cho nhóm chúng em có những kiến thức cơ bản làm nền tảng để thực hiện đề tài này.

Đặc biệt, nhóm chúng em xin gửi lời cảm ơn và lòng biết ơn sâu sắc nhất tới Thầy Phạm Thi Vương đã giúp nhóm chúng em hoàn thành tốt báo cáo môn học của mình.

Trong thời gian một học kỳ thực hiện đề tài, nhóm chúng em đã vận dụng những kiến thức nền tảng đã tích lũy đồng thời kết hợp với việc học hỏi và nghiên cứu những kiến thức mới. Từ đó, nhóm chúng em vận dụng tối đa những gì đã thu thập được để hoàn thành một báo cáo đồ án tốt nhất. Tuy nhiên, trong quá trình thực hiện, nhóm chúng em không tránh khỏi những thiếu sót. Chính vì vậy, nhóm chúng em rất mong nhận được những sự góp ý từ phía các Thầy Cô nhằm hoàn thiện những kiến thức mà nhóm chúng em đã học tập và là hành trang để nhóm chúng em thực hiện tiếp các đề tài khác trong tương lai.

Nhóm em xin chân thành cảm ơn quý Thầy Cô!

NHẬN XÉT CỦA GIẢNG VIÊN

………………………………………………………………………………………………....

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………

…………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………

**Mục Lục**

[Chương 1. Ngôn ngữ C# 6](#_Toc485720283)

[1.1 Tổng quan về ngôn ngữ C# 6](#_Toc485720284)

[1.1.1 Sự ra đời của C# 6](#_Toc485720285)

[1.1.2 C# Là ngôn ngữ đơn giản 6](#_Toc485720286)

[1.1.3 C# Là ngôn ngữ hiện đại 7](#_Toc485720287)

[1.1.4 C# Là ngôn ngữ hướng đối tượng 7](#_Toc485720288)

[1.1.5 C# Là ngôn ngữ có ít từ khóa. 7](#_Toc485720289)

[1.1.6 C# Là ngôn ngữ hướng Module. 8](#_Toc485720290)

[1.1.7 C# Là ngôn ngữ sẽ phổ biến trong tương lai. 8](#_Toc485720291)

[1.2 So sánh với ngôn ngữ khác 9](#_Toc485720292)

[1.3 Nền tảng ngôn ngữ C# 11](#_Toc485720293)

[1.3.1 Kiểu dữ liệu 11](#_Toc485720294)

[1.3.2 Kiểu dữ liệu xây dựng sẵn 12](#_Toc485720295)

[1.3.3 Chọn kiểu dữ liệu 13](#_Toc485720296)

[1.3.4 Chuyển đổi kiểu dữ liệu 15](#_Toc485720297)

[1.3.5 Biến (Variable) 15](#_Toc485720298)

[1.3.6 Hằng (Constant) 16](#_Toc485720299)

[1.3.7 Kiểu liệt kê 16](#_Toc485720300)

[1.3.8 Kiểu chuỗi kí tự 17](#_Toc485720301)

[1.3.9 Định danh 17](#_Toc485720302)

[1.3.10 Biểu thức 18](#_Toc485720303)

[1.4 Khoảng trắng - namespace 19](#_Toc485720304)

[1.4.1 Khoảng trắng 19](#_Toc485720305)

[1.4.2 Namespace 20](#_Toc485720306)

[1.5 Câu lệnh (statement ) 20](#_Toc485720307)

[1.6 Toán tử 21](#_Toc485720308)

[1.6.1 Toán tử gán 21](#_Toc485720309)

[1.6.2 Toán tử toán học 21](#_Toc485720310)

[1.6.3 Toán tử tăng và giảm 22](#_Toc485720311)

[1.6.4 Toán tử quan hệ 23](#_Toc485720312)

[1.6.5 Toán tử logic 24](#_Toc485720313)

[1.6.6 Độ ưu tiên toán tử 24](#_Toc485720314)

[1.6.7 Toán tử ba ngôi 25](#_Toc485720315)

[1.7 Lớp – Đối tượng 26](#_Toc485720316)

[1.7.1 Định nghĩa lớp 26](#_Toc485720317)

[1.7.2 Tạo đối tượng 27](#_Toc485720318)

[1.7.3 Hàm khởi tạo (Constructor): 27](#_Toc485720319)

[1.7.4 Hàm Huỷ (Destructor) 28](#_Toc485720320)

[1.7.5 Khởi tạo biến thành viên 29](#_Toc485720321)

[1.7.6 Bộ khởi tạo sao chép 29](#_Toc485720322)

[1.7.7 Từ khóa this 29](#_Toc485720323)

[1.7.8 Thành viên static 30](#_Toc485720324)

[1.8 Hủy đối tượng (Garbage Collection) 31](#_Toc485720325)

[1.9 Kế thừa 32](#_Toc485720326)

[1.9.1 Thực thi kế thừa 32](#_Toc485720327)

[1.9.2 Gọi phương thức khởi dựng của lớp cơ sở 32](#_Toc485720328)

[1.9.3 Điều khiển truy xuất 33](#_Toc485720329)

[1.10 Đa hình 34](#_Toc485720330)

[1.11 Lớp trừu tượng 36](#_Toc485720331)

[1.12 Boxing và Unboxing dữ liệu 37](#_Toc485720332)

[1.13 Nạp chồng toán tử 38](#_Toc485720333)

[1.13.2 Sử dụng toán tử 39](#_Toc485720334)

[1.13.3 Toán tử so sánh bằng 42](#_Toc485720335)

[1.14 Cấu trúc 43](#_Toc485720336)

[1.14.1 Định nghĩa cấu trúc 43](#_Toc485720337)

[1.14.2 Tạo cấu trúc 44](#_Toc485720338)

[Chương 2 : Nền tảng .NET 45](#_Toc485720339)

[2.1 Tổng quan .Net framework 45](#_Toc485720340)

[2.2 Lịch sử phát triển 46](#_Toc485720341)

[Chương 3. Entity Framework 48](#_Toc485720342)

[3.1 Tổng quan về Entity Framework 48](#_Toc485720343)

[3.1.1 Giới thiệu 48](#_Toc485720344)

[3.1.2 Kiến trúc của Entity Framework 50](#_Toc485720345)

[3.2 Giới thiệu về Entity Data Model 51](#_Toc485720346)

[3.3 Mô hình MVC 53](#_Toc485720347)

[3.3.1 Giới thiệu về mô hình MVC 53](#_Toc485720348)

[3.3.2 Các thành phần trong MVC 54](#_Toc485720349)

[3.3.3 Sự tương tác giữa các lớp trong mô hình MVC 54](#_Toc485720350)

[3.3.4 Ưu điểm 55](#_Toc485720351)

[3.3.5 Nhược điểm 56](#_Toc485720352)

[3.3.6 Liên hệ MVC và Entity Framework 57](#_Toc485720353)

[Chương 4. 59](#_Toc485720354)

[Chương 5. Phát triển ứng dụng với Entity Framework 60](#_Toc485720355)

[5.1. Code first 60](#_Toc485720356)

[5.2. Model first 71](#_Toc485720357)

[5.3. Database first 77](#_Toc485720358)

# Chương 1. Ngôn ngữ C#

## 1.1 Tổng quan về ngôn ngữ C#

### 1.1.1 Sự ra đời của C#

Nhiều người tin rằng không cần thiết có một ngôn ngữ lập trình mới. Java, C++, Perl, Microsoft Visual Basic, và những ngôn ngữ khác được nghĩ rằng đã cung cấp tất cả những chức năng cần thiết. Ngôn ngữ C# là một ngôn ngữ được dẫn xuất từ C và C++, nhưng nó được tạo từ nền tảng phát triển hơn. Microsoft bắt đầu với công việc trong C 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. Không dừng lại ở đó, Microsoft đưa ra một số mục đích khi xây dựng ngôn ngữ này.

### 1.1.2 C# Là ngôn ngữ đơn giản

C# loại bỏ một vài sự phức tạp và rối rắm của những ngôn ngữ như Java và C++, bao gồm việc loại bỏ những macro, những template, đa kế thừa, và lớp cơ sở ảo (virtual base class). Chúng là những nguyên nhân gây ra sự nhầm lẫn hay dẫn đến những vấn đề cho các người phát triển C++. Nếu chúng ta là người học ngôn ngữ này đầu tiên thì chắc chắn là ta sẽ không trải qua những thời gian để học nó! Nhưng khi đó ta sẽ không biết được hiệu quả của ngôn ngữ C# khi loại bỏ những vấn đề trên.

Ngôn ngữ C# đơn giản vì nó dựa trên nền tảng C và C++. Nếu chúng ta thân thiện với C và C++ hoậc thậm chí là Java, chúng ta sẽ thấy C# khá giống về diện mạo, cú pháp, biểu thức, toán tử và những chức năng khác được lấy trực tiếp từ ngôn ngữ C và C++, nhưng nó đã được cải tiến để làm cho ngôn ngữ đơn giản hơn. Một vài trong các sự cải tiến là loại bỏ các dư thừa, hay là thêm vào những cú pháp thay đổi. Ví dụ như, trong C++ có ba toán tử làm việc với các thành viên là ::, . , và ->. Để biết khi nào dùng ba toán tử này cũng phức tạp và dễ nhầm lẫn. Trong C#, chúng được thay thế với một toán tử duy nhất gọi là . (dot). Đối với người mới học thì điều này và những việc cải tiến khác làm bớt nhầm lẫn và đơn giản hơn.

### 1.1.3 C# Là ngôn ngữ hiện đại

Điều gì làm cho một ngôn ngữ hiện đại? Những đặc tính như là xử lý ngoại lệ, thu gom bộ nhớ tự động, những kiểu dữ liệu mở rộng, và bảo mật mã nguồn là những đặc tính được mong đợi trong một ngôn ngữ hiện đại. C# chứa tất cả những đặc tính trên. Nếu là người mới học lập trình có thể chúng ta sẽ cảm thấy những đặc tính trên phức tạp và khó hiểu. Tuy nhiên, cũng đừng lo lắng chúng ta sẽ dần dần được tìm hiểu những đặc tính qua các chương trong cuốn sách này.

**Ghi chú:** Con trỏ được tích hợp vào ngôn ngữ C++. Chúng cũng là nguyên nhân gây ra những rắc rối của ngôn ngữ này. C# loại bỏ những phức tạp và rắc rối phát sinh bởi con trỏ. 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++.

### 1.1.4 C# Là ngôn ngữ hướng đối tượng

Những đặc điểm chính của ngôn ngữ hướng đối tượng (**Object - Oriented Language**) là sự đóng gói (**Encapsulation**), sự kế thừa (**Inheritance**), và đa hình (**Polymorphism**). C# hỗ trợ tất cả những đặc tính trên. Phần hướng đối tượng của C# sẽ được trình bày chi tiết trong một chương riêng ở phần sau.

### 1.1.5 C# Là ngôn ngữ có ít từ khóa.

C# là ngôn ngữ sử dụng giới hạn những từ khóa. Phần lớn các từ khóa được sử dụng để mô tả thông tin. Chúng ta có thể nghĩ rằng một ngôn ngữ có nhiều từ khóa thì sẽ mạnh hơn. Điều này không phải sự thật, ít nhất là trong trường hợp ngôn ngữ C#, chúng ta có thể tìm thấy rằng ngôn ngữ này có thể được sử dụng để làm bất cứ nhiệm vụ nào. Bảng sau liệt kê các từ khóa của ngôn ngữ C#

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| abstract | default | foreach | object | sizeof | unsafe |
| as | delegate | goto | operator | stackalloc | ushort |
| base | do | if | out | static | using |
| bool | double | implicit | override | string | virtual |
| break | else | in | params | struct | volatile |
| byte | enum | int | private | switch | void |
| case | event | interface | protected | this | while |
| catch | explicit | internal | public | throw |  |
| char | extern | is | readonly | true |  |
| checked | false | lock | ref | try |  |
| class | finally | long | return | typeof |  |
| const | fixed | namespace | sbyte | uint |  |
| continue | float | new | sealed | ulong |  |
| decimal | for | null | short | unchecked |  |

### 1.1.6 C# Là ngôn ngữ hướng Module.

Mã nguồn C# có thể được viết trong những phần được gọi là những lớp, những lớp này chứa các phương thức thành viên của nó. Những lớp và những phương thức có thể được sử dụng lại trong ứng dụng hay các chương trình khác. Bằng cách truyền các mẫu thông tin đến những lớp hay phương thức chúng ta có thể tạo ra những mã nguồn dùng lại có hiệu quả.

### 1.1.7 C# Là ngôn ngữ sẽ phổ biến trong tương lai.

C# là một trong những ngôn ngữ lập trình mới nhất. Nhưng ngôn ngữ này có một số lý do để trở thành một ngôn ngữ phổ biến. Một trong những lý do chính là Microsoft và sự cam kết của .NET.

Microsoft muốn ngôn ngữ C# trở nên phổ biến. Mặc dù một công ty không thể làm một sản phẩm trở nên phổ biến, nhưng nó có thể hỗ trợ. Cách đây không lâu, Microsoft đã gặp sự thất bại về hệ điều hành Microsoft Bob. Mặc dù Microsoft muốn Bob trở nên phổ biến nhưng thất bại. C# thay thế tốt hơn để đem đến thành công sơ với Bob. Thật sự là không biết khi nào mọi người trong công ty Microsoft sử dụng Bob trong công việc hằng ngày của họ. Tuy nhên, với C# thì khác, nó được sử dụng bởi Microsoft. Nhiều sản phẩm của công ty này đã chuyển đổi và viết lại bằng C#. Bằng cách sử dụng ngôn ngữ này Microsoft đã xác nhận khả năng của C# cần thiết cho những người lập trình.

Micorosoft .NET là một lý do khác để đem đến sự thành công của C#. .NET là một sự thay đổi trong cách tạo và thực thi những ứng dụng. Ngoài hai lý do trên ngôn ngữ C# cũng sẽ trở nên phổ biến do những đặc tính của ngôn ngữ này được đề cập trong mục trước như: đơn giản, hướng đối tượng, mạnh mẽ...

## 1.2 So sánh với ngôn ngữ khác

Microsoft nói rằng C# mang đến sức mạnh của ngôn ngữ C++ với sự dễ dàng của ngôn ngữ Visual Basic. Có thể nó không dễ như Visual Basic, nhưng với phiên bản Visual Basic.NET (Version 7) thì ngang nhau. Bởi vì chúng được viết lại từ một nền tảng. Chúng ta có thể viết nhiều chương trình với ít mã nguồn hơn nếu dùng C#.

Mặc dù C# loại bỏ một vài các đặc tính của C++, nhưng bù lại nó tránh được những lỗi mà thường gặp trong ngôn ngữ C++. Điều này có thể tiết kiệm được hàng giờ hay thậm chí hàng ngày trong việc hoàn tất một chương trình. Chúng ta sẽ hiểu nhiều về điều này trong các chương của giáo trình.

Một điều quan trọng khác với C++ là mã nguồn C# không đòi hỏi phải có tập tin header. Tất cả mã nguồn được viết trong khai báo một lớp. Như đã nói ở bên trên. .NET runtime trong C# thực hiện việc thu gom bộ nhớ tự động. Do điều này nên việc sử dụng con trỏ trong C# ít quan trọng hơn trong C++. Những 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).

C# cũng từ bỏ ý tưởng đa kế thừa như trong C++. Và sự khác nhau khác là C# đưa thêm thuộc tính vào trong một lớp giống như trong Visual Basic. Và những thành viên của lớp được gọi duy nhất bằng toán tử “.” khác với C++ có nhiều cách gọi trong các tình huống khác nhau.

Một ngôn ngữ khác rất mạnh và phổ biến là Java, giống như C++ và C# được phát triển dựa trên C. Nếu chúng ta quyết định sẽ học Java sau này, chúng ta sẽ tìm được nhiều cái mà học từ C# có thể được áp dụng.

Điểm giống nhau 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 Just – In – Time 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 bản hơn Java và cũng cho phép nhiều sự mở rộng với kiểu dữ liệu giá trị. Ví dụ, ngôn ngữ C# hỗ trợ kiểu liệt kệ (enumerator), kiểu này được giới hạn đến một tập hằng được định nghĩa trước, và kiểu dữ liệu cấu trúc đây là kiểu dữ liệu giá trị do người dùng định nghĩa. Chúng ta sẽ được tìm hiểu kỹ hơn về kiểu dữ liệu tham chiếu và kiểu dữ liệu giá trị sẽ được trình bày trong phần sau. Tương tự như Java, C# cũng từ bỏ tính đa kế thừa trong một lớp, tuy nhiên mô hình kế thừa đơn này được mở rộng bởi tính đa kế thừa nhiều Interface.

## 1.3 Nền tảng ngôn ngữ C#

### 1.3.1 Kiểu dữ liệu

C# là ngôn ngữ lập trình mạnh về kiểu dữ liệu, một ngôn ngữ mạnh về kiểu dữ liệu là phải khai báo kiểu của mỗi đối tượng khi tạo (kiểu số nguyên, số thực, kiểu chuỗi, kiểu điều khiển...) và trình biên dịch sẽ giúp cho người lập trình không bị lỗi khi chỉ cho phép một loại kiểu dữ liệu có thể được gán cho các kiểu dữ liệu khác. Kiểu dữ liệu của một đối tượng là một tín hiệu để trình biên dịch nhận biết kích thước của một đối tượng (kiểu int có kích thước là 4 byte) và khả năng của nó (như một đối tượng button có thể vẽ, phản ứng khi nhấn,...).

Tương tự như C++ hay Java, C# chia thành hai tập hợp kiểu dữ liệu chính: Kiểu xây dựng sẵn (built- in) mà ngôn ngữ cung cấp cho người lập trình và kiểu được người dùng định nghĩa (user-defined) do người lập trình tạo ra.

C# phân tập hợp kiểu dữ liệu này thành hai loại: Kiểu dữ liệu giá trị (value) và kiểu dữ liệu tham chiếu (reference). Việc phân chi này do sự khác nhau khi lưu kiểu dữ liệu giá trị và kiểu dữ liệu tham chiếu trong bộ nhớ. Đối với một kiểu dữ liệu giá trị thì sẽ được lưu giữ kích thước thật trong bộ nhớ đã cấp phát là stack. Trong khi đó thì địa chỉ của kiểu dữ liệu tham chiếu thì được lưu trong stack nhưng đối tượng thật sự thì lưu trong bộ nhớ heap.

**Ghi chú:** Tất cả các kiểu dữ liệu xây dựng sẵn là kiểu dữ liệu giá trị ngoại trừ các đối tượng và chuỗi. Và tất cả các kiểu do người dùng định nghĩa ngoại trừ kiểu cấu trúc đều là kiểu dữ liệu tham chiếu.

Ngoài ra C# cũng hỗ trợ một kiểu con trỏ C++, nhưng hiếm khi được sử dụng, và chỉ khi nào làm việc với những đoạn mã lệnh không được quản lý (unmanaged code). Mã lệnh không được quản lý là các lệnh được viết bên ngoài nền.MS.NET, như là các đối tượng COM.

### 1.3.2 Kiểu dữ liệu xây dựng sẵn

Ngôn ngữ C# đưa ra các kiểu dữ liệu xây dựng sẵn rất hữu dụng, phù hợp với một ngôn ngữ lập trình hiện đại, mỗi kiểu dữ liệu được ánh xạ đến một kiểu dữ liệu được hỗ trợ bởi hệ thống xác nhận ngôn ngữ chung (Common Language Specification: CLS) trong MS.NET. Việc ánh xạ các kiểu dữ liệu nguyên thuỷ 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 cứ ngôn ngữ khác được biên dịch bởi .NET, như VB.NET.

Mỗi kiểu dữ liệu có một sự xác nhận và kích thước không thay đổi, không giống như C++, int trong C# luôn có kích thước là 4 byte bởi vì nó được ánh xạ từ kiểu Int32 trong .NET.

Bảng sau sẽ mô tả một số các kiểu dữ liệu được xây dựng sẵn

|  |  |  |  |
| --- | --- | --- | --- |
| **Kiểu C#** | **Số byte** | **Kiểu .NET** | **Mô tả** |
| byte | 1 | Byte | Số nguyên dương không dấu từ 0-255 |
| char | 2 | Char | Ký tự Unicode |
| bool | 1 | Boolean | Giá trị logic true/ false |
| sbyte | 1 | Sbyte | Số nguyên có dấu ( từ -128 đến 127) |
| short | 2 | Int16 | Số nguyên có dấu giá trị từ -32768 đến 32767. |
| ushort | 2 | Uịnt16 | Số nguyên không dấu 0 – 65.535 |
| int | 4 | Int32 | Số nguyên có dấu –2.147.483.647 và 2.147.483.647 |
| uint | 4 | Uint32 | Số nguyên không dấu 0 – 4.294.967.295 |
| float | 4 | Single | Kiểu dấu chấm động, giá trị xấp xỉ từ 3,4E- 38 đến 3,4E+38, với 7 chữ số có nghĩa.. |
| double | 8 | Double | Kiểu dấu chấm động có độ chính xác gấp đôi, giá trị xấp xỉ từ 1,7E-308 đến 1,7E+308, với 15,16 chữ số có nghĩa. |
| decimal | 8 | Decimal | Có độ chính xác đến 28 con số và giá trị thập phân, được dùng trong tính toán tài chính, kiểu này đòi hỏi phải có hậu tố “m” hay “M” theo sau giá trị. |

### 1.3.3 Chọn kiểu dữ liệu

Thông thường để chọn một kiểu dữ liệu nguyên để sử dụng như short, int hay long thường dựa vào độ lớn của giá trị muốn sử dụng. Ví dụ, một biến ushort có thể lưu giữ giá trị từ 0 đến 65.535, trong khi biến ulong có thể lưu giữ giá trị từ 0 đến 4.294.967.295, do đó tùy vào miền giá trị của phạm vi sử dụng biến mà chọn các kiểu dữ liệu thích hợp nhất. Kiểu dữ liệu int thường được sử dụng nhiều nhất trong lập trình vì với kích thước 4 byte của nó cũng đủ để lưu các giá trị nguyên cần thiết.

Kiểu số nguyên có dấu thường được lựa chọn sử dụng nhiều nhất trong kiểu số trừ khi có lý do chính đáng để sử dụng kiểu dữ liệu không dấu.

|  |
| --- |
| **Stack và Heap** |
| Stack là một cấu trúc dữ liệu lưu trữ thông tin dạng xếp chồng tức là vào sau ra trước (Last In First Out : LIFO), điều này giống như chúng ta có một chồng các đĩa, ta cứ xếp các đĩa vào chồng và khi lấy ra thì đĩa nào nằm trên cùng sẽ được lập ra trước, tức là đĩa vào sau sẽ được lấy ra trước. Trong C#, kiểu giá trị như kiểu số nguyên được cấp phát trên stack, đây là vùng nhớ được thiết lập để lưu các giá trị, và vùng nhớ này được tham chiếu bởi tên của biến. Kiểu tham chiếu như các đối tượng thì được cấp phát trên heap. Khi một đối tượng được cấp phát trên heap thì địa chỉ của nó được trả về, và địa chỉ này được gán đến một tham chiếu. Thỉnh thoảng cơ chế thu gom sẽ hũy đối tượng trong stack sau khi một vùng trong stack được đánh dấu là kết thúc. Thông thường một vùng trong stack được định nghĩa bởi một hàm. Do đó, nếu chúng ta khai báo một biến cục bộ trong một hàm là một đối tượng thì đối tượng này sẽ đánh dấu để hũy khi kết thúc hàm. Những đối tượng trên heap sẽ được thu gom sau khi một tham chiếu cuối cùng đến đối tượng đó được gọi |

Cách tốt nhất khi sử dụng biến không dấu là giá trị của biến luôn luôn dương, biến này thường thể hiện một thuộc tính nào đó có miền giá trị dương. Ví dụ khi cần khai báo một biến lưu giữ tuổi của một người thì ta dùng kiểu byte (số nguyên từ 0 -255) vì tuổi của người không thể nào âm được.

Kiểu float, double, và decimal đưa ra nhiều mức độ khác nhau về kích thước cũng như độ chính xác.Với thao tác trên các phân số nhỏ thì kiểu float là thích hợp nhất. Tuy nhiên lưu ý rằng trình biên dịch luôn luôn hiểu bất cứ một số thực nào cũng là một số kiểu double trừ khi chúng ta khai báo rõ ràng. Để gán một số kiểu float thì số phải có ký tự f theo sau.

Kiểu dữ liệu ký tự thể hiện các ký tự Unicode, bao gồm các ký tự đơn giản, ký tự theo mã Unicode và các ký tự thoát khác được bao trong những dấu nháy đơn. Ví dụ, A là một ký tự đơn giản trong khi \u0041 là một ký tự Unicode. Ký tự thoát là những ký tự đặc biệt bao gồm hai ký tự liên tiếp trong đó ký tự dầu tiên là dấu chéo ‘\’. Ví dụ, \t là dấu tab. Bảng sau trình bày các ký tự đặc biệt.

|  |  |
| --- | --- |
| **Ký tự** | **Ý nghĩa** |
| \’ | Dấu nháy đơn |
| \” | Dấu nháy kép |
| \\ | Dấu chéo |
| \0 | Ký tự null |
| \a | Alert |
| \b | Backspace |
| \f | Sang trang form feed |
| \n | Dòng mới |
| \r | Đầu dòng |
| \t | Tab ngang |
| \v | Tab dọc |
|  |  |

### 1.3.4 Chuyển đổi kiểu dữ liệu

Những đối tượng của một kiểu dữ liệu này có thể được chuyển sang những đối tượng của một kiểu dữ liệu khác thông qua cơ chế chuyển đổi tường minh hay ngầm định. Chuyển đổi nhầm định được thực hiện một cách tự động, trình biên dịch sẽ thực hiện công việc này. Còn chuyển đổi tường minh diễn ra khi chúng ta gán ép một giá trị cho kiểu dữ liệu khác. Việc chuyển đổi giá trị ngầm định được thực hiện một cách tự động và đảm bảo là không mất thông tin. Ví dụ, chúng ta có thể gán ngầm định một số kiểu short (2 byte) vào một số kiểu int (4 byte) một cách ngầm định. Sau khi gán hoàn toàn không mất dữ liệu vì bất cứ giá trị nào của short cũng thuộc về int.Tuy nhiên, nếu chúng ta thực hiện chuyển đổi ngược lại, chắc chắn chúng ta sẽ bị mất thông tin. Nếu giá trị của số nguyên đó lớn hơn 32.767 thì nó sẽ bị cắt khi chuyển đổi. Trình biên dịch sẽ không thực hiện việc chuyển đổi ngầm định từ số kiểu int sang số kiểu short.

### 1.3.5 Biến (Variable)

Một biến là một vùng lưu trữ với một kiểu dữ liệu. Trong ví dụ trước cả x, và y điều là biến. Biến có thể được gán giá trị và cũng có thể thay đổi giá trị khi thực hiện các lệnh trong chương trình. Để tạo một biến chúng ta phải khai báo kiểu của biến và gán cho biến một tên duy nhất. Biếncó thể được khởi tạo giá trị ngay khi được khai báo, hay nó cũng có thể được gán một giá trị mới vào bất cứ lúc nào trong chương trình. C# đòi hỏi các biến phải được khởi tạo trước khi được sử dụng, tuy nhiên không nhất thiết lúc nào chúng ta cũng phải khởi tạo biến. Nhưng để dùng được thì bắt buộc phải gán cho chúng một giá trị trước khi có một lệnh nào tham chiếu đến biến đó. Điều này được gọi là gán giá trị xác định cho biến và C# bắt buộc phải thực hiện điều này.

### 1.3.6 Hằng (Constant)

Hằng cũng là một biến nhưng giá trị của hằng không thay đổi. Biến là công cụ rất mạnh, tuy nhiên khi làm việc với một giá trị được định nghĩa là không thay đổi, ta phải đảm bảo giá trị của nó không được thay đổi trong suốt chương trình. Ví dụ, khi lập một chương trình thí nghiệm hóa học liên quan đến nhiệt độ sôi, hay nhiệt độ đông của nước, chương trình cần khai báo hai biến là DoSoi và DoDong, nhưng không cho phép giá trị của hai biến này bị thay đổi hay bị gán. Để ngăn ngừa việc gán giá trị khác, ta phải sử dụng biến kiểu hằng. Hằng được phân thành ba loại: giá trị hằng (Literal), biểu tượng hằng (Symbolic Constants), kiểu liệu kê (Enumerations).

### 1.3.7 Kiểu liệt kê

Kiểu liệt kê đơn giản là tập hợp các tên hằng có giá trị không thay đổi (thường được gọi là danh sách liệt kê).

Mỗi kiểu liệt kê có một kiểu dữ liệu cơ sở, kiểu dữ liệu có thể là bất cứ kiểu dữ liệu nguyên nào như int, short, long... tuy nhiên kiểu dữ lịêu của liệt kê không chấp nhận kiểu ký tự. Để khai báo một kiểu liệt kê ta thực hiện theo cú pháp sau:  
 [thuộc tính] [bổ sung] enum <tên liệt kê> [:kiểu cơ sở] {danh sách các thành phần liệt kê};

Thành phần thuộc tính và bổ sung là tự chọn sẽ được trình bày trong phần sau của sách.  
Trong phần này chúng ta sẽ tập trung vào phần còn lại của khai báo. Một kiểu liệt kê bắt đầu với từ khóa **enum**, tiếp sau là một định danh cho kiểu liệt kê:

enum NhietDoNuoc

Thành phần kiểu cơ sở chính là kiểu khai báo cho các mục trong kiểu liệt kê. Nếu bỏ qua thành phần này thì trình biên dịch sẽ gán giá trị mặc định là kiểu nguyên int, tuy nhiên chúng ta có thể sử dụng bất cứ kiểu nguyên nào như ushort hay long,..ngoại trừ kiểu ký tự.

### 1.3.8 Kiểu chuỗi kí tự

Kiểu dữ liệu chuỗi khá thân thiện với người lập trình trong bất cứ ngôn ngữ lập trình nào, kiểu dữ liệu chuỗi lưu giữ một mảng những ký tự. Để khai báo một chuỗi chúng ta sử dụng từ khoá string tương tự như cách tạo một thể hiện của bất cứ đối tượng nào:

string chuoi;

Một hằng chuỗi được tạo bằng cách đặt các chuỗi trong dấu nháy đôi:  
“Xin chao” Đây là cách chung để khởi tạo một chuỗi ký tự với giá trị hằng:  
string chuoi = “Xin chao”.

### 1.3.9 Định danh

Định danh là tên mà người lập trình chỉ định cho các kiểu dữ liệu, các phương thức, biến, hằng, hay đối tượng.... Một định danh phải bắt đầu với một ký tự chữ cái hay dấu gạch dưới, các ký tự còn lại phải là ký tự chữ cái, chữ số, dấu gạch dưới.Theo qui ước đặt tên của Microsoft thì đề nghị sử dụng cú pháp lạc đà (Camel Notation) bắt đầu bằng ký tự thường để đặt tên cho các biến là cú pháp Pascal (Pascal Notation) với ký tự đầu tiên hoa cho cách đặt tên hàm và hầu hết các định danh còn lại. Hầu như Microsoft không còn dùng cú pháp Hungary như iSoNguyen hay dấu gạch dưới Bien\_Nguyen để đặt các định danh.

Các định danh không được trùng với các từ khoá mà C# đưa ra, do đó chúng ta không thể tạo các biến có tên như class hay int được. Ngoài ra, C# cũng phân biệt các ký tự thường và ký tự hoa vì vậy C# xem hai biến bienNguyen và bienguyen là hoàn toàn khác nhau.

### 1.3.10 Biểu thức

Những câu lệnh mà thực hiện việc đánh giá một giá trị gọi là biểu thức. Một phép gán một giá trị cho một biến cũng là một biểu thức:

var1 = 24;

Trong câu lệnh trên phép đánh giá hay định lượng chính là phép gán có giá trị là 24 cho biến var1. Lưu ý là toán tử gán (‘=’) không phải là toán tử so sánh. Do vậy khi sử dụng toán tử này thì biến bên trái sẽ nhận giá trị của phần bên phải. Các toán tử của ngôn ngữ C# như phép so sánh hay phép gán sẽ được trình bày chi tiết trong mục toán tử của chương này. Do var1 = 24 là một biểu thức được định giá trị là 24 nên biểu thức này có thể được xem như phần bên phải của một biểu thức gán khác:

var2 = var1 = 24;

Lệnh này sẽ được thực hiện từ bên phải sang khi đó biến var1 sẽ nhận được giá trị là 24 và tiếp sau đó thì var2 cũng được nhận giá trị là 24. Do vậy cả hai biến đều cùng nhận một giá trị là 24. Có thể dùng lệnh trên để khởi tạo nhiều biến có cùng một giá trị như:  
a = b = c = d = 24;

## 1.4 Khoảng trắng - namespace

### 1.4.1 Khoảng trắng

Trong ngôn ngữ C#, những khoảng trắng, khoảng tab và các dòng được xem như là khoảng trắng (whitespace), giống như tên gọi vì chỉ xuất hiện những khoảng trắng để đại diện cho các ký tự đó. C# sẽ bỏ qua tất cả các khoảng trắng đó, do vậy chúng ta có thể viết như sau:

var1 = 24; hay var1 = 24 ;

và trình biên dịch C# sẽ xem hai câu lệnh trên là hoàn toàn giống nhau. Tuy nhiên lưu ý là khoảng trắng trong một chuỗi sẽ không được bỏ qua. Nếu chúng ta viết:

System.WriteLine(“Xin chao!”);

mỗi khoảng trắng ở giữa hai chữ “Xin” và “chao” đều được đối xử bình thường như các ký tự khác trong chuỗi.

Hầu hết việc sử dụng khoảng trắng như một sự tùy ý của người lập trình. Điều cốt yếu là việc sử dụng khoảng trắng sẽ làm cho chương trình dễ nhìn dễ đọc hơn Cũng như khi ta viết một văn bản trong MS Word nếu không trình bày tốt thì sẽ khó đọc và gây mất cảm tình cho người xem. Còn đối với trình biên dịch thì việc dùng hay không dùng khoảng trắng là không khác nhau. Tuy nhiên, củng cần lưu ý khi sử dụng khoảng trắng như sau:

int x = 24;

tương tự như: int x=24;

nhưng không giống như: intx=24;

trình biên dịch nhận biết được các khoảng trắng ở hai bên của phép gán là phụ và có thể bỏ qua, nhưng khoảng trắng giữa khai báo kiểu và tên biến thì không phải phụ hay thêm mà bắt buộc phải có tối thiểu một khoảng trắng. Điều này không có gì bất hợp lý, vì khoảng trắng cho phép trình biên dịch nhận biết được từ khoá int và không thể nào nhận được intx. Tương tự như C/C++, trong C# câu lệnh được kết thúc với dấu chấm phẩy ‘;’. Do vậy có thể một câu lệnh trên nhiều dòng, và một dòng có thể nhiều câu lệnh nhưng nhất thiết là hai câu lệnh phải cách nhau một dấu chấm phẩy.

### 1.4.2 Namespace

Nhằm tránh sự xung đột giữa việc sử dụng các thư viện khác nhau từ các nhà cung cấp. Ngoài ra, namespace được xem như là tập hợp các lớp đối tượng, và cung cấp duy nhất các định danh cho các kiểu dữ liệu và được đặt trong một cấu trúc phân cấp. Việc sử dụng namespace trong khi lập trình là một thói quen tốt, bởi vì công việc này chính là cách lưu các mã nguồn để sử dụng về sau. Ngoài thư viện namespace do MS.NET và các hãng thứ ba cung cấp, ta có thể tạo riêng cho mình các namespace. C# đưa ra từ khóa **using** đề khai báo sử dụng namespace trong chương trình:

using < Tên namespace >  
Để tạo một namespace dùng cú pháp sau:  
namespace <Tên namespace>  
{  
< Định nghĩa lớp A>  
< Định nghĩa lớp B >  
.....  
}

## 1.5 Câu lệnh (statement )

Trong C# một chỉ dẫn lập trình đầy đủ được gọi là câu lệnh. Chương trình bao gồm nhiều câu lệnh tuần tự với nhau. Mỗi câu lệnh phải kết thúc với một dấu chấm phẩy, ví dụ như:

int x; // một câu lệnh  
 x = 32; // câu lệnh khác  
 int y =x; // đây cũng là một câu lệnh

những câu lệnh này sẽ được xử lý theo thứ tự. Đầu tiên trình biên dịch bắt đầu ở vị trí đầu của danh sách các câu lệnh và lần lượt đi từng câu lệnh cho đến lệnh cuối cùng, tuy nhiên chỉ đúng cho trường hợp các câu lệnh tuần tự không phân nhánh. Có hai loại câu lệnh phân nhánh trong C# là : phân nhánh không có điều kiện (Unconditional Branching Statement) và phân nhánh có điều kiện (Conditional Branching Statement).  
Ngoài ra còn có các câu lệnh làm cho một số đoạn chương trình được thực hiện nhiều lần, các câu lệnh này được gọi là câu lệnh lặp hay vòng lặp. Bao gồm các lệnh lặp **for**, **while**, **do**, **in**, và **each** sẽ được đề cập tới trong mục tiếp theo.

## 1.6 Toán tử

Toán tử được kí hiệu bằng một biểu tượng dùng để thực hiện một hành động. Các kiểu dữ liệu cơ bản của C# như kiểu nguyên hỗ trợ rất nhiều các toán tử như toán tử gán, toán tử toán học, logic...

### 1.6.1 Toán tử gán

Đến lúc này toán tử gán khá quen thuộc với chúng ta, hầu hết các chương trình minh họa từ đầu sách đều đã sử dụng phép gán. Toán tử gán hay phép gán làm cho toán hạng bên trái thay đổi giá trị bằng với giá trị của toán hạng bên phải. Toán tử gán là toán tử hai ngôi. Đây là toán tử đơn giản nhất thông dụng nhất và cũng dễ sử dụng nhất.

### 1.6.2 Toán tử toán học

Ngôn ngữ C# cung cấp năm toán tử toán học, bao gồm bốn toán tử đầu các phép toán cơ bản. Toán tử cuối cùng là toán tử chia nguyên lấy phần dư. Chúng ta sẽ tìm hiểu chi tiết các phép toán này trong phần tiếp sau.

* **Các phép toán số học cơ bản (+,-,\*,/)** : Các phép toán này không thể thiếu trong bất cứ ngôn ngữ lập trình nào, C# cũng không ngoại lệ, các phép toán số học đơn giản nhưng rất cần thiết bao gồm: phép cộng (+), phép trừ (-), phép nhân (\*), phép chia (/) nguyên và không nguyên. Khi chia hai số nguyên, thì C# sẽ bỏ phần phân số, hay bỏ phần dư, tức là nếu ta chia 8/3 thì sẽ được kết quả là 2 và sẽ bỏ phần dư là 2, do vậy để lấy được phần dư này thì C# cung cấp thêm toán tử lấy dư sẽ được trình bày trong phần kế tiếp. Tuy nhiên, khi chia cho số thực có kiểu như float, double, hay decimal thì kết quả chia được trả về là một số thực.
* **Phép toán chia lấy dư :** Để tìm phần dư của phép chia nguyên, chúng ta sử dụng toán tử chia lấy dư (%). Ví dụ, câu lệnh sau 8%3 thì kết quả trả về là 2 (đây là phần dư còn lại của phép chia nguyên). Thật sự phép toán chia lấy dư rất hữu dụng cho người lập trình . Khi chúng ta thực hiện một phép chia dư n cho một số khác, nếu số này là bội số của n thì kết quả của phép chia dư là 0. Ví dụ 20 % 5 = 0 vì 20 là một bội số của 5. Điều này cho phép chúng ta ứng dụng trong vòng lặp, khi muốn thực hiện một công việc nào đó cách khoảng n lần, ta chỉ cần kiểm tra phép chia dư n, nếu kết quả bằng 0 thì thực hiện công việc.

### 1.6.3 Toán tử tăng và giảm

Khi sử dụng các biến số ta thường có thao tác là cộng một giá trị vào biến, trừ đi một giá trị từ biến đó, hay thực hiện các tính toán thay đổi giá trị của biến sau đó gán giá trị mới vừa tính toán cho chính biến đó.

|  |  |
| --- | --- |
| **Toán tử** | **Ý nghĩa** |
| += | Cộng thêm giá trị toán hạng bên phả vào giá trị toán hạng bên trái |
| -= | Toán hạng bên trái được trừ bớt đi mộ lượng bằng giá trị của toán hạng bên phải |
| \*= | Toán hạng bên trái được nhân với mộ lượng bằng giá trị của toán hạng bên phải. |
| /= | Toán hạng bên trái được chia với mộ lượng bằng giá trị của toán hạng bên phải. |
| %= | Toán hạng bên trái được chia lấy dư vớ một lượng bằng giá trị của toán hạng bên phải |

### 1.6.4 Toán tử quan hệ

Những toán tử quan hệ được dùng để so sánh giữa hai giá trị, và sau đó trả về kết quả là một giá trị logic kiểu bool (true hay false). Ví dụ toán tử so sánh lớn hơn (>) trả về giá trị là true nếu giá trị bên trái của toán tử lớn hơn giá trị bên phải của toán tử. Do vậy 5 > 2 trả về một giá trị là true, trong khi 2 > 5 trả về giá trị false.Các toán tử quan hệ trong ngôn ngữ C# được trình bày ở bảng bên dưới. Các toán tử trong bảng được minh họa với hai biến là value1 và value2, trong đó value1 có giá trị là 100 và value2 có giá trị là 50.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tên toán tử** | **Kí hiệu** | **Biểu thức so sánh** | **Kết quả so sánh** |
| So sánh bằng | == | value1 == 100 value1 == 50 | true false |
| Không bằng | != | value2 != 100 value2 != 90 | false true |
| Lớn hơn | > | value1 > value2 value2 > value1 | true false |
| Lớn hơn hay bằng | >= | value2 >= 50 | true |
| Nhỏ hơn | < | value1 < value2 value2 < value1 | false true |
| Nhỏ hơn hay bằng | <= | value1 <= value2 | false |

### 1.6.5 Toán tử logic

Trong câu lệnh **if** mà chúng ta đã tìm hiểu trong phần trước, thì khi điều kiện là true thì biểu thức bên trong **if** mới được thực hiện. Đôi khi chúng ta muốn kết hợp nhiều điều kiện với nhau như: bắt buộc cả hai hay nhiều điều kiện phải đúng hoặc chỉ cần một trong các điều kiện đúng là đủ hoặc không có điều kiện nào đúng...C# cung cấp một tập hợp các toán tử logic để phục vụ cho người lập trình.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tên toán tử** | **Ký hiệu** | **Biểu thức logic** | **Giá trị** | **Logic** |
| and | && | (x == 3) && (y == 7) | false | Cả hai điều kiện phải đúng |
| or | || | (x == 3) || (y == 7) | true | Chỉ cần một điều kiện đúng |
| not | ! | ! (x == 3 ) | true | Biểu thức trong ngoặc phải sai. |

### 1.6.6 Độ ưu tiên toán tử

Trình biên dịch phải xác định thứ tự thực hiện các toán tử trong trường hợp một biểu thức có nhiều phép toán, giả sử, có biểu thức sau:

var1 = 5+7\*3;

biểu thức trên có ba phép toán để thực hiện bao gồm (=, +,\*). Ta thử xét các phép toán theo thứ tự từ trái sang phải, đầu tiên là gán giá trị 5 cho biến var1, sau đó cộng 7 vào 5 là 12 cuối cùng là nhân với 3, kết quả trả về là 36, điều này thật sự có vấn đề, không đúng với mục đích yêu cầu của chúng ta. Do vậy việc xây dựng một trình tự xử lý các toán tử là hết sức cần thiết. Các luật về độ ưu tiên xử lý sẽ bảo trình biên dịch biết được toán tử nào được thực hiện trước trong biểu thức.

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Loại toán tử** | **Toán tử** | **Thứ tự** |
| 1 | Phép toán cơ bản | (x) x.y f(x) a[x] x++ x—new typeof sizeof checked unchecked | Trái |
| 2 | + - ! ~ ++x –x (T)x | Trái |  |
| 3 | Phép nhân | \* / % | Trái |
| 4 | Phép cộng | + - | Trái |
| 5 | Dịch bit | << >> | Trái |
| 6 | Quan hệ | < > <= >= is | Trái |
| 7 | So sánh bằng | == != | Phải |
| 8 | Phép toán logic AND | & | Trái |
| 9 | Phép toán logic XOR | ^ | Trái |
| 10 | Phép toán logic OR | | | Trái |
| 11 | Điều kiện AND | && | Trái |
| 12 | Điều kiện OR | || | Trái |
| 13 | Điều kiện | ?: | Phải |
| 14 | Phép gán | = \*= /= %= += -= <<= >>= &= ^= |= | Phải |

Các phép toán được liệt kê cùng loại sẽ có thứ tự theo mục thứ thự của bảng: thứ tự trái tức là độ ưu tiên của các phép toán từ bên trái sang, thứ tự phải thì các phép toán có độ ưu tiên từ bên phải qua trái. Các toán tử khác loại thì có độ ưu tiên từ trên xuống dưới, do vậy các toán tử loại cơ bản sẽ có độ ưu tiên cao nhất và phép toán gán sẽ có độ ưu tiên thấp nhất trong các toán tử.

### 1.6.7 Toán tử ba ngôi

Hầu hết các toán tử đòi hỏi có một toán hạng như toán tử (++, --) hay hai toán hạng như (+,-,\*,/,...). Tuy nhiên, C# còn cung cấp thêm một toán tử có ba toán hạng (?:). Toán tử này có cú pháp sử dụng như sau:

<Biểu thức điều kiện > ? <Biểu thức thứ 1> : <Biểu thức thứ 2>

Toán tử này sẽ xác định giá trị của một biểu thức điều kiện, và biểu thức điều kiện này phải trả về một giá trị kiểu bool. Khi điều kiện đúng thì <biểu thức thứ 1> sẽ được thực hiện, còn ngược lại điều kiện sai thì <biểu thức thứ 2> sẽ được thực hiện. Có thể diễn giải theo ngôn ngữ tự nhiên thì toán tử này có ý nghĩa : “*Nếu điều kiện đúng thì làm công việc thứ nhất, còn ngược lại điều kiện sai thì làm công việc thứ hai*”.

## 1.7 Lớp – Đối tượng

### 1.7.1 Định nghĩa lớp

Để định nghĩa một kiểu dữ liệu mới hay một lớp đầu tiên phải khai báo rồi sau đó mới định nghĩa các thuộc tính và phương thức của kiểu dữ liệu đó. Khai báo một lớp bằng cách sử dụng từ khoá **class**. Cú pháp đầy đủ của khai báo một lớp như sau:

[Thuộc tính] [Bổ sung truy cập] **class** <Định danh lớp> [: Lớp cơ sở]  
{

<Phần thân của lớp>

}

Định danh lớp chính là tên của lớp do người xây dựng chương trình tạo ra. Lớp cơ sở là lớp mà đối tượng sẽ kế thừa để phát triển ta sẽ bàn sau. Tất cả các thành viên của lớp được định nghĩa bên trong thân của lớp, phần thân này sẽ được bao bọc bởi hai dấu ({}).

**Ghi chú:** Trong ngôn ngữ C# phần kết thúc của lớp không có đấu chấm phẩy giống như khai báo lớp trong ngôn ngữ C/C++. Tuy nhiên nếu người lập trình thêm vào thì trình biên dịch C# vẫn chấp nhận mà không đưa ra cảnh báo lỗi.

**Thuộc tính truy cập** : Thuộc tính truy cập quyết định khả năng các phương thức của lớp bao gồm việc các phương thức của lớp khác có thể nhìn thấy và sử dụng các biến thành viên hay những phương thức bên trong lớp.

|  |  |
| --- | --- |
| **Thuộc tính** | **Giới hạn truy cập** |
| public | Không hạn chế. Những thành viên được đánh dấu public có thể được dùng bởi bất kì các phương thức của lớp bao gồm những lớp khác. |
| private | Thành viên trong một lớp A được đánh dấu là **private** thì chỉ được truy cập bởi các phương thức của lớp A. |
| protected | Thành viên trong lớp A được đánh dấu là **protected** thì chỉ được các phương thức bên trong lớp A và những phương thức dẫn xuất từ lớp A truy cập. |
| internal | Thành viên trong lớp A được đánh dấu là **internal** thì được truy cập bởi những phương thức của bất cứ lớp nào trong cùng khối hợp ngữ với A. |
| protected internal | Thành viên trong lớp A được đánh dấu là **protected internal** được truy cập bởi các phương thức của lớp A, các phương thức của lớp dẫn xuất của A, và bất cứ lớp nào trong cùng khối hợp ngữ của A. |

### 1.7.2 Tạo đối tượng

Những kiểu dữ liệu chuẩn của C# như int, char, float,… là những kiểu dữ liệu giá trị, và các biến được tạo ra từ các kiểu dữ liệu này được lưu trên stack. Tuy nhiên, với các đối tượng kiểu dữ liệu tham chiếu thì được tạo ra trên heap, sử dụng từ khóa **new** để tạo một đối tượng:

ThoiGian t = new ThoiGian();

t thật sự không chứa giá trị của đối tượng ThoiGian, nó chỉ chứa địa chỉ của đối tượng được tạo ra trên heap, do vậy t chỉ chứa tham chiếu đến một đối tượng mà thôi.

### 1.7.3 Hàm khởi tạo (Constructor):

Khi muốn định nghĩa một phương thức khởi dựng riêng ta phải cung cấp các tham số để hàm khởi dựng có thể khởi tạo các giá trị khác ngoài giá trị mặc định cho các đối tượng. Để định nghĩa một bộ khởi dựng riêng ta phải khai báo một phương thức có tên giống như tên lớp đã khai báo. Phương thức khởi dựng không có giá trị trả về và được khai báo là public. Nếu phương thức khởi dựng này được truyền tham số thì phải khai báo danh sách tham số giống như khai báo với bất kỳ phương thức nào trong một lớp.

### 1.7.4 Hàm Huỷ (Destructor)

Cú pháp phương thức hủy trong ngôn ngữ C# cũng giống như trong ngôn ngữ C++. Nhưng về hành động cụ thể chúng có nhiều điểm khác nhau. Ta khao báo một phương thức hủy trong C# như sau:

~Class1() {}

Tuy nhiên, trong ngôn ngữ C# thì cú pháp khai báo trên là một shortcut liên kết đến một phương thức kết thúc Finalize được kết với lớp cơ sở, do vậy khi viết

~Class1()

{

// Thực hiện một số công việc

}

Cũng tương tự như viết :

Class1.Finalize()  
{

// Thực hiện một số công việc

base.Finalize();  
}

Do sự tương tự như trên nên khả năng dẫn đến sự lộn xộn nhầm lẫn là không tránh khỏi, nên chúng ta phải tránh viết các phương thức hủy và viết các phương thức Finalize tường minh nếu có thể được.

### 1.7.5 Khởi tạo biến thành viên

Các biến thành viên có thể được khởi tạo trực tiếp khi khai báo trong quá trình khởi tạo, thay vì phải thực hiện việc khởi tạo các biến trong bộ khởi dựng. Để thực hiện việc khởi tạo nàyrất đơn giản là việc sử dụng phép gán giá trị cho một biến:  
 private int Giay = 30; // Khởi tạo

Việc khởi tạo biến thành viên sẽ rất có ý nghĩa, vì khi xác định giá trị khởi tạo như vậy thì biến sẽ không nhận giá trị mặc định mà trình biên dịch cung cấp. Khi đó nếu các biến này không được gán lại trong các phương thức khởi dựng thì nó sẽ có giá trị mà ta đã khởi tạo.

### 1.7.6 Bộ khởi tạo sao chép

Bộ khởi dựng sao chép thực hiện việc tạo một đối tượng mới bằng cách sao chép tất cả các biến từ một đối tượng đã có và cùng một kiểu dữ liệu. Ví dụ chúng ta muốn đưa một đối tượng ThoiGian vào bộ khởi dựng lớp ThoiGian để tạo một đối tượng ThoiGian mới có cùng giá trị với đối tượng ThoiGian cũ. Hai đối tượng này hoàn toàn khác nhau và chỉ giống nhau ở giá trị biến thành viên sao khi khởi dựng. Ngôn ngữ C# không cung cấp bộ khởi dựng sao chép, do đó chúng ta phải tự tạo ra.

### 1.7.7 Từ khóa this

Từ khóa **this** được dùng để tham chiếu đến thể hiện hiện hành của một đối tượng. Tham chiếu **this** này được xem là con trỏ ẩn đến tất các phương thức không có thuộc tính tĩnh trong một lớp. Mỗi phương thức có thể tham chiếu đến những phương thức khác và các biến thành viên thông qua tham chiếu **this** này.

* Sử dụng khi các biến thành viên bị che lấp bởi tham số đưa vào.
* Sử dụng tham chiếu **this** để truyền đối tượng hiện hành vào một tham số của một phương thức của đối tượng khác.
* Các thứ ba sử dụng tham chiếu **this** là mảng chỉ mục (indexer).

### 1.7.8 Thành viên static

Những thuộc tính và phương thức trong một lớp có thể là những thành viên thể hiện (Instance members) hay những thành viên tĩnh (Static members). Những thành viên thể hiện hay thành viên của đối tượng liên quan đến thể hiện của một kiểu dữ liệu. Trong khi thành viên tĩnh được xem như một phần của lớp. Chúng ta có thể truy cập đến thành viên tĩnh của một lớp thông qua tên lớp đã được khai báo.

**Ghi chú:** Trong ngôn ngữ C# không cho phép truy cập đến các phương thức tĩnh và các biến thành viên tĩnh thông qua một thể hiện, nếu chúng ta cố làm điều đó thì trình biên dịch C# sẽ báo lỗi, điều này khác với ngôn ngữ C++.Trong một số ngôn ngữ thì có sự phân chia giữa phương thức của lớp và các phương thức khác (toàn cục) tồn tại bên ngoài không phụ thuộc bất cứ một lớp nào. Tuy nhiên, điều này không cho phép trong C#, ngôn ngữ C# không cho phép tạo các phương thức bên ngoài của lớp, nhưng ta có thể tạo được các phương thức giống như vậy bằng cách tạo các phương thức tĩnh bên trong một lớp. Phương thức tĩnh hoạt động ít nhiều giống như phương thức toàn cục, ta truy cập phương thức này mà không cần phải tạo bất cứ thể hiện hay đối tượng của lớp chứa phương thức toàn cục. Tuy nhiên, lợi ích của phương thức tĩnh vượt xa phương thức toàn cục vì phương thức tĩnh được bao bọc trong phạm vi của một lớp nơi nó được định nghĩa, do vậy ta sẽ không gặp tình trạng lộn xộn giữa các phương thức trùng tên do chúng được đặt trong namespace.

## 1.8 Hủy đối tượng (Garbage Collection)

Ngôn ngữ C# cung cấp cơ chế thu dọn (Garbage Collection) và do vậy không cần phải khai báo tường minh các phương thức hủy. Tuy nhiên, khi làm việc với các đoạn mã không được quản lý thì cần phải khai báo tường minh các phương thức hủy để giải phóng các tài nguyên. C# cung cấp ngần định một phương thức để thực hiện điều khiển công việc này, phương thức đó là Finalize() hay còn gọi là bộ kết thúc. Phương thức Finalize này sẽ được gọi bởi cơ chế thu dọn khi đối tượng bị hủy.

Phương thức kết thúc chỉ giải phóng các tài nguyên mà đối tượng nắm giữ, và không tham chiếu đến các đối tượng khác. Nếu với những đoạn mã bình thường tức là chứa các tham chiếu kiểm soát được thì không cần thiết phải tạo và thực thi phương thức Finalize(). Chúngta chỉ làm điều này khi xử lý các tài nguyên không kiểm soát được.

Chúng ta không bao giờ gọi một phương thức Finalize() của một đối tượng một cách trực tiếp, ngoại trừ gọi phương thức này của lớp cơ sở khi ở bên trong phương thức Finalize() của chúng ta. Trình thu dọn sẽ thực hiện việc gọi Finalize() cho chúng ta.

|  |
| --- |
| **Cách Finalize thực hiện** |
| Bộ thu dọn duy trì một danh sách những đối tượng có phương thức Finalize. Danh sách này được cập nhật mỗi lần khi đối tượng cuối cùng được tạo ra hay bị hủy. Khi một đối tượng trong danh sách kết thúc của bộ thu dọn được chọn đầu tiên. Nó sẽ được đặt vào hàng đợi (queue) cùng với những đối tượng khác đang chờ kết thúc. Sau khi phương thức Finalize của đối tượng thực thi bộ thu dọn sẽ gom lại đối tượng và cập nhật lại danh sách hàng đợi, cũng như là danh sách kết thúc đối tượng. |

## 1.9 Kế thừa

Trong ngôn ngữ C#, quan hệ đặc biệt hóa được thực thi bằng cách sử dụng sự kế thừa. Đây không phải là cách duy nhất để thực thi đặc biệt hóa, nhưng nó là cách chung nhất và tự nhiên nhất để thực thi quan hệ này.

### 1.9.1 Thực thi kế thừa

Trong ngôn ngữ C# để tạo một lớp dẫn xuất từ một lớp ta thêm dấu hai chấm vào sau tên lớp dẫn xuất và trước tên lớp cơ sở: public class ListBox : Window Đoạn lệnh trên khai báo một lớp mới tên là ListBox, lớp này được dẫn xuất từ Window. Chúng ta có thể đọc dấu hai chấm có thể được đọc như là “dẫn xuất từ”. Lớp dẫn xuất sẽ kế thừa tất cả các thành viên của lớp cơ sở, bao gồm tất cả các phương thức và biến thành viên của lớp cơ sở. Lớp dẫn xuất được tự do thực thi các phiên bản của một phương thức của lớp cơ sở. Lớp dẫn xuất cũng có thể tạo một phương thức mới bằng việc đánh dấu với từ khóa new.

### 1.9.2 Gọi phương thức khởi dựng của lớp cơ sở

Từ khóa new được sử dụng ở đây để chỉ ra rằng người lập trình đang tạo ra một phiên bản mới cho phương thức này bên trong lớp dẫn xuất. Nếu lớp cơ sở có phương thức khởi dựng mặc định, thì lớp dẫn xuất không cần bắt buộc phải gọi phương thức khởi dựng của lớp cơ sở một cách tường minh. Thay vào đó phương thức khởi dựng mặc định của lớp cơ sở sẽ được gọi một cách ngầm định. Tuy nhiên, nếu lớp cơ sở không có phương thức khởi dựng mặc định, thì tất cả các lớp dẫn xuất của nó phải gọi phương thức khởi dựng của lớp cơ sở một cách tường minh thông qua việc sử dụng từ khóa base.

Khi chúng ta gọi phương thức DrawWindow của một đối tượng của lớp ListBox thì phương thức ListBox.DrawWindow() sẽ được thực hiện, không phải phương thức Window.DrawWindow() của lớp cơ sở Window. Tuy nhiên, ta có thể gọi phương thức DrawWindow() của lớp cơ sở thông qua từ khóa base:

base.DrawWindow(); // gọi phương thức cơ sở Từ khóa base chỉ đến lớp cơ sở cho đối tượng hiện hành.

**Ghi chú*:*** Nếu chúng ta không khai báo bất cứ phương thức khởi dựng nào, thì trình biên dịch sẽ tạo riêng một phương thức khởi dựng cho chúng ta. Khi mà chúng ta viết riêng các phương thức khởi dựng hay là sử dụng phương thức khởi dựng mặc định do trình biên dịch cung cấp hay không thì phương thức khởi dựng mặc định không lấy một tham số nào hết. Tuy nhiên, lưu ý rằng khi ta tạo bất cứ phương thức khởi dựng nào thì trình biên dịch sẽ không cung cấp phương thức khởi dựng cho chúng ta.

### 1.9.3 Điều khiển truy xuất

Khả năng hiện hữu của một lớp và các thành viên của nó có thể được hạn chế thông qua việc sử dụng các bổ sung truy cập: public, private, protected, internal, và protected internal. Như chúng ta đã thấy, public cho phép một thành viên có thể được truy cập bởi một phương thức thành viên của những lớp khác. Trong khi đó private chỉ cho phép các phương thức thành viên trong lớp đó truy xuất. Từ khóa protected thì mở rộng thêm khả năng của private cho phép truy xuất từ các lớp dẫn xuất của lớp đó.

Internal mở rộng khả năng cho phép bất cứ phương thức của lớp nào trong cùng một khối kết hợp (assembly) có thể truy xuất được. Một khối kết hợp được hiểu như là một khối chia xẻ và dùng lại trong CLR. Thông thường, khối này là tập hợp các tập tin vật lý được lưu trữ trong một thư mục bao gồm các tập tin tài nguyên, chương trình thực thi theo ngôn ngữ IL,... Từ khóa internal protected đi cùng với nhau cho phép các thành viên của cùng một khối assembly hoặc các lớp dẫn xuất của nó có thể truy cập. Chúng ta có thể xem sự thiết kế này giống như là internal hay protected.

## 1.10 Đa hình

Có hai cách thức khá mạnh để thực hiện việc kế thừa. Một là sử dụng lại mã nguồn, khi chúng ta tạo ra lớp ListBox, chúng ta có thể sử dụng lại một vài các thành phần trong lớp cơ sở như Window. Tuy nhiên, cách sử dụng thứ hai chứng tỏ được sức mạnh to lớn của việc kế thừa đó là tính đa hình (Polymorphism). Theo tiếng Anh từ này được kết hợp từ poly là nhiều và morph có nghĩa là form (hình thức). Do vậy, đa hình được hiểu như là khả năng sử dụng nhiều hình thức của một kiểu mà không cần phải quan tâm đến từng chi tiết. Khi một tổng đài điện thoại gởi cho máy điện thoại của chúng ta một tín hiệu có cuộc gọi. Tổng đài không quan tâm đến điện thoại của ta là loại nào. Có thể ta đang dùng một điện thoại cũ dùng motor để rung chuông, hay là một điện thoại điện tử phát ra tiếng nhạc số. Hoàn toàn các thông tin về điện thoại của ta không có ý nghĩa gì với tổng đài, tổng đài chỉ biết một kiểu cơ bản là điện thoại mà thôi và diện thoại này sẽ biết cách báo chuông. Còn việc báo chuông như thế nào thì tổng đài không quan tâm. Tóm lại, tổng đài chỉ cần bảo điện thoại hãy làm điều gì đó để reng. Còn phần còn lại tức là cách thức reng là tùy thuộc vào từng loại điện thoại. Đây chính là tính đa hình.

#### 1.10.1 Kiểu đa hình

Do một ListBox là một Window và một Button cũng là một Window, chúng ta mong muốn sử dụng cả hai kiểu dữ liệu này trong tình huống cả hai được gọi là Window. Ví dụ như trong một form giao diện trên MS Windows, form này chứa một tập các thể hiện của Window. Khi form được hiển thị, nó yêu cầu tất cả các thể hiện của Window tự thực hiện việc tô vẽ. Trong trường hợp này, form không muốn biết thành phần thể hiện là loại nào như Button, CheckBox,...,. Điều quan trọng là form kích hoạt toàn bộ tập hợp này tự thực hiện việc vẽ. Hay nói ngắn gọn là form muốn đối xử với những đối tượng Window này một cách đa hình.

#### 1.10.2 Phương thức đa hình

Để tạo một phương thức hỗ tính đa hình, chúng ta cần phải khai báo khóa virtual trong phương thức của lớp cơ sở. Ví dụ, để chỉ định rằng phương thức DrawWindow() của lớp Window trong ví dụ 5.1 là đa hình, đơn giản là ta thêm từ khóa virtual vào khai báo như sau: public virtual void DrawWindow() Lúc này thì các lớp dẫn xuất được tự do thực thi các cách xử riêng của mình trong phiên bản mới của phương thức DrawWindow(). Để làm được điều này chỉ cần thêm từ khóa **override** để chồng lên phương thức ảo DrawWindow() của lớp cơ sở. Sau đó thêm các đoạn mã nguồn mới vào phương thức viết chồng này.

#### 1.10.3 Từ khóa override

Trong ngôn ngữ C#, người lập trình có thể quyết định phủ quyết một phương thức ảo bằng cách khai báo tường minh từ khóa override. Điều này giúp cho ta đưa ra một phiên bản mới của chương trình và sự thay đổi của lớp cơ sở sẽ không làm ảnh hưởng đến chương trình viết trong các lớp dẫn xuất. Việc yêu cầu sử dụng từ khóa override sẽ giúp ta ngăn ngừa vấn đề này. Bây giờ ta thử bàn về vấn đề này, giả sử lớp cơ sở Window của ví dụ trước được viết bởi một công ty A. Cũng giả sử rằng lớp ListBox và RadioButton đươc viết từ những người lập trình của công ty B và họ dùng lớp cơ sở Window mua được của công ty A làm lớp cơ sở cho hai lớp trên. Người lập trình trong công ty B không có hoặc có rất ít sự kiểm soát về những thay đổi trong tương lai với lớp Window do công ty A phát triển.

## 1.11 Lớp trừu tượng

Mỗi lớp con của lớp Window nên thực thi một phương thức DrawWindow() cho riêng mình. Tuy nhiên điều này không thực sự đòi hỏi phải thực hiện một cách bắt buộc. Để yêu cầu các lớp con (lớp dẫn xuất) phải thực thi một phương thức của lớp cơ sở, chúng ta phải thiết kế một phương thức một cách trừu tượng.

Một phương thức trừu tượng không có sự thực thi. Phương thức này chỉ đơn giản tạo ra một tên phương thức và ký hiệu của phương thức, phương thức này sẽ được thực thi ở các lớp dẫn xuất.

Những lớp trừu tượng được thiết lập như là cơ sở cho những lớp dẫn xuất, nhưng việc tạo các thể hiện hay các đối tượng cho các lớp trừu tượng được xem là không hợp lệ. Một khi chúng ta khai báo một phương thức là trừu tượng, thì chúng ta phải ngăn cấm bất cứ việc tạo thể hiện cho lớp này.

Do vậy, nếu chúng ta thiết kế phương thức DrawWindow() như là trừu tượng trong lớp Window, chúng ta có thể dẫn xuất từ lớp này, nhưng ta không thể tạo bất cứ đối tượng cho lớp này. Khi đó mỗi lớp dẫn xuất phải thực thi phương thức DrawWindow(). Nếu lớp dẫn xuất không thực thi phương thức trừu tượng của lớp cơ sở thì lớp dẫn xuất đó cũng là lớp trừu tượng, và ta cũng không thể tạo các thể hiện của lớp này được.

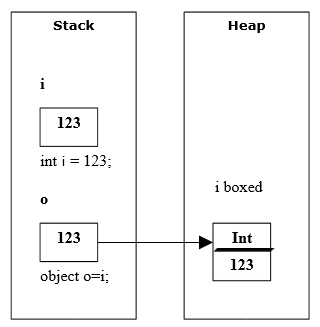
Phương thức trừu tượng được thiết lập bằng cách thêm từ khóa abstract vào đầu của phần định nghĩa phương thức, cú pháp thực hiện như sau: abstract public void DrawWindow( );

Do phương thức không cần phần thực thi, nên không có dấu ({}) mà chỉ có dấu chấm phẩy (;) sau phương thức. Như thế với phương thức DrawWindow() được thiết kế là trừu tượng thì chỉ cần câu lệnh trên là đủ.

## 1.12 Boxing và Unboxing dữ liệu

Boxing và unboxing là những xử lý cho phép kiểu dữ liệu giá trị (như int, long,...) được đối xử như kiểu dữ liệu tham chiếu (các đối tượng). Một giá trị được đưa vào bên trong của đối tượng, được gọi là Boxing. Trường hợp ngược lại, Unboxing sẽ chuyển từ đối tượng ra một giá trị.

**Boxing được thực hiện ngầm định:** Boxing là một sự chuyển đổi ngầm định của một kiểu dữ liệu giá trị sang kiểu dữ liệu tham chiếu là đối tượng. Boxing một giá trị bằng cách tạo ra một thể hiển của đối tượng cần dùng và sao chép giá trị trên vào đối tượng mới tạo. Ta có hình vẽ sau minh họa quá trình Boxing một số nguyên.

****

Boxing được thực hiện ngầm định khi chúng ta đặt một kiểu giá trị vào một tham chiếu đang chờ đợi và giá trị sẽ được đưa vào đối tượng một cách tự động ngầm định. Ví dụ, nếu chúng ta gán một kiểu dư liệu cơ bản như kiểu nguyên int vào một biến kiểu Object (điều này hoàn toàn hợp lệ vì kiểu int được dẫn xuất từ lớp Object) thì giá trị này sẽ được đưa vào biến Object.

Unboxing phải được thực hiện tường minh : Việc đưa một giá trị vào một đối tượng được thực hiện một cách ngầm định. Và sự thực hiện ngược lại, unboxing, tức là đưa từ một đối tượng ra một giá trị phải được thực hiện một cách tường minh. Chúng ta phải thiết lập theo hai bước sau:

* Phải chắc chắn rằng đối tượng đã boxing đúng kiểu giá trị đưa ra.
* Sao chép giá trị từ thể hiện hay đối tượng vào biến kịểu giá trị.

Để thực hiện unboxing thành công, thì đối tượng được unboxing phải được tham chiếu đến một đối tượng, và đối tượng này đã được tạo ra bằng việc boxing một giá trị cùng với kiểu của giá trị được đưa ra.

## 1.13 Nạp chồng toán tử

#### 1.13.1 Sử dụng từ khóa operator

Trong ngôn ngữ C#, các toán tử là các phương thức tĩnh, giá trị trả về của nó thể hiện kết quả của một toán tử và những tham số là các toán hạng. Khi chúng ta tạo một toán tử cho mộtlớp là chúng ta đã thực việc nạp chồng (overloaded) những toán tử đó, cũng giống như là chúng ta có thể nạp chồng bất cứ phương thức thành viên nào. Do đó, để nạp chồng toán tử cộng (+) chúng ta có thể viết như sau:

public static Fraction operator + ( Fraction lhs, Fraction rhs)

Trong toán tử trên ta có sự qui ước đặt tên của tham số là lhs và rhs. Tham số tên lhs thay thế cho “left hand side” tức là toán hạng bên trái, tương tự tham số tên rhs thay thế cho “right hand side” tức là toán hạng bên phải.

Cú pháp ngôn ngữ C# cho phép nạp chồng một toán tử bằng cách viết từ khóa operator và theo sau là toán tử được nạp chồng. Từ khóa operator là một bổ sung phương thức (method operator). Như vậy, để nạp chồng toán tử cộng (+) chúng ta có thể viết operator +. Khi chúng ta viết: Fraction theSum = firstFraction + secondFraction;

Thì toán tử nạp chồng + được thực hiện, với firstFraction được truyền vào như là tham số đầu tiên, và secondFraction được truyền vào như là tham số thứ hai. Khi trình biên dịch gặp biểu thức: firstFraction + secondFraction thì trình biên dịch sẽ chuyển biểu thức vào: Fraction.operator+(firstFraction, secondFraction).

Kết quả sau khi thực hiện là một đối tượng Fraction mới được trả về, trong trường hợp này phép gán sẽ được thực hiện để gán một đối tượng Fraction cho theSum.

### 1.13.2 Sử dụng toán tử

Nạp chồng toán tử có thể làm cho mã nguồn của chúng ta trực quan và những hành động của lớp mà chúng ta xây dựng giống như các lớp được xây dựng sẵn. Tuy nhiên, việc nạp chồng toán tử cũng có thể làm cho mã nguồn phức tạp một cách khó quản lý nếu chúng ta phá vỡ cách thể hiện thông thường để sử dụng những toán tử. Hạn chế việc sử dụng tùy tiện các nạp chồng toán tử bằng những cách sử dụng mới và những cách đặc trưng.

Ví dụ, mặc dù chúng ta có thể hấp dẫn bởi việc sử dụng nạp chồng toán tử gia tăng (++) trong lớp Employee để gọi một phương thức gia tăng mức lương của nhân viên, điều này có thể đem lại rất nhiều nhầm lẫn cho các lớp client truy cập lớp Employee. Vì bên trong của lớp còn có thể có nhiều trường thuộc tính số khác, như số tuổi, năm làm việc,...ta không thể dành toán tử gia tăng duy nhất cho thụôc tính lương được. Cách tốt nhất là sử dụng nạp chồng toán tử một cách hạn chế, và chỉ sử dụng khi nào nghĩa nó rõ ràng và phù hợp với các toán tử của các lớp được xây dựng sẵn.

Khi thường thực hiện việc nạp chồng toán tử so sánh bằng (==) để kiểm tra hai đối tượng xem có bằng nhau hay không. Ngôn ngữ C# nhấn mạnh rằng nếu chúng ta thực hiện nạp chồng toán tử bằng, thì chúng ta phải nạp chồng toán tử nghịch với toán tử bằng là toán tử không bằng (!=). Tương tự, khi nạp chồng toán tử nhỏ hơn (<) thì cũng phải tạo toán tử (>) theo từng cặp. Cũng như toán tử (>=) đi tương ứng với toán tử (<=).

Theo sau là một số luật được áp dụng để thực hiện nạp chồng toán tử:

* Định nghĩa những toán tử trong kiểu dữ liệu giá trị, kiểu do ngôn ngữ xây dựng sẵn.
* Cung cấp những phương thức nạp chồng toán tử chỉ bên trong của lớp nơi mà những phương thức được định nghĩa.
* Sử dụng tên và những kí hịêu qui ước được mô tả trong Common Language Specification (CLS).

Sử dụng nạp chồng toán tử trong trường hợp kết quả trả về của toán tử là thật sự rõ ràng. Ví dụ, như thực hiện toán tử trừ (-) giữa một giá trị Time với một giá trị Time khác là một toán tử có ý nghĩa. Tuy nhiên, nếu chúng ta thực hiện toán tử or hay toán tử and giữa hai đối tượng Time thì kết quả hoàn toàn không có nghĩa gì hết.

Nạp chồng toán tử có tính chất đối xứng. Ví dụ, nếu chúng ta nạp chồng toán tử bằng (==) thì cũng phải nạp chồng toán tử không bằng (!=). Do đó khi thực hiện toán tử có tính chất đối xứng thì phải thực hiện toán tử đối xứng lại như: < với >, <= với >=.

Phải cung cấp các phương thức thay thế cho toán tử được nạp chồng. Đa số các ngôn ngữ điều không hỗ trợ nạp chồng toán tử. Vì nguyên do này nên chúng ta phải thực thi các phương thức thứ hai có cùng chức năng với các toán tử. Common Language Specification (CLS) đòi hỏi phải thực hiện phương thức thứ hai tương ứng.

Bảng dưới trình bày một số toán tử cùng với biểu tượng của toán tử và các tên của phương thức thay thế các toán tử.

|  |  |  |
| --- | --- | --- |
| Biểu tượng | Tên phương thức thay thế | Tên toán tử |
| + | Add | Toán tử cộng. |
| - | Subtract | Toán tử trừ. |
| \* | Multiply | Toán tử nhân. |
| / | Divide | Toán tử chia. |
| % | Mod | Toán tử lấy dư. |
| ^ | Xor | Toán tử or loại trừ. |
| & | BitwiseAnd | Toán tử and nhị phân. |
| | | BitwiseOr | Toán tử or nhị phân |
| && | And | Toán tử and logic. |
| || | Or | Toán tử or logic |
| = | Assign | Toán tử gán |
| << | LeftShift | Toán tử dịch trái |
| >> | RightShift | Toán tử dịch phải |
| == | Equals | Toán tử so sánh bằng. |
| > | Compare | Toán tử lớn hơn. |
| < | Compare | Toán tử so sánh nhỏ hơn. |
| != | Compare | Toán tử so sánh không bằng. |
| >= | Compare | Toán tử so sánh lớn hơn hoặc bằng. |
| <= | Compare | Toán tử so sánh nhỏ hơn hoặc bằng. |
| \*= | Multiply | Toán tử nhân rồi gán trở lại |
| -= | Subtract | Toán tử trừ rồi gán trở lại. |

### 1.13.3 Toán tử so sánh bằng

Nếu chúng ta nạp chồng toán tử bằng (==), thì chúng ta cũng nên phủ quyết phương thức ảo Equals() được cung cấp bởi lớp object và chuyển lại cho toán tử bằng thực hiện. Điều này cho phép lớp của chúng ta thể tương thích với các ngôn ngữ .NET khác không hỗ trợ tính nạp chồng toán tử nhưng hỗ trợ nạp chồng phương thức. Những lớp FCL không sử dụng nạp chồng toán tử, nhưng vẫn mong đợi lớp của chúng ta thực hiện những phương thức cơ bản này. Do đó ví dụ lớp ArrayList mong muốn chúng ta thực thi phương thức Equals(). Lớp object thực thi phương thức Equals() với khai báo sau: public override bool Equals( object 0 )

Bằng cách phủ quyết phương thức này, chúng ta cho phép lớp Fraction hành động một cách đa hình với tất cả những lớp khác. Bên trong thân của phương thức Equals() chúng ta cần phải đảm bảo rằng chúng ta đang so sánh với một Fraction khác, và nếu như chúng ta đã thực thi một toán tử so sánh bằng thì có thể định nghĩa phương thức Equals() như sau:

pubic override bool Equals( object o) {

if ( !(o is Fraction) )

{

return false;

}

return this == (Fraction) o;

}

Toán tử is được sử dụng để kiểm tra kiểu của đối tượng lúc chạy chương trình có tương thích với toán hạng trong trường hợp này là Fraction. Do o là Fraction nên toán tử is sẽ trả về true.

## 1.14 Cấu trúc

### 1.14.1 Định nghĩa cấu trúc

Cú pháp để khai báo một cấu trúc cũng tương tự như cách khai báo một lớp:

[thuộc tính] [bổ sung truy cập] struct <tên cấu trúc> [: danh sách giao diện]

{

[thành viên của cấu trúc]

}

Không giống như những lớp, cấu trúc không hỗ trợ việc thừa kế. Chúng được thừa kế ngầm định từ lớp object (tương tự như tất cả các kiểu dữ liệu trong C#, bao gồm các kiểu dữ liệu xây dựng sẵn) nhưng không thể kế thừa từ các lớp khác hay cấu trúc khác. Cấu trúc cũng được ngầm định là sealed, điều này có ý nghĩa là không có lớp nào hay bất cứ cấu trúc nào có thể dẫn xuất từ nó. Tuy nhiên, cũng giống như các lớp, cấu trúc có thể thực thi nhiều giao diện. Sau đây là một số sự khác nhau nữa là:

* Không có bộ hủy và bộ khởi tạo mặc định tùy chọn: Những cấu trúc không có bộ hủy và cũng không có bộ khởi tạo mặc định không tham số tùy chọn. Nếu chúng ta không cung cấp bất cứ bộ khởi tạo nào thì cấu trúc sẽ được cung cấp một bộ khởi tạo mặc định, khi đó giá trị 0 sẽ được thiết lập cho tất cả các dữ liệu thành viên hay những giá trị mặc định tương ứng cho từng kiểu dữ liệu (bảng 4.2). Nếu chúng ta cung cấp bất cứ bộ khởi dựng nào thì chúng ta phải khởi tạo tất cả các trường trong cấu trúc.
* Không cho phép khởi tạo: chúng ta không thể khởi tạo các trường thể hiện (instance fields) trong cấu trúc, do đó đoạn mã nguồn sau sẽ không hợp lệ:

private int xVal = 20;

private int yVal = 50;

mặc dù điều này thực hiện tốt đối với lớp.

Cấu trúc được thiết kế hướng tới đơn giản và gọn nhẹ. Trong khi các dữ liệu thành viên private hỗ trợ việc che dấu dữ liệu và sự đóng gói. Một vài người lập trình có cảm giác rằng điều này phá hỏng cấu trúc. Họ tạo một dữ liệu thành viên public, do vậy đơn giản thực thi một cấu trúc. Những người lập trình khác có cảm giác rằng những thuộc tính cung cấp một giao diện rõ ràng, đơn giản và việc thực hiện lập trình tốt đòi hỏi phải che dấu dữ liệu thậm chí với dữ liệu rất đơn giản. Chúng ta sẽ chọn cách nào, nói chung là phụ thuộc vào quan nệm thiết kế của từng người lập trình. Dù chọn cách nào thì ngôn ngữ C# cũng hỗ trợ cả hai cách tiếp cận.

### 1.14.2 Tạo cấu trúc

Chúng ta tạo một thể hiện của cấu trúc bằng cách sử dụng từ khóa new trong câu lệnh gán, như khi chúng ta tạo một đối tượng của lớp.

* Cấu trúc là một kiểu giá trị
* Gọi bộ khởi dựng mặc định
* Tạo cấu trúc không gọi new: Bởi vì Location là một cấu trúc không phải là lớp, do đó các thể hiện của nó sẽ được tạo trong stack. Tuy nhiên, toán tử new gọi bộ khởi dựng của lớp Location, không giống như với một lớp, cấu trúc có thể được tạo ra mà không cần phải gọi toán tử new.

# Chương 2 : Nền tảng .NET

## 2.1 Tổng quan .Net framework

**.NET Framework** là một nền tảng lập trình và cũng là một nền tảng thực thi ứng dụng chủ yếu trên hệ điều hành Microsoft Windows được phát triển bởi Microsoft. Các chương trình được viết trên nền .NET Framework sẽ được triển khai trong môi trường phần mềm (ngược lại với môi trường phần cứng) được biết đến với tên Common Language Runtime (CLR). Môi trường phần mềm này là một máy ảo trong đó cung cấp các dịch vụ như an ninh phần mềm (Security), quản lý bộ nhớ (Memory Management), và các xử lý lỗi ngoại lệ (Exception Handling).

.NET framework bao gồm tập các thư viện lập trình lớn, và những thư viện này hỗ trợ việc xây dựng các chương trình phần mềm như lập trình giao diện; truy cập, kết nối cơ sở dữ liệu; ứng dụng web; các giải thuật, cấu trúc dữ liệu; giao tiếp mạng... CLR cùng với bộ thư viện này là 2 thành phần chính của.NET framework.

.NET framework đơn giản hóa việc viết ứng dụng bằng cách cung cấp nhiều thành phần được thiết kế sẵn, người lập trình chỉ cần học cách sử dụng và tùy theo sự sáng tạo mà gắn kết các thành phần đó lại với nhau. Nhiều công cụ được tạo ra để hỗ trợ xây dựng ứng dụng.NET, và IDE (*Integrated Developement Environment*) được phát triển và hỗ trợ bởi chính Microsoft là Visual Studio.

## 2.2 Lịch sử phát triển

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Phiên bản** | **Số hiệu phiên bản** | **Ngày phát hành** | **Visual Studio** | **Được phát hành kèm theo** |
| 1.0 | 1.0.3705.0 | 13 tháng 2 năm 2002 | Visual Studio.NET | Windows XP Tablet and Media Center Editions |
| 1.1 | 1.1.4322.573 | 24 tháng 4 năm 2003 | Visual Studio.NET 2003 | Windows Server 2003 |
| 2.0 | 2.0.50727.42 | 7 tháng 11 năm 2005 | Visual Studio 2005 | Windows Server 2003 R2 |
| 3.0 | 3.0.4506.30 | 6 tháng 11 năm 2006 |  | Windows Vista, Windows Server 2008 |
| 3.5 | 3.5.21022.8 | 19 tháng 11 năm 2007 | Visual Studio 2008 | Windows 7, Windows Server 2008 R2 |
| 4.0 | 4.0.30319.1 | 12 tháng 4 năm 2010 | Visual Studio 2010 |  |
| 4.5 | 4.5.50709 | 15 tháng 8 năm 2012 | Visual Studio 2012 | Windows 8, Windows Server 2012 |
| 4.6 |  | 20 tháng 7 năm 2015 | Visual Studio 2015 | Windows 10 |

# Chương 3. Entity Framework

## 3.1 Tổng quan về Entity Framework

### 3.1.1 Giới thiệu

Microsoft ADO.NET Entity Framework là một nền tảng sử dụng làm việc với cơ sở dữ liệu quan hệ thông qua cơ chế ánh xạ đối tượng Object / Relational Mapping (ORM).

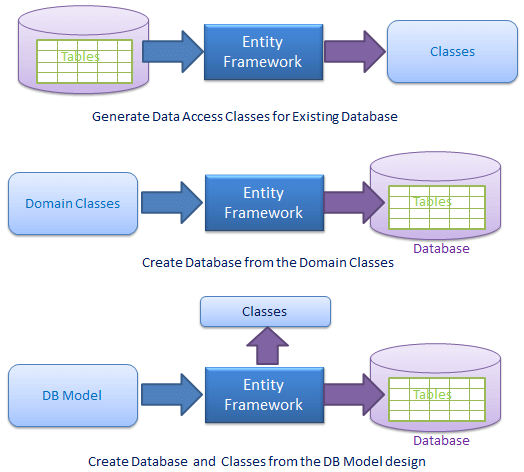
Entity Framework cho phép các nhà phát triển làm việc với cơ sở dữ liệu quan hệ trên mô hình các đối tượng không phải làm việc trực tiếp trên các bảng trong cơ sở dữ liệu, do đó người phát triển không phải viết các đoạn mã truy cập cơ sở dữ liệu thường cần phải viết.

Sử dụng Entity Framework, các nhà phát triển ứng dụng sử dụng các truy vấn LINQ, sau đó lấy ra và thao tác dữ liệu như các đối tượng. Ngoài ra  Entity Framework cung cấp các dịch vụ như theo dõi sự thay đổi, lazy loading, và biên dịch truy vấn để các nhà phát triển có thể tập trung vào xử lý nghiệp vụ ứng dụng, không phải tập trung vào các truy cập dữ liệu cơ bản.

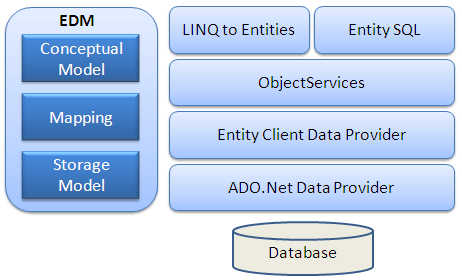
Entity framework là một nền tảng nâng cao của ADO.NET vì vậy Entity framework cung cấp cho các nhà phát triển ứng dụng một cơ chế tự động cho việc truy cập và lưu trữ dữ liệu trong cơ sở dữ liệu.

**Entity Framework sử dụng trong 3 trường hợp:**

* Thứ nhất, nếu đã có cơ sở dữ liệu hoặc muốn thiết kế cơ sở dữ liệu trước các phần khác của ứng dụng.
* Thứ hai, thiết kế các lớp domain trước và sau đó tạo ra cơ sở dữ liệu từ các lớp domain.
* Thứ ba, thiết kế lược đồ dữ liệu trên thiết kế trực quan và sau đó tạo ra các cơ sở dữ liệu và các lớp.



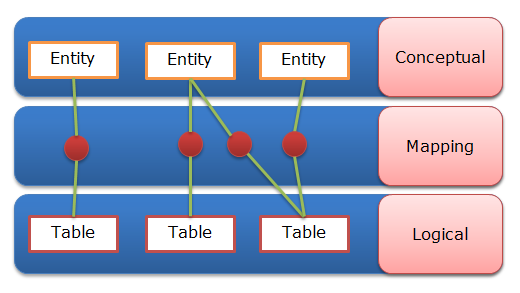
### 3.1.2 Kiến trúc của Entity Framework



Kiến trúc Entity Framework bao gồm các thành phần sau:

* **EDM (Entity Data Model)**: EDM bao gồm ba phần chính – mô hình khái niệm, ánh xạ và mô hình lưu trữ.
* **Mô hình khái niệm (Conceptual Model)**: Mô hình khái niệm chứa các lớp model và các mối quan hệ.
* **Mô hình lưu trữ (Storage Model)**: Mô hình lưu trữ là mô hình thiết kế cơ sở dữ liệu bao gồm bảng, view, thủ tục lưu trữ, và các mối quan hệ và khóa chính.
* **Ánh xạ (Mapping)**: Ánh xạ bao gồm các thông tin về cách thức mô hình khái niệm được ánh xạ tới mô hình lưu trữ.
* **LINQ to Entities:** LINQ to Entities là ngôn ngữ truy vấn được sử dụng để viết các truy vấn trên mô hình đối tượng. Nó trả về các thực thể, được định nghĩa trong mô hình khái niệm. Chúng ta có thể sử dụng các câu lệnh LINQ ở đây.
* **Entity SQL**: Entity SQL là ngôn ngữ truy vấn giống như LINQ to Entities. Tuy nhiên, Entity SQL khó hơn LINQ to Entities và các nhà phát triển sẽ phải tìm hiểu nó một cách riêng biệt.
* **Object Service**: Object Service là một điểm vào chính cho việc truy cập dữ liệu từ cơ sở dữ liệu và trả lại Object Service trở lại. Object Service có nhiệm vụ chuyển đổi dữ liệu thực thể được cung cấp bởi client thành thực thể cấu trúc đối tượng.
* **Entity Client Data Provider**: Nhiệm vụ chính của lớp này là chuyển đổi L2E hoặc Entity SQL thành câu lệnh truy vấn SQL mà cơ sở dữ liệu thực thi được.
* **ADO.Net Data Provider**: Lớp này có nhiệm vụ giao tiếp với cơ sở dữ liệu bằng cách sử dụng ADO.Net chuẩn.

## 3.2 Giới thiệu về Entity Data Model



**Entity Data Model (EDM)** là mô hình dữ liệu được mô tả thông qua các ngôn ngữ theo chuẩn XML. EDM được chia làm 3 lớp là: Conceptual, Mapping và Logical. Mỗi lớp này được định nghĩa bởi ngôn ngữ riêng theo định dạng XML:

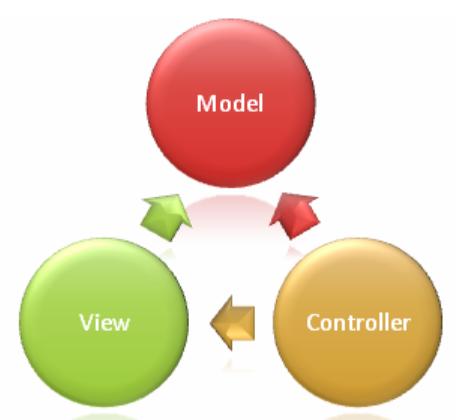
* **Conceptual – Conceptual Schema Definition Language (CSDL):** là ngôn ngữ định nghĩa các entity, relationship, hàm trong tập tin với phần mở rộng .csdl. Có thể tạo được các entity class (object layer).
* **Mapping – Mapping specification language (MSL)**: định nghĩa các ánh xạ giữa lớp conceptual và logical, nội dung này được lưu trong tập tin .msl.
* **Logical – Store Schema Definition Language (SSDL):** định nghĩa mô hình lưu trữ của dữ liệu, lưu trữ trong tập tin .ssdl.

Các nội dung của ba tập tin .csdl, .msl và .ssdl được lưu trữ trong cùng tập tin .**edmx** trong Visual Studio. Tập tin này có thể được tạo ra tự động từ database và công cụ Entity Framework Model Wizard của Visual Studio. Trong quá trình biên dịch, các tập tin .csdl, .msl và .ssdl sẽ được tạo ra dựa vào tập tin .edmx này.

**Note**: Khi mở tập tin .edmx này, VS tự động hiển thị giao diện trực quan của nó bằng công cụ mặc định là ADO.NET Entity Data Model Designer. Để xem nội dung của tập tin này, bạn nhấn chuột phải, nhấn **Open with…**và chọn mục XML Editor hoặc một trình xem dưới dạng văn bản bất kì.

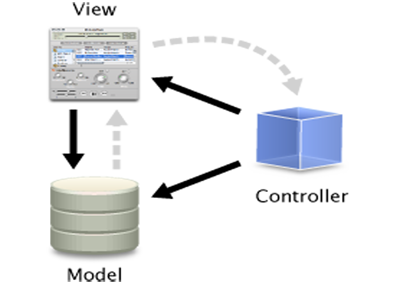
## 3.3 Mô hình MVC

### 3.3.1 Giới thiệu về mô hình MVC



Mô hình MVC (Model - View - Controller) là một kiến trúc phần mềm hay mô hình thiết kế được sử dụng trong kỹ thuật phần mềm. Nó giúp cho các developer tách ứng dụng của họ ra 3 thành phần khác nhau Model, View và Controller. Mỗi thành phần có một nhiệm vụ riêng biệt và độc lập với các thành phần khác.

### 3.3.2 Các thành phần trong MVC



**Model:** Đây là thành phần chứa tất cả các nghiệp vụ logic, phương thức xử lý, truy xuất database, đối tượng mô tả dữ liệu như các Class, hàm xử lý...

**View:** Đảm nhận việc hiển thị thông tin, tương tác với người dùng, nơi chứa tất cả các đối tượng GUI như textbox, images...Hiểu một cách đơn giản, nó là tập hợp các form hoặc các file HTML.

**Controller:** Giữ nhiệm vụ nhận điều hướng các yêu cầu từ người dùng và gọi đúng những phương thức xử lý chúng... Chẳng hạn thành phần này sẽ nhận request từ url và form để thao tác trực tiếp với Model.

### 3.3.3 Sự tương tác giữa các lớp trong mô hình MVC

**Controller – View** sẽ lấy những hình ảnh, nút bấm…hoặc hiển thị dữ liệu được trả ra từ Controller để người dùng có thể quan sát và thao tác. Trong sự tương tác này cũng có thể không có dữ liệu được lấy từ Model và khi đó nó chỉ chịu trách nhiệm hiển thị đơn thuần như hình ảnh, nút bấm…

**Controller – Model** là luồng xử lý khi controller tiếp nhận yêu cầu và các tham số đầu vào từ người dùng, controller sẽ sử dụng các lớp/hàm trong Model cần thiết để lấy ra những dữ liệu chính xác.

**View – Model** có thể tương tác với nhau mà không qua Controller, nó chỉ đảm nhận hiển thị dữ liệu chứ không phải qua bất kỳ xử lý nghiệp vụ logics nào. Nó giống như các vùng dữ liệu hiển thị tĩnh trên các website như block slidebar…

Mô hình MVC có thể áp dụng cho nhiều loại dự án khác nhau : dự án website, dự án ứng dụng – phần mềm,… nhưng trong phạm vi bài viết chúng ta sẽ chỉ đi tìm hiểu mô hình này được áp dụng như nào trong các dự án xây dựng và phát triển website.

### 3.3.4 Ưu điểm

* Các dự án có thể áp dụng ngay mô hình MVC mà không phụ thuộc môi trường, nền tảng xây dựng hay ngôn ngữ lập trình phát triển;
* Quy hoạch các class/ function vào các thành phần riêng biệt Controller – Model – View, khi đó sẽ dễ dàng xây dựng – phát triển – quản lý – vận hành và bảo trì một dự án, tạo sự rõ ràng, trong sáng trong quá trình phát triển dự án, kiểm soát được các luồng xử lý và tạo ra các thành phần xử lý nghiệp vụ chuyên biệt hóa.
* Tạo thành mô hình chuẩn cho nhiều dự án, các chuyên gia sẽ tiếp cận – tìm hiểu những dự án đó một cách nhanh chóng và hiệu quả . Nếu bạn nắm rõ mô hình MVC của một dự án nào đó, thì khi tiếp cận với một dự án khác mà bạn chưa từng biết hoặc tiếp xúc, nhưng nó lại được xây dựng với mô hình MVC thì sẽ không khó khăn gì mà cực kỳ dễ dàng. Học một nhưng có thể hiểu và sử dụng được mười.
* Giúp các chuyên gia lập trình, nhà quản lý, nhà đầu tư, PM… có thể hiểu được dự án hoạt động ra sao hoặc giúp các lập trình viên dễ dàng quản lý – phát triển dự án. Nó không phải ngôn ngữ, nhưng khi họ cùng nhìn vào nó thì sẽ tự hiểu nó là gì, khi đó họ có thể trao đổi các yêu cầu và bàn bạc công việc.
* Đây là một mô hình chuẩn, nó tối ưu nhất hiện nay so với nhiều mô hình khác và được sử dụng trong nhiều dự án và nhiều lĩnh vực, đặc biệt trong công nghệ sản xuất ứng dụng – phần mềm. Các lập trình viên sử dụng mô hình chuẩn MVC để có thể dễ dàng phân phối và chuyển giao công nghệ.
* Đây là mô hình đơn giản, xử lý những nghiệp vụ đơn giản, và dễ dàng triển khai với các dự án nhỏ.

### 3.3.5 Nhược điểm

Yêu cầu về chuyên môn khá cao, có kiến thức vững về các mô hình chuẩn. Khó triển khai với những dự án yêu cầu phức tạp hơn. Hiện nay đang có một khái niệm mô hình mới đó là HMVC đang dần thay thế cho MVC

### 3.3.6 Liên hệ MVC và Entity Framework



Trong hình trên, Entity Framework dường như có vị trí rất tự do, xuất hiện ở cả 3 tầng Data Access Layer, Business Layer, và cả Presentation Layer. Có thể hình dung Entity Framework như là 1 thư viện, hễ tầng nào cần thì chỉ cần gọi đến. Mô hình này xuất phát có thể là sự cẩu thả của lập trình viên khi không thích tuân theo tiêu chuẩn quan hệ giữa các tầng hoặc cách giải thích khác là sự lỏng lẻo giúp cho việc lập trình thuận tiện hơn. Trong vài mô hình người ta gom tầng Data Access Layer và Business Layer chỉ làm tầng Data Model, và vị trí Entity Framework cũng bao hàm tầng Data Model và Presentation Layer như hình.

Có thể hiểu 1 cách chung nhất Entity Framework có vị trí trung gian, đóng vai trò kết nối giữa cơ sở dữ liệu và các thành phần khác của 1 dự án Web khi cần đến. Ngoài ra, có nhiều cách hiểu về vị trí của Entity Framework ở đâu trong mô hình Web, ta sẽ dần khám phá để đưa ra cách hiểu và định nghĩa riêng trong quá trình thiết kế và xây dựng dự án Web ASP.NET.

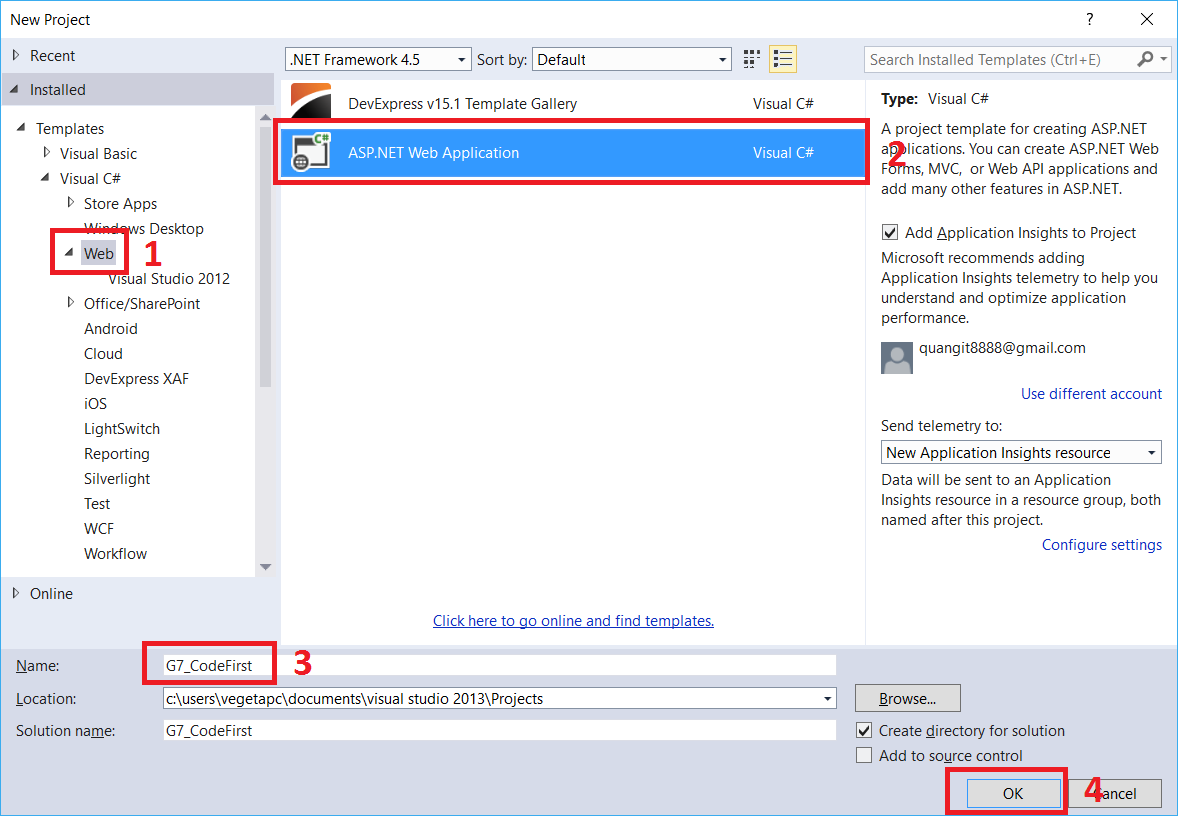
# Chương 4.

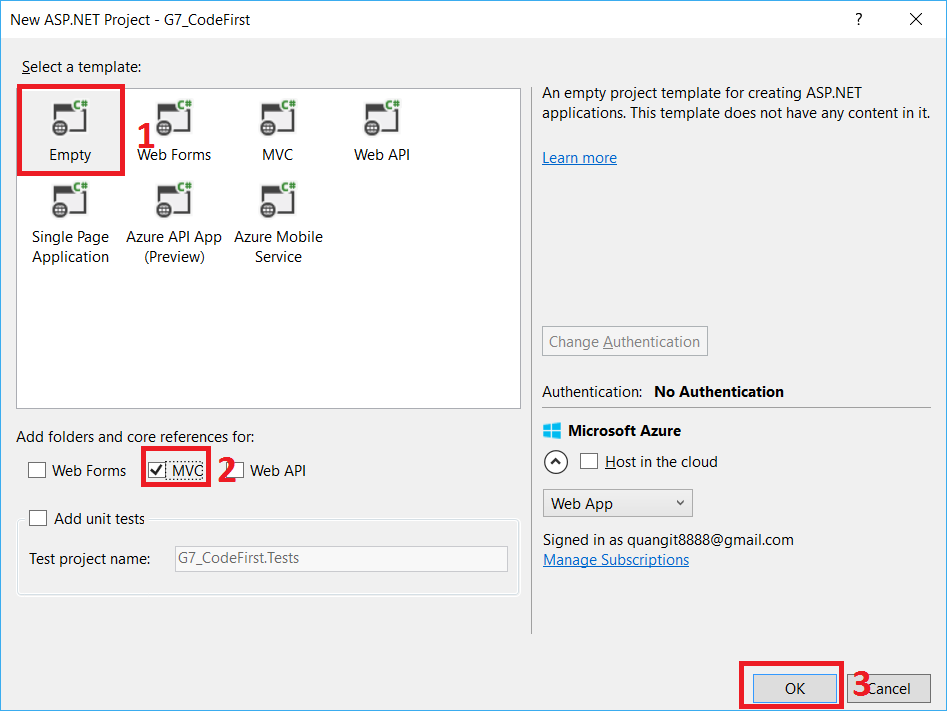
# Chương 5. Phát triển ứng dụng với Entity Framework

Mục tiêu: Tạo website ASP.NET kết hợp MVC để đăng ký môn học cho sinh viên gồm các chức năng CRUID (Create – Read – Update – Insert – Delete).

1. **Code first**

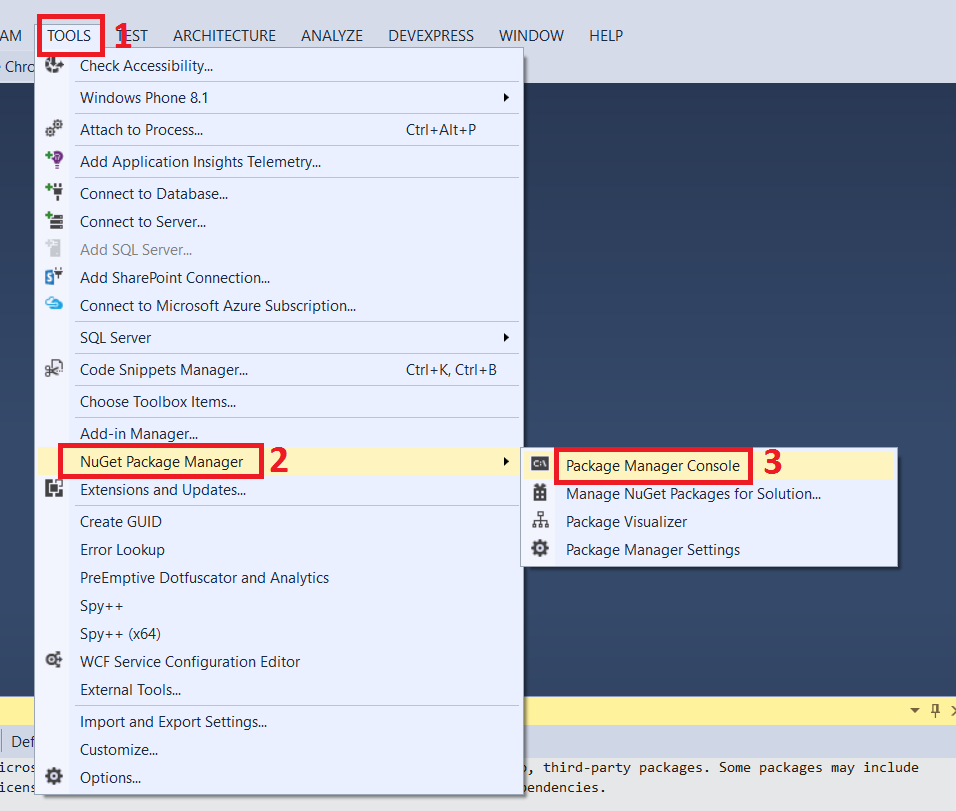
Tạo project web ASP.NET Web Application MVC:



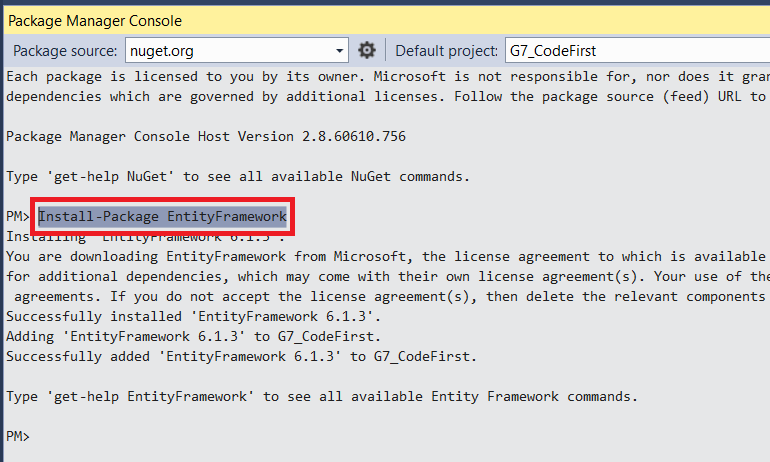


Cài đặt Entity Framework vào project.

Cách cài đặt bằng Package manager console:



Nhập “Install-Package EntityFramework” vào console sau đó enter.



Tạo 3 class như sau ở trong thư mục Models với các thuộc tính và phương thức get set như sau.

public class Student

{

[Key]

public int StudentID { get; set; }

public string StudentName { get; set; }

public int StudentAge { get; set; }

public float StudentGrade { get; set; }

public int StudentYearOfGraduation { get; set; }

}

public class Subject

{

[Key]

public int SubjectID { get; set; }

public string SubjectName { get; set; }

public int SubjectNumOfStudent { get; set; }

}

public class RegisterSubject

{

[Key]

public int RegSubID { get; set; }

public int? StudentID { get; set; }

[ForeignKey("StudentID")]

public virtual Student Student { get; set; }

public int? SubjectID { get; set; }

[ForeignKey("SubjectID")]

public virtual Subject Subject { get; set; }

}

Tạo class SchoolContext kế thừa từ class DbContext (cũng trong thư mục Models).

public class SchoolContext : DbContext

{

public SchoolContext()

: base("name=SchoolContext")

{

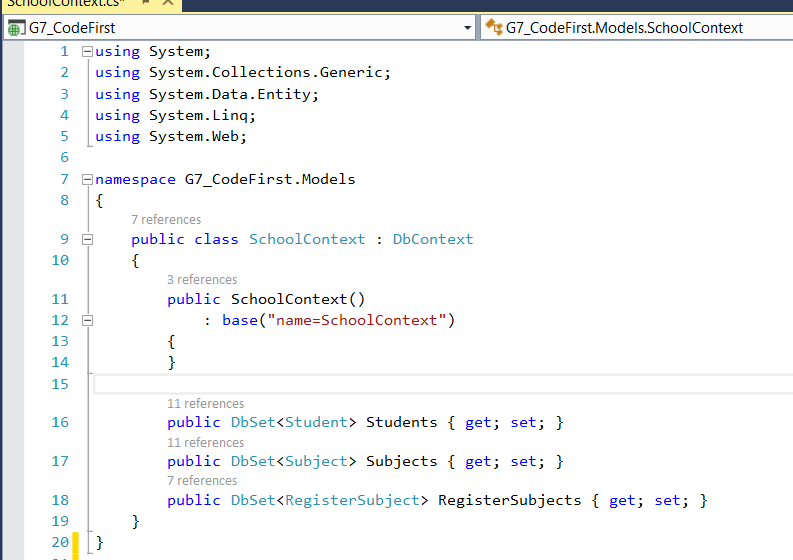
}

public DbSet<Student> Students { get; set; }

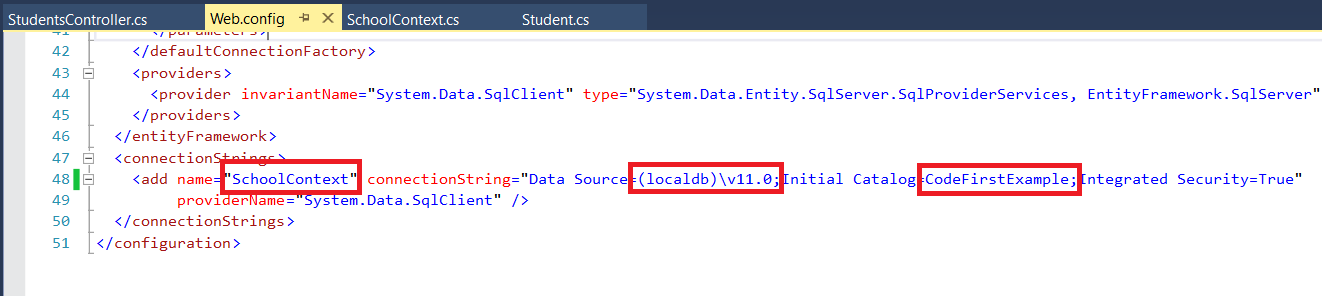
public DbSet<Subject> Subjects { get; set; }

public DbSet<RegisterSubject> RegisterSubjects { get; set; }

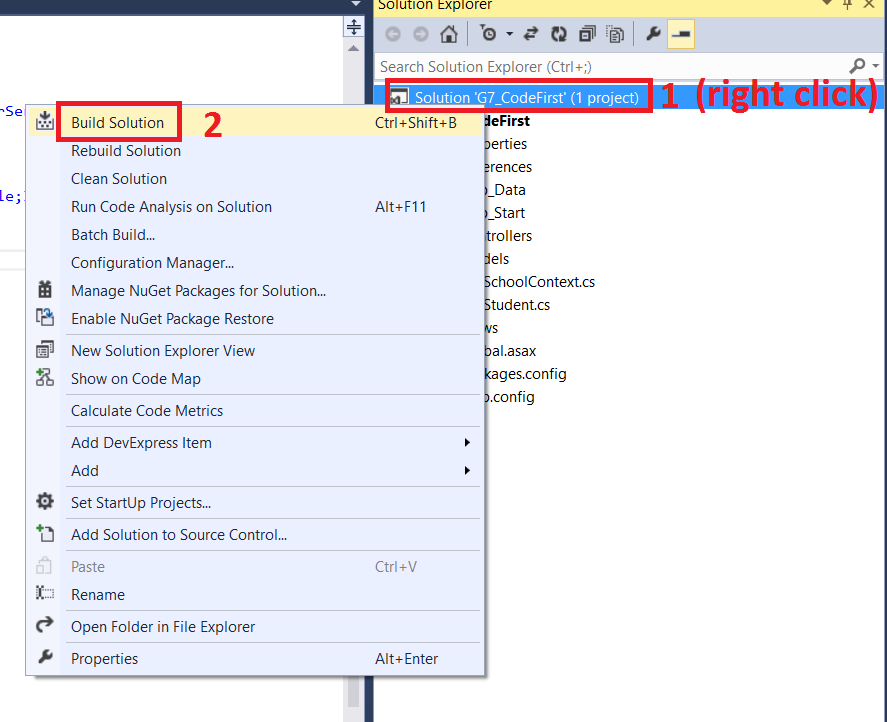
}



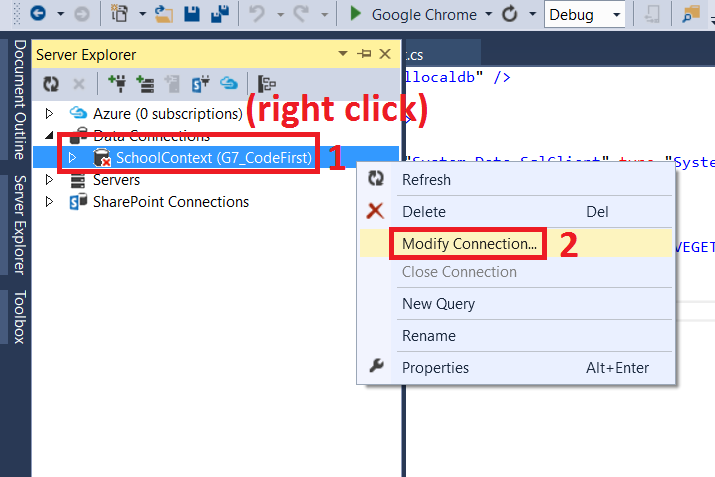
Khai báo chuỗi connectionStrings như sau để Code First làm việc được với SQL Server ở trong file Web.config.

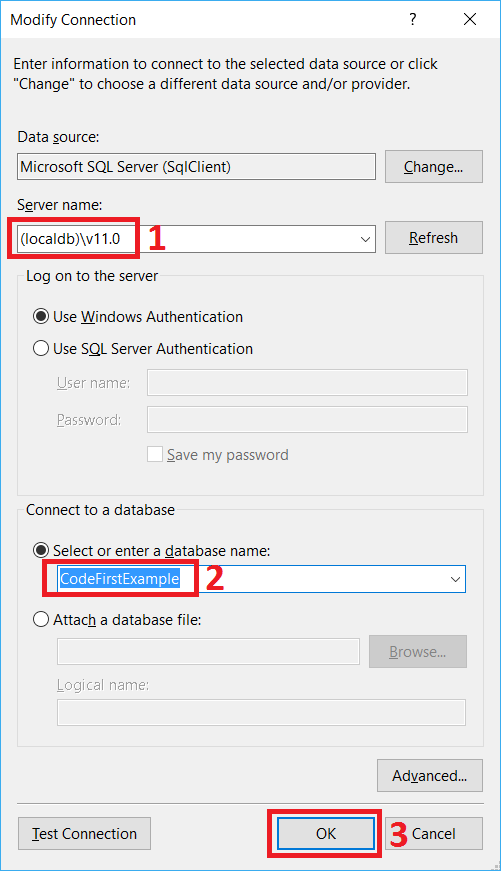


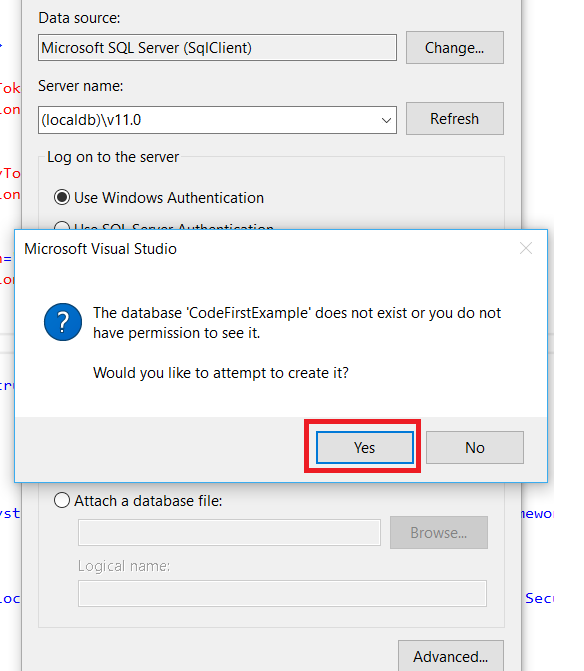
Build lại project.



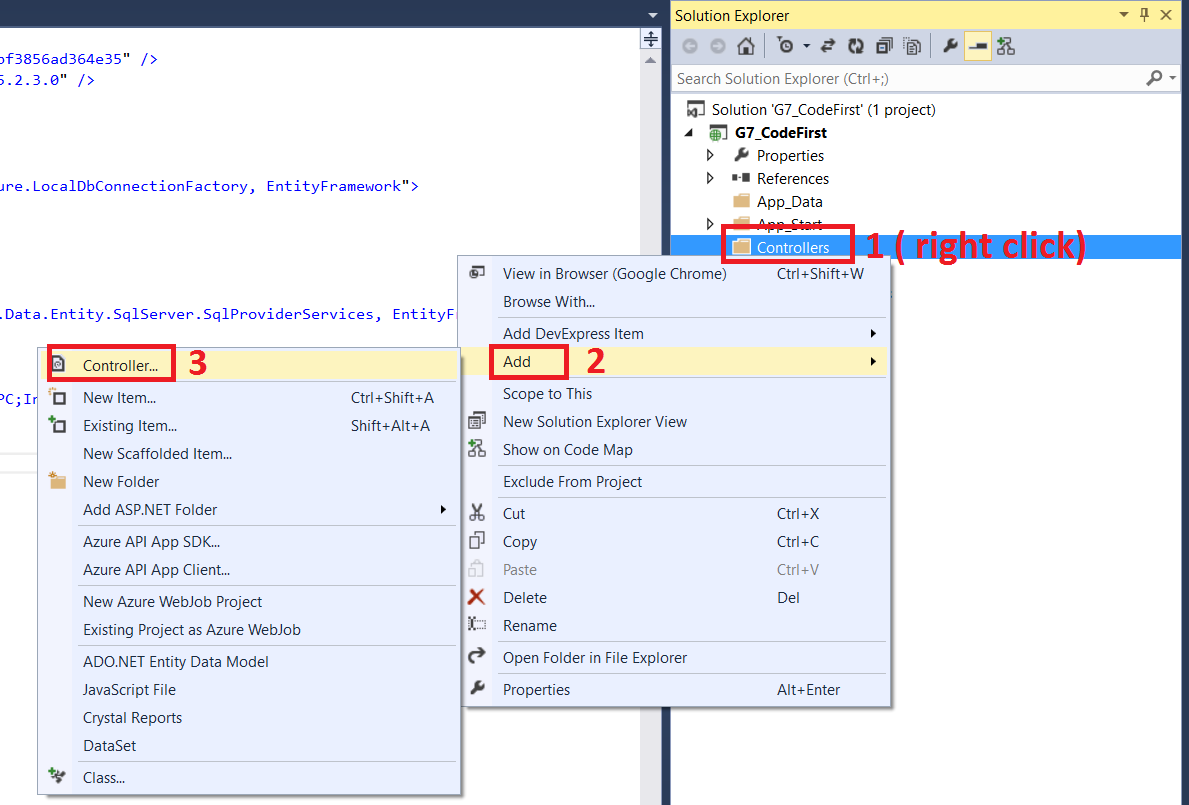
Tạo database (nếu chưa có)

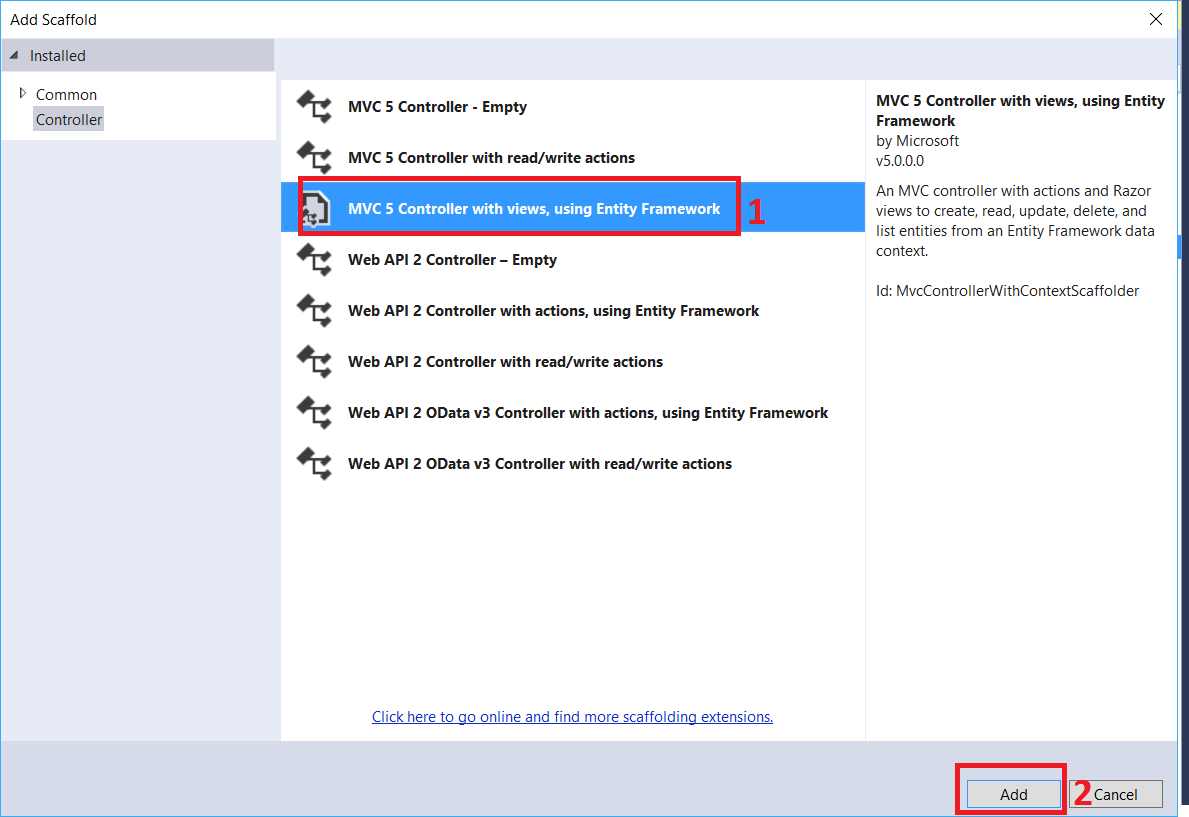


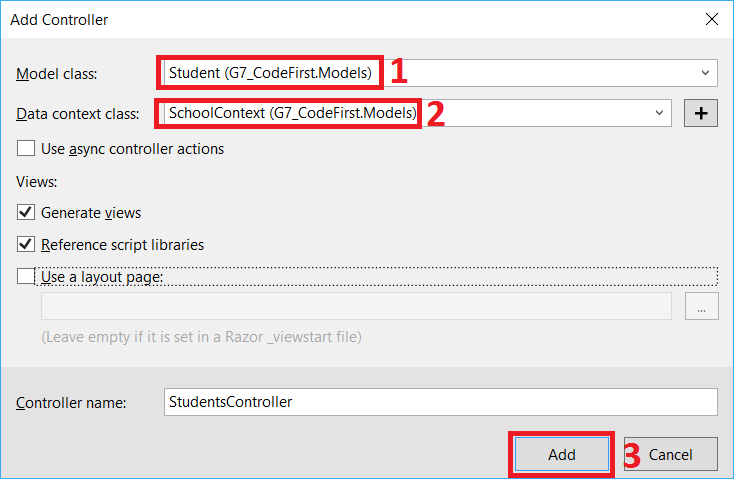




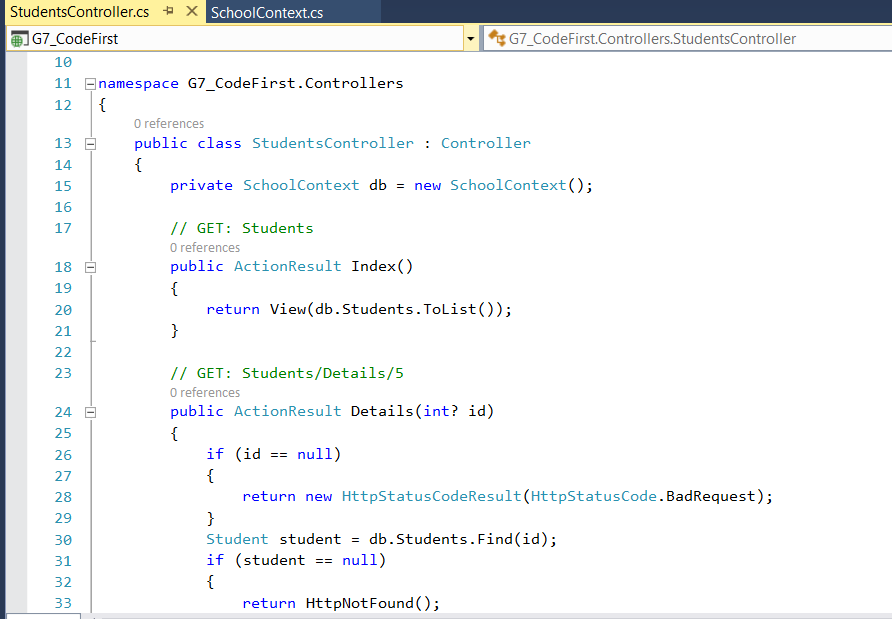
Thêm controller cho model vừa tạo.



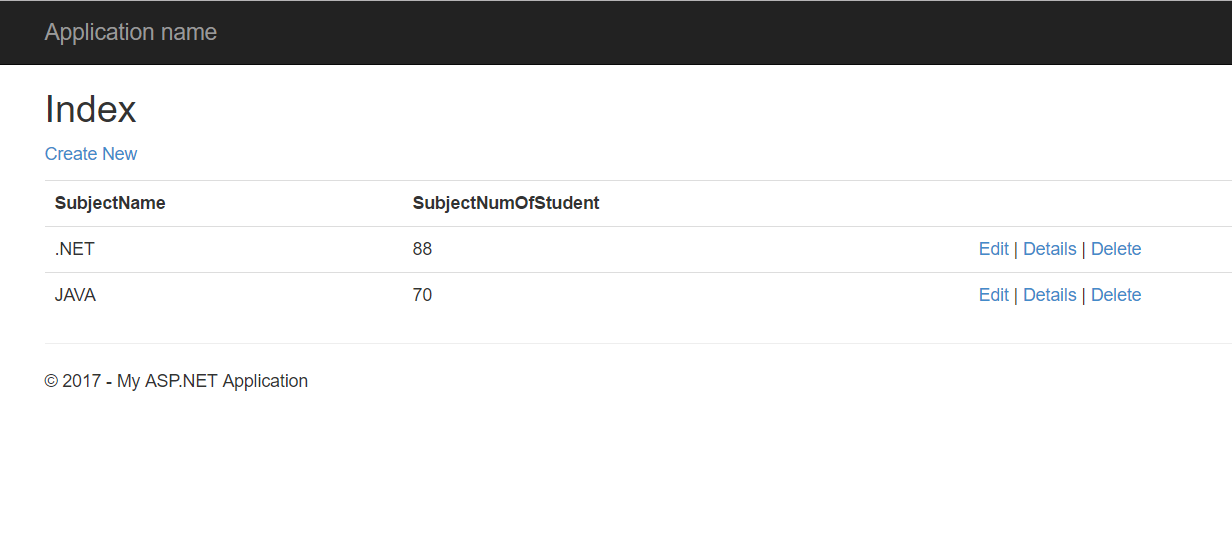




Controller sẽ được tự động gen ra các phương thức CRUD.



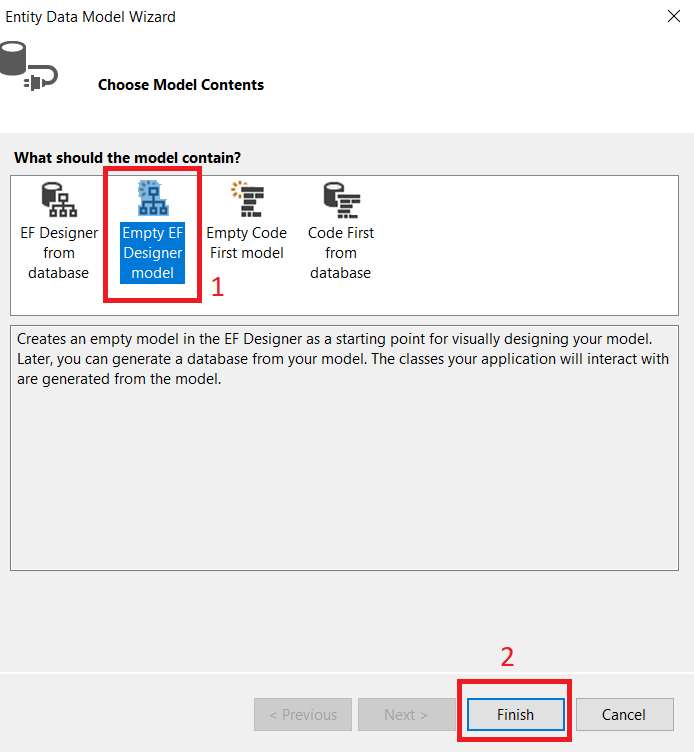
Làm tương tự với 2 class còn lại. Kết quả hiển thị.



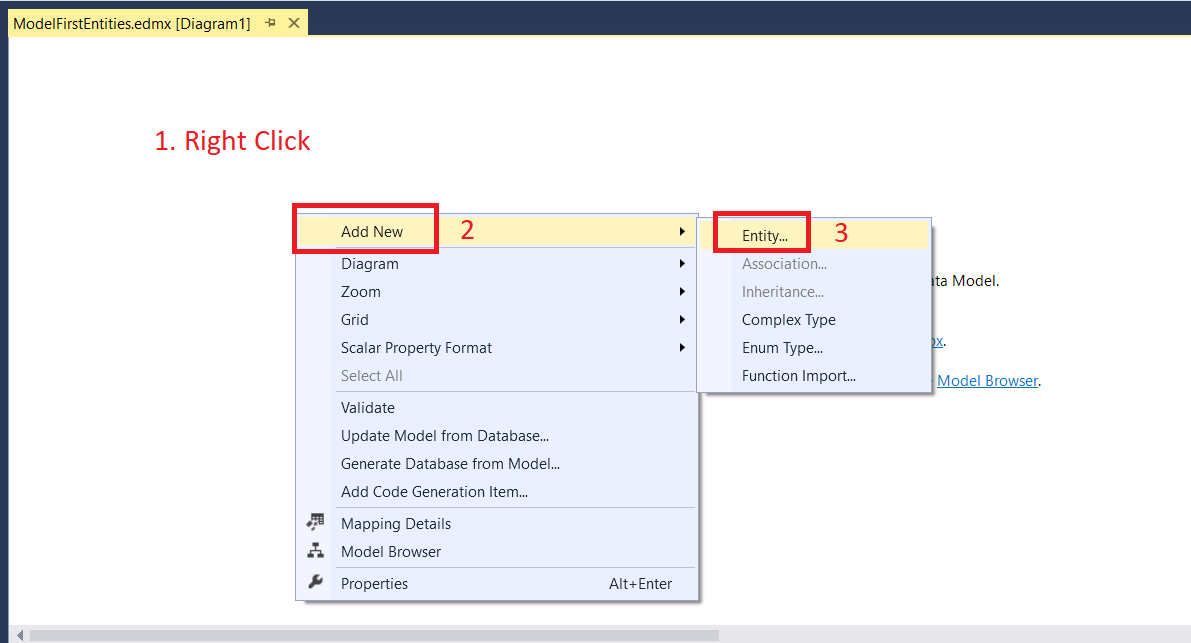
1. **Model first**

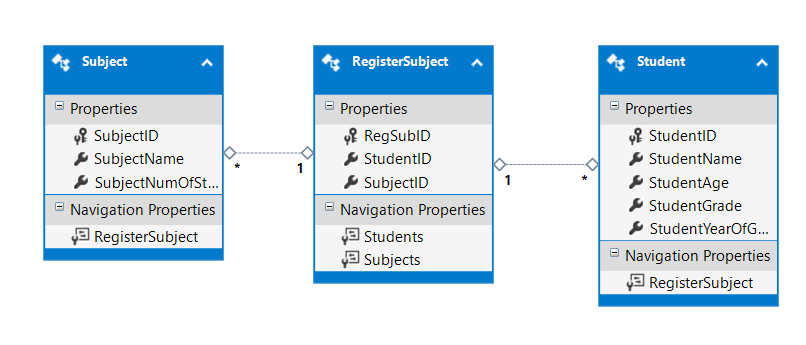
Tạo project mới giống như CodeFirst.

Sau đó thêm mới item EDM đặt tên là ModelFirstEntities trong thư mục Models như sau:

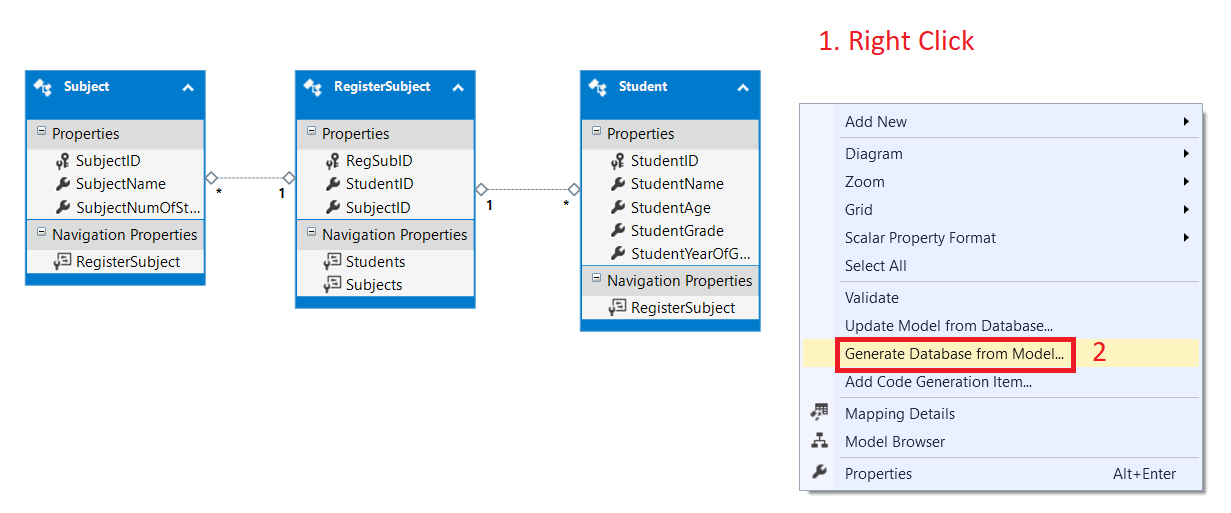


Sau đó tạo 3 entity gồm Student, Subject, RegisterSubject với khoá chính, khoá ngoại và các thuộc tính như sau:

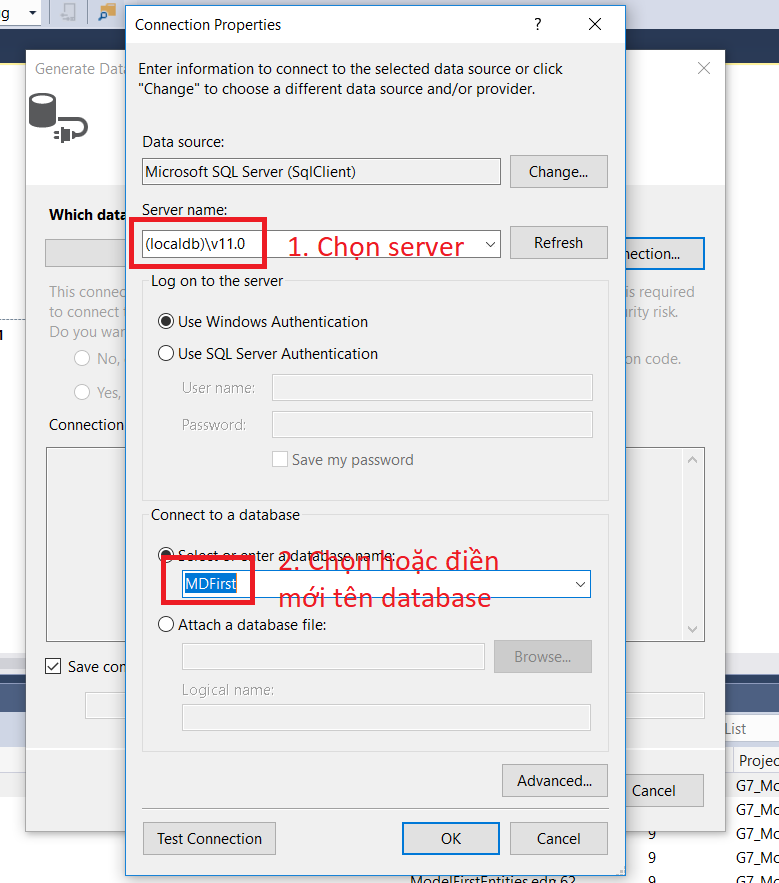




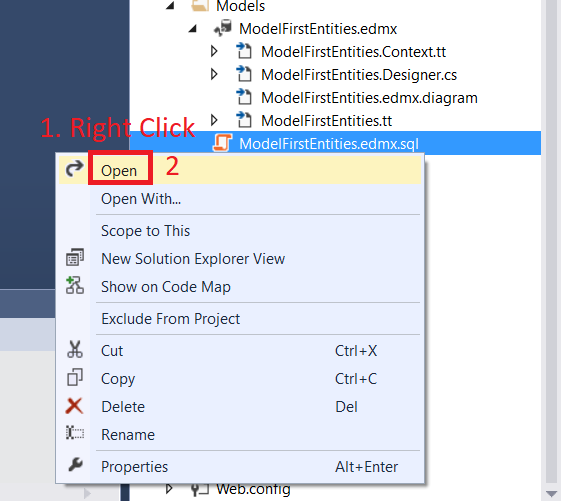
Sau khi tạo model, tiến hành generate database từ model:

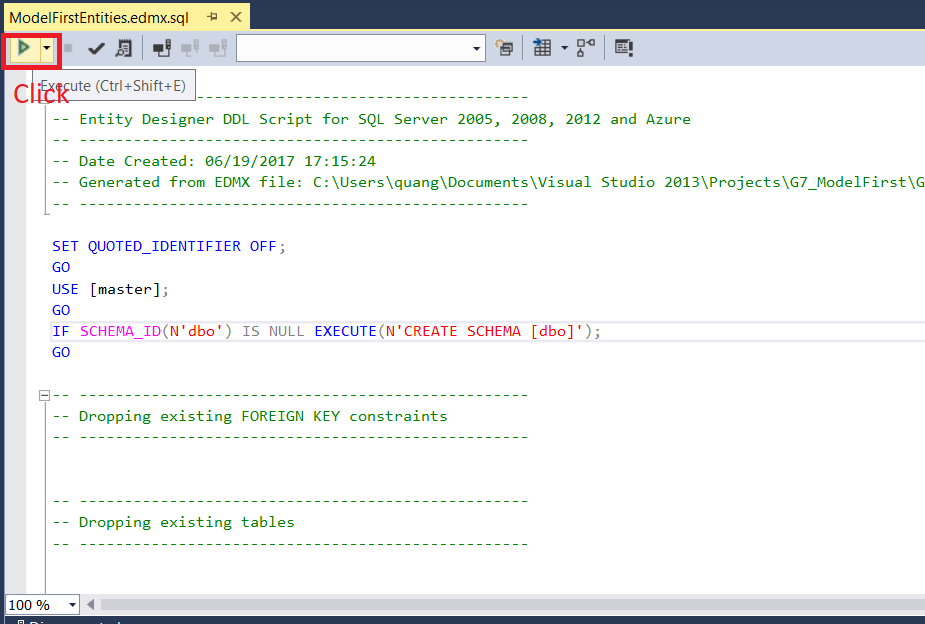


Chọn CSDL



Sau đó chạy file ModelFirstEntities.edmx.sql để thực thi các câu lệnh tạo bảng:





Các bước tiếp theo để tạo website tương tự CodeFirst.

Tiếp tục từ bước tạo Controller.

1. **Database first**

Tạo database như sau hoặc sử dụng file: /SOURCE CODE/dbfrist.sql

CREATE TABLE [dbo].[Students](

[StudentID] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,

[StudentName] [nvarchar](max) NULL,

[StudentAge] [int] NULL,

[StudentGrade] [real] NULL,

[StudentYearOfGraduation] [int] NULL

);

CREATE TABLE [dbo].[Subjects](

[SubjectID] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,

[SubjectName] [nvarchar](max) NULL,

[SubjectNumOfStudent] [int] NULL

);

CREATE TABLE [dbo].[RegisterSubjects](

[RegSubID] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,

[RegStudent\_StudentID] [int] NULL,

[RegSubject\_SubjectID] [int] NULL

);

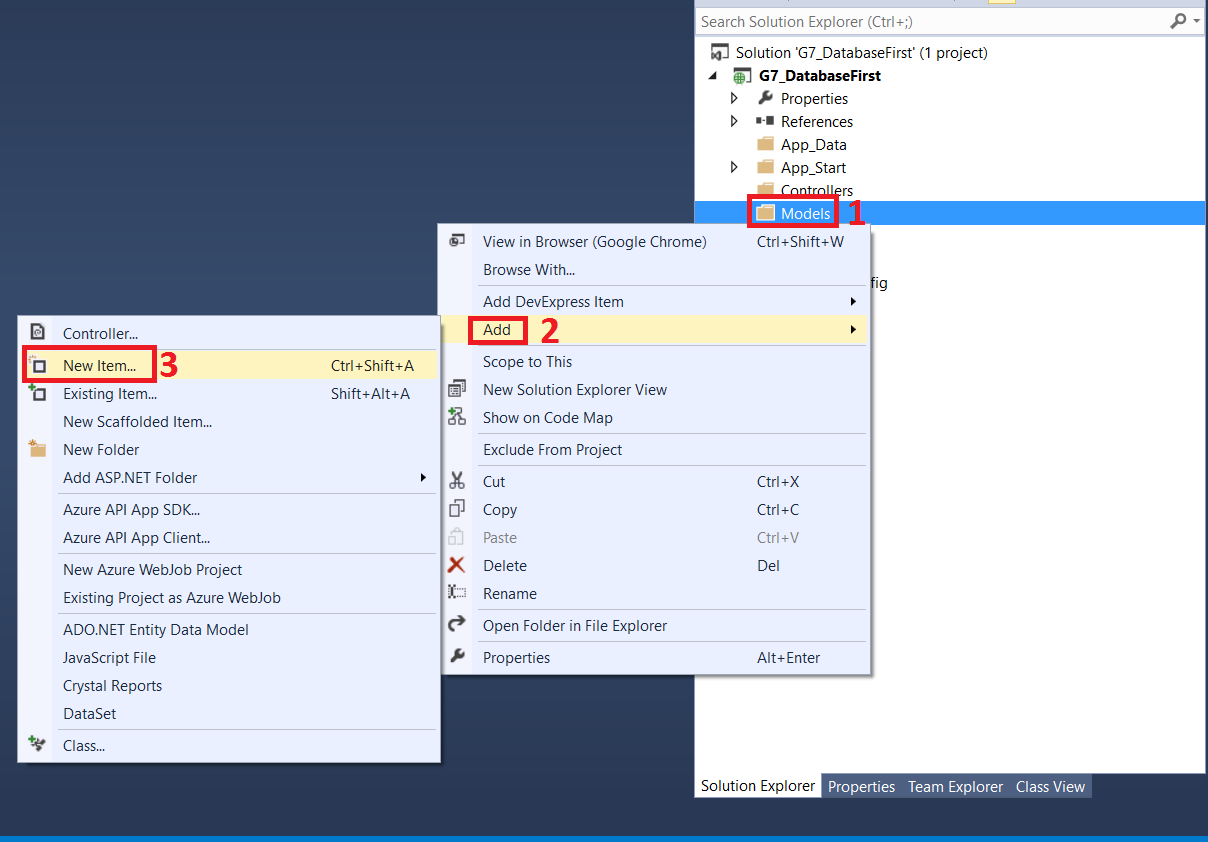
ALTER TABLE [dbo].[RegisterSubjects] ADD CONSTRAINT [FK\_RegisterSubjects\_Students] FOREIGN KEY([RegStudent\_StudentID])

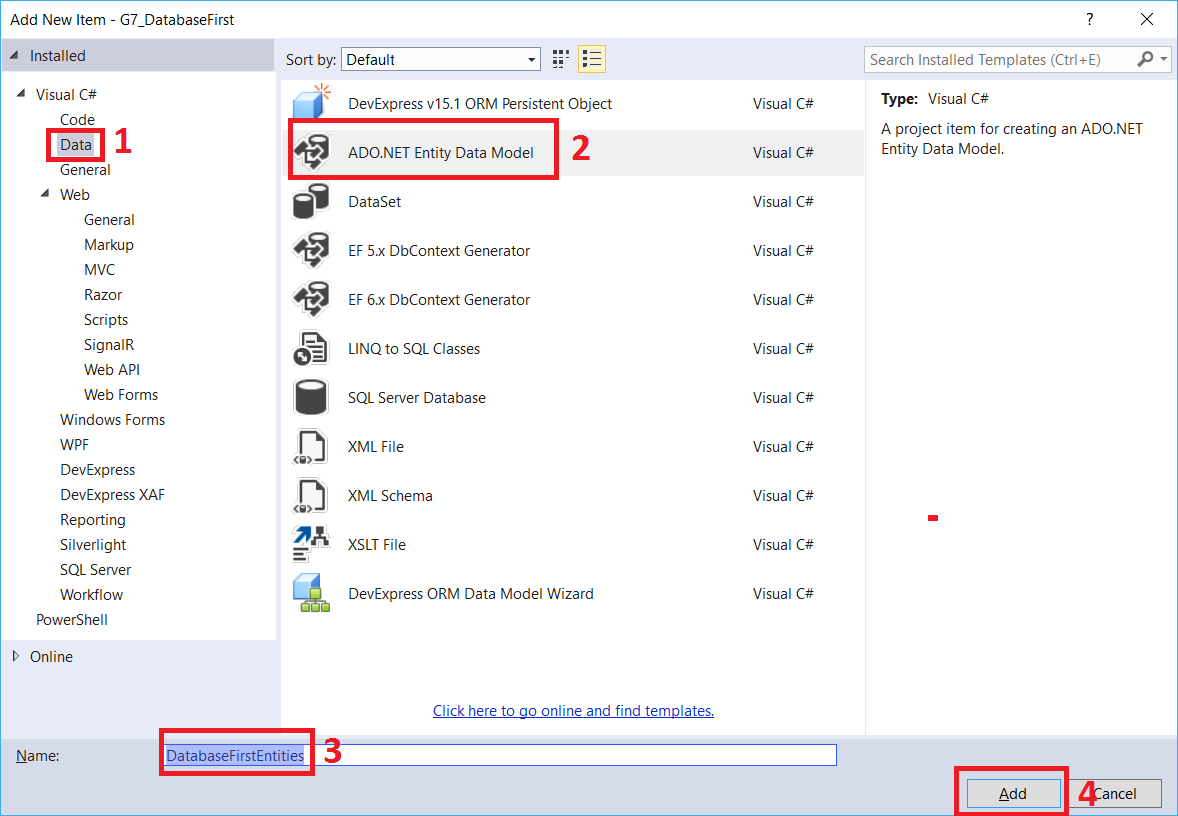
REFERENCES [dbo].[Students]([StudentID])

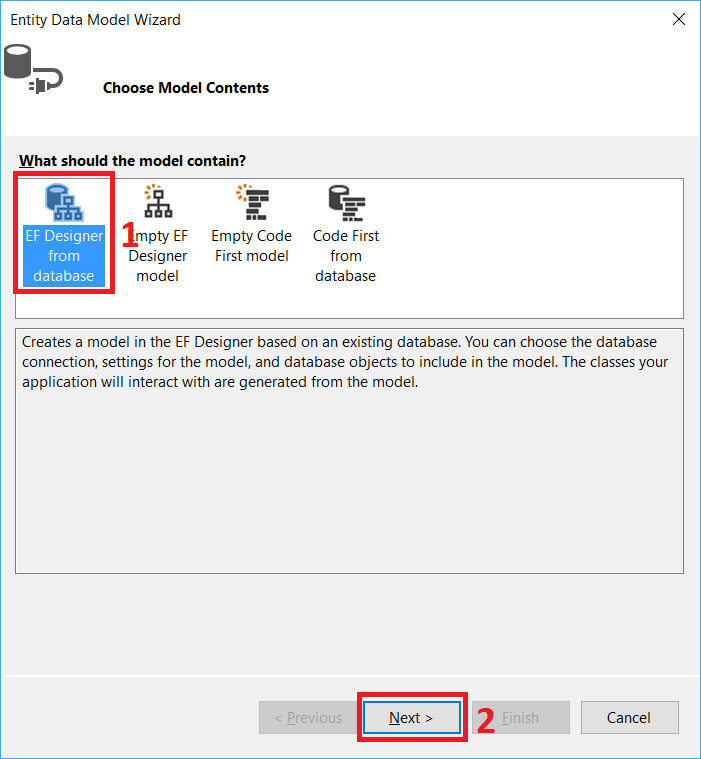
ALTER TABLE [dbo].[RegisterSubjects] ADD CONSTRAINT [FK\_RegisterSubjects\_Subjects] FOREIGN KEY([RegSubject\_SubjectID])

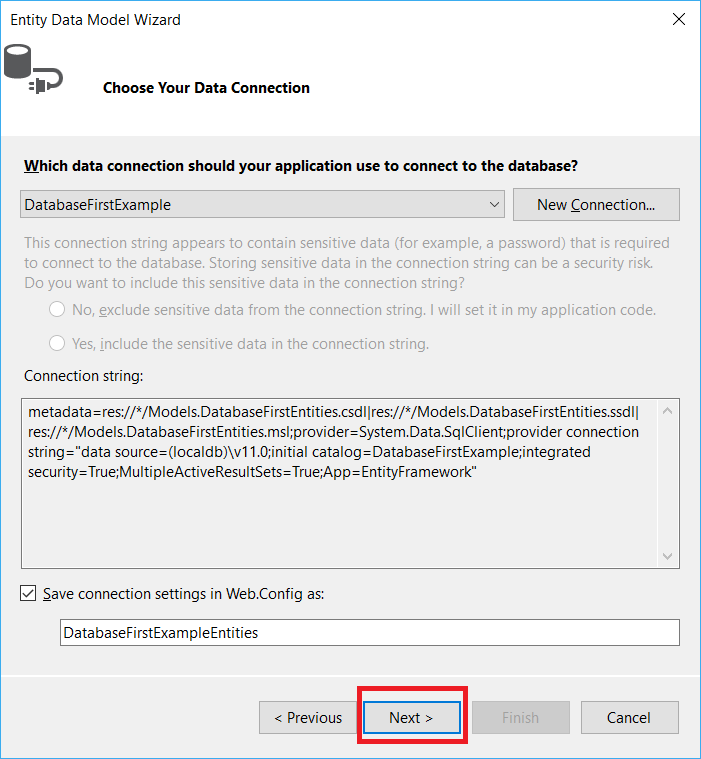
REFERENCES [dbo].[Subjects] ([SubjectID])

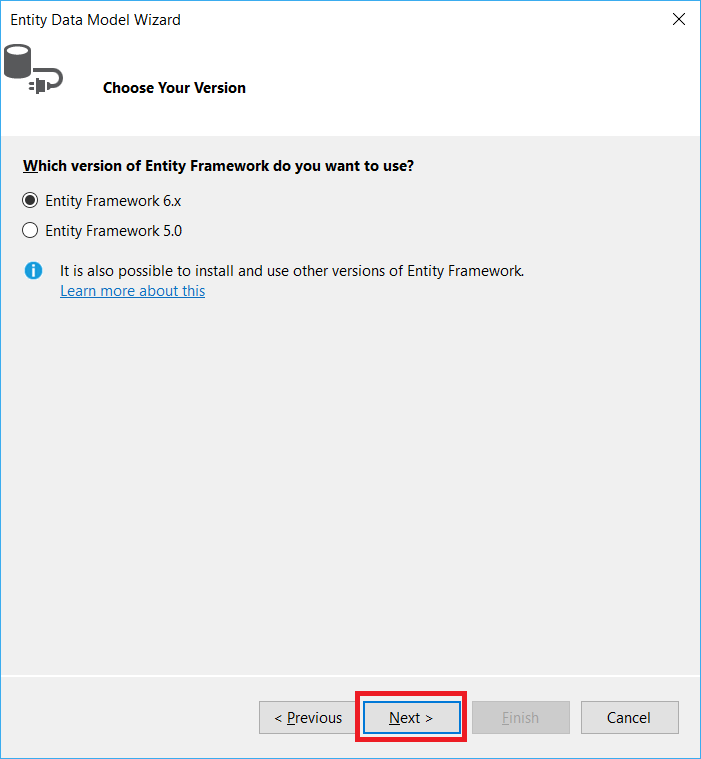
Thêm item EDM vào thư mục Models.



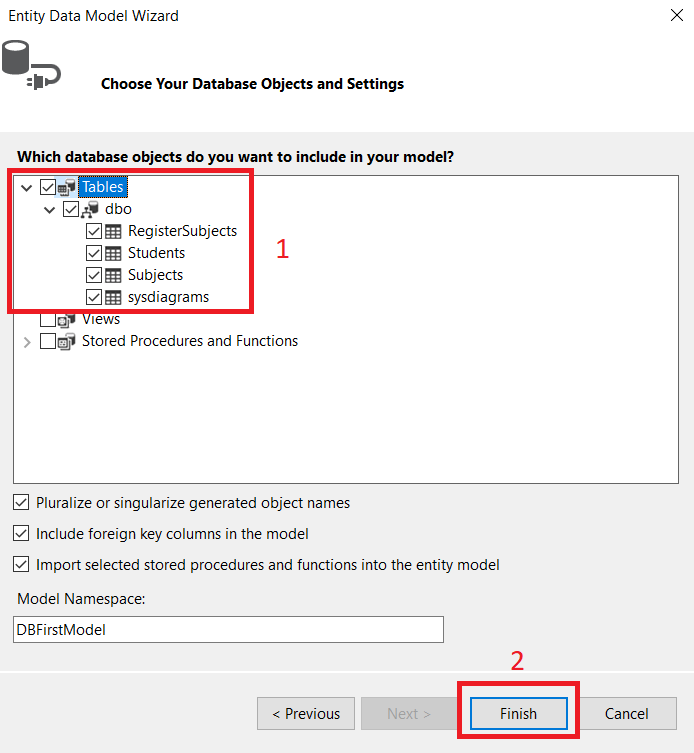








Chọn table cần sử dụng.



Build project.

Các bước thêm controller tương tự Code First.

Kết quả

