**4 tính chất đặc thù của lập trình hướng đối tượng**

Với [lập trình hướng đối tượng](http://vi.wikipedia.org/wiki/L%E1%BA%ADp_tr%C3%ACnh_h%C6%B0%E1%BB%9Bng_%C4%91%E1%BB%91i_t%C6%B0%E1%BB%A3ng) (OOP), có 2 câu hỏi thực dụng:

* Nó có những tính chất đặc thù gì? Để ta có thể dễ nhớ, dễ hiểu, dễ hình dung về nó.
* Khi học một ngôn ngữ OOP nào đó, làm sao để kiểm tra xem ta đã nắm bắt được ngôn ngữ này hay chưa, bằng cách viết thử code để diễn tả các đặc thù này bằng cú pháp của ngôn ngữ này?

## Trả lời câu hỏi 1

[OOP](http://vi.wikipedia.org/wiki/L%E1%BA%ADp_tr%C3%ACnh_h%C6%B0%E1%BB%9Bng_%C4%91%E1%BB%91i_t%C6%B0%E1%BB%A3ng) có 4 tính chất đặc thù chính, các ngôn ngữ OOP nói chung đều có cách để diễn tả:

* Tính đóng gói: Có thể gói dữ liệu (data, ~ biến, trạng thái) và mã chương trình (code, ~ phương thức) thành một cục gọi là lớp (class) để dễ quản lí. Trong cục này thường data rất rối rắm, không tiện cho người không có trách nhiệm truy cập trực tiếp, nên thường ta sẽ che dấu data đi, chỉ để lòi phương thức ra ngoài. Ví dụ hàng xóm sang mượn búa, thay vì bảo hàng xóm cứ tự nhiên vào lục lọi, ta sẽ bảo: "Ấy bác ngồi chơi để tôi bảo cháu lấy cho". Ngôn ngữ Ruby "phát xít" đến nỗi dấu tiệt data, cấm không cho truy cập từ bên ngoài. Ngoài ra, các lớp liên quan đến nhau có thể được gom chung lại thành package (tùy ngôn ngữ mà còn gọi là module, namespace v.v.).
* Tính trừu tượng: Có câu "program to interfaces, not to concrete implementations". Nghĩa là khi viết chương trình theo phong cách hướng đối tượng, khi thiết kế các đối tượng, ta cần rút tỉa ra những đặc trưng của chúng, rồi trừu tượng hóa thành các interface, và thiết kế xem chúng sẽ tương tác với nhau như thế nào. Nói cách khác, chúng ta định ra các interface và các contract mà chúng cần thỏa mãn.
* Tính thừa kế: Lớp cha có thể chia sẻ dữ liệu và phương thức cho các lớp con, các lớp con khỏi phải định nghĩa lại những logic chung, giúp chương trình ngắn gọn. Nếu lớp cha là interface, thì lớp con sẽ di truyền những contract trừu tượng từ lớp cha.
* Tính đa hình: Đối tượng có thể thay đổi kiểu (biến hình). (1) Với các ngôn ngữ OOP có kiểu, có thể mượn phát biểu của C++ "con trỏ kiểu lớp cha có thể dùng để trỏ đến đối tượng kiểu lớp con". Như vậy khi khai báo chỉ cần khai báo p có kiểu lớp cha, còn sau đó nó trỏ đến đâu thì kệ cha con nó: nếu cha và con cùng có phương thức m, thì từ p cứ lôi m ra gọi thì chắc chắn gọi được, không cần biết hiện tại p đang trỏ đến cha hay con. Khi lớp B thừa kế từ lớp A, thì đối tượng của lớp B có thể coi là đối tượng của lớp A, vì B chứa nhiều thứ thừa kế từ A. (2) Với ngôn ngữ OOP không có kiểu như Ruby, có thể mượn phát biểu của [phương pháp xác định kiểu kiểu con vịt](http://en.wikipedia.org/wiki/Duck_typing): "nếu p đi như vịt nói như vịt, thì cứ coi nó là vịt". Như vậy nếu lớp C có phương thức m, mà có thể gọi phương thức m từ đối tượng p bất kì nào đó, thì cứ coi p có kiểu là C.

Để dễ nhớ, có thể chia 4 đặc thù làm 2 nhóm:

1. Nhóm 1: tính chất 1. Tính đóng gói là tính dễ nhận thấy nhất nếu bạn bắt đầu học OOP sau khi đã học qua những ngôn ngữ thủ tục như C và Pascal (thường trường phổ thông ở Việt Nam đều dạy).
2. Nhóm 2: tính chất 2, 3, và 4 đi một dây với nhau.

## Trả lời câu hỏi 2

Nên làm bài tập nhỏ về các con vật sau, gọi tắt là bài 4OOP:

* Tạo interface Animal có phương thức say\_hello. <- Thể hiện tính trừu tượng, có nghĩa ta định ra contract là rằng dù là con vật gì đi nữa thì nó cũng có phương thức say\_hello để chào hỏi gì đấy.
* Tạo 2 lớp Cat và Dog kế thừa từ Animal. Khi khởi tạo chúng sẽ có tên. Chúng override lại phương thức say\_hello để chào hỏi theo cách riêng của chúng. <- Thể hiện tính đóng gói (đóng gói biến tên và phương thức say\_hello với nhau) và tính thừa kế (Cat và Dog mang đặc điểm chung là có say\_hello từ Animal).
* Tạo lớp Zoo để quản lí nhiều Animal, có (1) phương thức add, remove để thêm, bớt các Animal (các đối tượng của các lớp thừa kế từ Animal), (2) phương thức say\_hello\_all để gọi say\_hello của tất cả đối tượng nó quản lí. <- Thể hiện tính đa hình, Zoo gọi chỉ gọi một phương thức say\_hello, nhưng tùy con vật mà lời chào hỏi sẽ khác nhau.

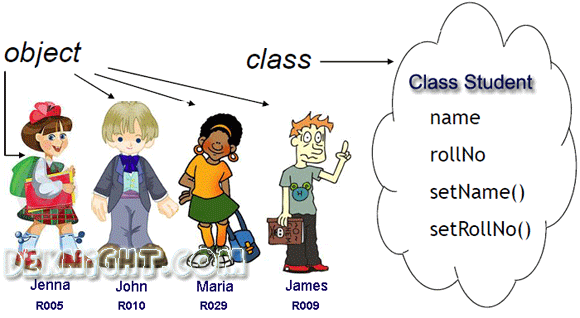
Đây là bài tập gối đầu giường, khi được yêu cầu viết phải viết được ngay không ngắc ngứ.

Lập trình hướng đối tượng là gì?

Lập trình hướng đối tượng là một kỹ thuật lập trình cho phép lập trình viên tạo ra các đối tượng trong code **trừu tượng hóa** các đối tượng thực tế trong cuộc sống. Hướng tiếp cận này hiện đang rất thành công và đã trở thành một trong những khuôn mẫu phát triển phần mềm, đặc biệt là các phần mềm cho doanh nghiệp.

Khi phát triển ứng dụng sử dụng OOP, chúng ta sẽ định nghĩa các lớp **(class)** để mô hình các đối tượng thực tế. Trong ứng dụng **các lớp** này sẽ được **khởi tạo** thành các **đối tượng** và trong suốt thời gian ứng dụng chạy, các phương thức (**method**) của đối tượng này sẽ được gọi.

Lớp định nghĩa đối tượng sẽ như thế nào: gồm những **phương thức** và**thuộc tính** (**property**) gì. **Một đối tượng chỉ là một thể hiện của lớp**. Các lớp tương tác với nhau bởi các **public API**: **là tập các phương thức, thuộc tính public** của nó.



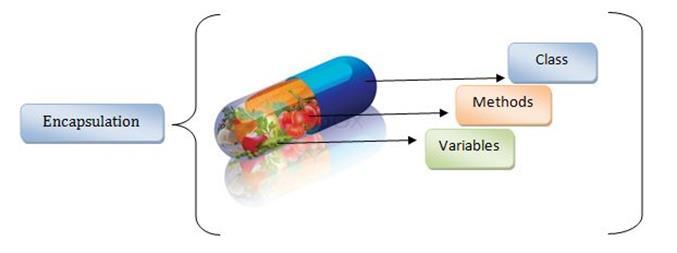
OOP có **3 nguyên lý cơ bản** chúng ta sẽ cùng tìm hiểu chi tiết sau đây đó là:

Tính đóng gói (Encapsulation)

Tính đóng gói tức là quy tắc yêu cầu trạng thái bên trong của một đối tượng được bảo vệ và tránh truy cập được từ  code bên ngoài (tức là code bên ngoài không thể trực tiếp nhìn thấy và thay đổi trạng thái của đối tượng đó). Bất cứ truy cập nào tới trạng thái bên trong này bắt buộc phải thông qua một public API để đảm bảo trạng thái của đối tượng luôn hợp lệ bởi vì các public API chịu trách nhiệm thực hiện kiểm tra tính hợp lệ cũng như trình tự cập nhật trạng thái của đối tượng đó.

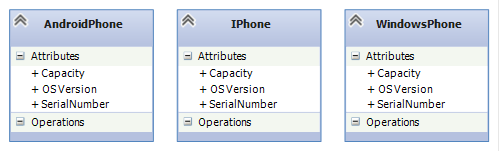
Nói chung trạng thái đối tượng không hợp lệ thường do: chưa được kiểm tra tính hợp lệ, các bước thực hiện không đúng trình tự hoặc bị bỏ qua nên **trong OOP có một quy tắc quan trọng cần nhớ đó là phải luôn khai báo các trạng thái bên trong của đối tượng là private và chỉ cho truy cập qua các public/protected method/property**. Khi sử dụng các đối tượng ta không cần biết bên trong nó làm việc như thế nào, ta chỉ cần biết các public API là gì và điều này đảm bảo những gì thay đổi đối tượng sẽ được kiểm tra bởi các quy tắc logic bên trong, tránh đối tượng bị sử dụng không chính xác.

Nguyên lý đóng gói như thế này ở đâu ta cũng có thể bắt gặp ví dụ như thiết kế viên thuốc, chúng ta chỉ biết nó chữa bệnh này, bệnh kia và một số thành phần chính còn cụ thể bên trong nó có những gì thì hoàn toàn không biết.

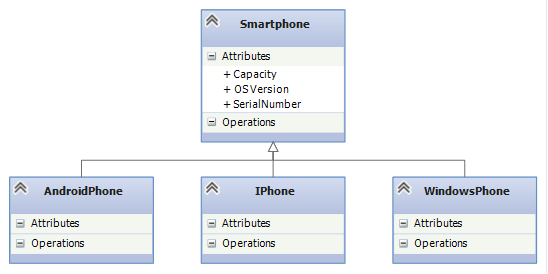


**Tính kế thừa (Inheritance)**

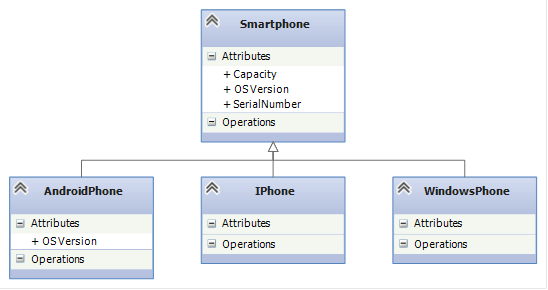
Khi bắt đầu xây dựng ứng dụng chúng ta sẽ bắt đầu việc thiết kế các lớp, thông thường chúng ta sẽ thấy có trường hợp một số lớp dường như có quan hệ với những lớp khác, chúng có những đặc tính khá giống nhau. VD: 3 lớp AndroidPhone, IPhone, WindowsPhone ở hình dưới:



Mỗi lớp đều đại diện cho một loại smartphone khác nhau nhưng lại có những thuộc tính giống nhau. Thay vì sao chép những thuộc tính này, sẽ hay hơn nếu ta đặt chúng ở một nơi có thể dùng bởi những lớp khác. Điều này được thực hiện bởi tính kế thừa trong OOP: chúng ta có thể định nghĩa lớp cha – base class (trong trường hợp này là Smartphone ) và có những lớp con kế thừa từ nó (derived class), tạo ra một mối quan hệ cha/con như hình dưới:



Bây giờ, các lớp con có thể kế thừa 3 thuộc tính từ lớp cha. Nếu các chức năng của lớp cha đã được định nghĩa đầy đủ thì lập trình viên sẽ không phải làm bất cứ việc gì ở lớp con. Còn nếu một lớp con muốn chức năng khác so với định nghĩa ở lớp cha thì nó có thể ghi đè (**override**) chức năng đã được định nghĩa trên lớp cha này.



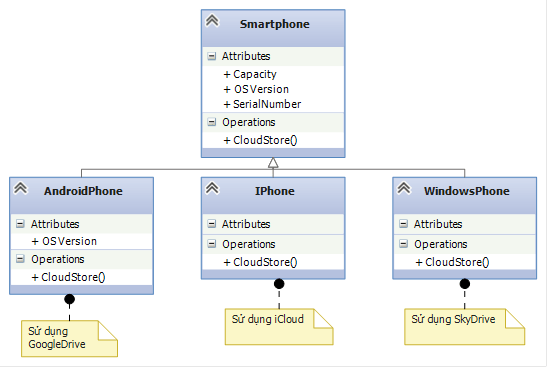
Như hình trên bây giờ nếu gọi thuộc tính **OSVersion** trên lớp **AndroidPhone** thì nó sẽ dùng định nghĩa được khai báo ở lớp này, trong khi hai lớp Iphone & Windows Phone vẫn dùng định nghĩa được khai báo ở lớp cha Smartphone.

**Tính đa hình (Polymorphism)**

Với đa số lập trình viên thì tính Kế thừa và Đóng gói trong OOP khá dễ hiểu còn tính Đa hình khi mới tiếp cận sẽ thấy khó hiểu hơn một chút. Tuy nhiên đây lại là một tính chất có thể nói là chứa đựng hầu hết sức mạnh của lập trình hướng đối tượng. Hiểu một cách đơn giản: **Đa hình là khái niệm mà hai hoặc nhiều lớp có những phương thức giống nhau nhưng có thể thực thi theo những cách thức khác nhau.**

Ví dụ như ở phần trên, mỗi một smartphone kế thừa từ lớp Smartphone nhưng có thể lưu trữ dữ liệu trên cloud theo những cách khác nhau:

* AndroidPhone lưu trữ bằng Google Drive
* Iphone lưu trên iCloud
* WindowsPhone sử dụng SkyDrive.

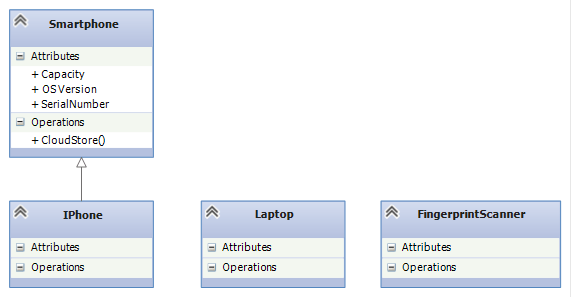


Bởi vì tất cả đều là **Smartphone** nên nếu ta viết một hàm **dùng kiểu Smartphone làm tham số** thì khi gọi hàm ta có thể truyền vào một đối tượng kiểu AndroidPhone, Iphone hoặc WindowsPhone bởi vì chúng đều kế thừa từ lớp Smartphone nên được chấp nhận (hiểu nôm na một AndroidPhone, Iphone, WindowsPhone cũng là một Smartphone). Bên cạnh đó hàm này thậm chí không cần quan tâm smartphone nào được truyền vào do nó chỉ cần biết đối tượng đang xử lý ở đây là Smartphone với những public method/property đã được định nghĩa. **Nếu các lớp con không định nghĩa lại (overrides) phương thức CloudStore() thì phương thức CloudStore() trên lớp cha (Smartphone) sẽ được gọi. Còn nếu lớp con override lại phương thức CloudStore() của lớp cha như ở hình trên thì phương thức CloudStore() trên lớp con sẽ được gọi mặc dù code trong hàm đang thao tác với đối tượng kiểu Smartphone.**

Tính Đa hình như trên là một tính chất rất mạnh mẽ bởi vì nó mang lại cho code khả năng tổng quát hóa cao. **Chúng ta không cần tạo ra phương thức cho mỗi kiểu kế thừa từ lớp cha Smartphone mà chỉ cần nhận một biến kiểu Smartphone và có thể làm việc với bất cứ lớp nào kế thừa từ nó**. Điều duy nhất không làm được ở đây là sử dụng những phương thức mà chỉ được khai báo trên các lớp con. VD: nếu ta có một phương thức trên lớp IPhone gọi là **OpenSiri()** nhưng không được khai báo trên lớp Smartphone, khi đó muốn gọi nó sẽ bắt buộc phải ép kiểu từ Smartphone sang IPhone trước khi gọi.

**Interface**

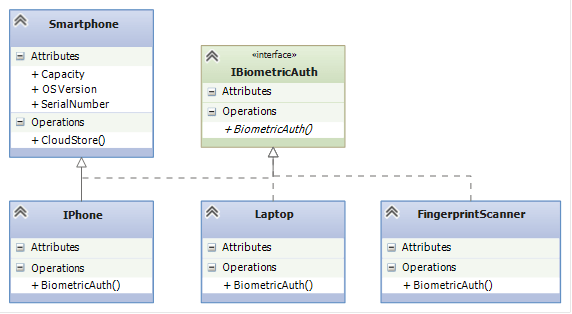
Đa hình dựa trên Kế thừa không phải bao giờ cũng là lựa chọn tốt nhất. VD: Ta có biểu đồ lớp như sau



Ta thấy rõ ràng rằng 3 lớp bên dưới (Iphone, Laptop, FingerprintScanner) đều là những thứ có thể truy cập được bằng vân tay nhưng chúng thực hiện theo những cách khác nhau. Những lớp này có chung một hành động tạm gọi là định danh bằng sinh trắc học – **BiometricAuth**(). Nếu ta cố gắng gộp cả 3 lớp này vào thành 1 lớp chung sẽ không hay vì rất khó để tìm ra điểm chung tổng quát của chúng ngoài việc có thể truy cập bằng vân tay.

Do vậy thay vì sử dụng Kế thừa ở đây, ta có thể sử dụng một kỹ thuật khác đó là Interface. **Interface đơn giản là một giao ước chỉ ra rằng code của bạn sẽ thực thi và hỗ trợ một public API cụ thể nào đó**. Tuy nhiên các public API này được thực hiện như thế nào thì không được chỉ ra trên Interface mà sẽ được chỉ ra trên lớp thực thi interface này. Về cơ bản giao ước là một danh sách các public method/property mà chắc chắn sẽ được thực thi trong lớp của bạn.

Áp vào VD trên, ta có thể tạo ra một interface là **IBiometricAuth** với một phương thức là **BiometricAuth()**. Tiếp theo cho các lớp Iphone, Laptop, FingerprintScanner thực thi interface **IBiometricAuth** này như hình sau



Vì mỗi lớp trên đều thực thi interface **IBiometricAuth** nên ta có thể đảm bào rằng chúng đều có phương thức **BiometricAuth()** và khai báo của phương thức sẽ giống y như được định nghĩa trên interface **IBiometricAuth**. Tương tự Kế thừa dựa trên Đa hình, sử dụng Interface cho phép chúng ta khai báo phương thức nhận tham số kiểu **IBiometricAuth** nhưng chấp nhận bất cứ đối tượng nào truyền vào mà kiểu của nó thực thi interface **IBiometricAuth** này. Các lớp thực thi **IBiometricAuth** không cần phải có chung lớp cha ngoại trừ interface **IBiometricAuth**. Trong phương thức ở trên, ta có thể gọi bất cứ phương thức nào đã được định nghĩa trên interface **IBiometricAuth** của đối tượng truyền vào mà không cần quan tâm kiểu thực sự của nó là gì: Không cần quan tâm nó là Iphone, Laptop hay FingerprintScanner, chỉ cần biết nó hỗ trợ interface **IBiometricAuth** là có thể gọi được phương thức **BiometricAuth(),**hết sức linh hoạt và mềm dẻo.

**Tóm lại**

Trong bài viết này tôi đã tóm lược lại 3 nguyên lý cơ bản của lập trình hướng đối tượng sao cho đơn giản và dễ hiểu nhất. Mặc dù chúng rất cơ bản và hầu như ai học về lập trình cũng đã từng được học hoặc đọc nhưng hy vọng bài viết này sẽ mang đến một điều gì đó dễ dàng tiếp cận hơn cho các bạn, nhất là những lập trình viên mới tiếp cận OOP. Hãy chia sẻ phản hồi của bạn về bài viết và share cho bạn bè nếu bạn thấy nó hữu ích nhé. Nếu bạn có câu hỏi gì xin vui lòng comment dưới bài viết để chúng ta cùng thảo luận.

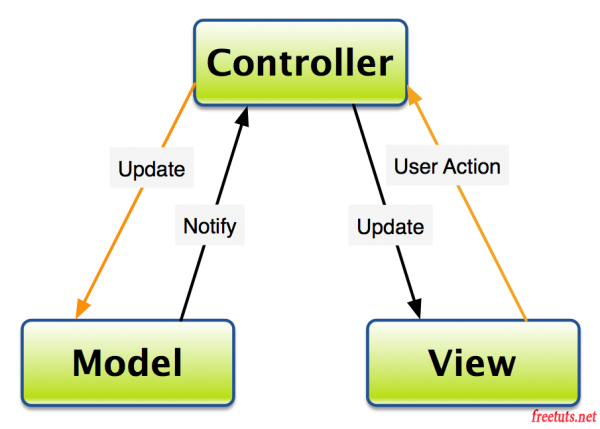
Sharing is learning. Sta

MVC là chữ viết tắt của **M**odel - **V**iew - **C**ontroller, đây là một mô hình kiến phần mềm được tạo ra với mục đích quản lý và xây dựng dự án phần mềm có hệ thống hơn. Mô hình này được dùng khá rộng rãi và đặc biệt là trong các ngôn ngữ lập trình web. Trong PHP hiện tại có khá nhiều Framework và tất cả đều xây dựng từ mô hình MVC, từ đó bạn có thể thấy sự quan trọng của nó như thế nào rồi đấy.

**Trong mô hình này thì**:

* **Model**: có nhiệm vụ thao tác với cơ sở dữ liệu, nghĩa là nó sẽ chứa tất cả các hàm, các phương thức truy vấn trực tiếp với dữ liệu và controller sẽ thông qua các hàm, phương thức đó để lấy dữ liệu rồi gửi qua **View**
* **View**: có nhiệm vụ tiếp nhận dữ liệu từ controller và hiển thị nội dung sang các đoạn mã HTML, bạn có thể hiểu nôm na đây người ta còn gọi là thành phần giao diện.
* **Controller**: đóng vài trò trung gian giữa Model và View. Nó có nhiệm vụ tiếp nhận yêu cầu từ client sau đó xử lý request, load model tương ứng và gửi data qua view tương ứng rồi trả kết quả về cho client

Để rõ ràng hơn thì bạn xem hình dưới đây:



Nhìn vào mô hình này các bạn thấy giữa model và view không hề có mối liên hệ mà nó sẽ thông qua controller để giao tiếp với nhau. Hiện trên mạng có khá nhiều mô hình vẽ ra nhưng mình thấy nó quá rắc rối nên mình chọn hình này cho bạn dễ hiểu nhất.

## 2. Ưu điểm và nhược điểm mô hình MVC

Bây giờ mình liệt kê một số ưu điểm và nhược điểm của mô hình MVC nhé.

**Ưu điểm**:

* Hệ thống phân ra từng phần nên dễ dáng phát triển
* Chia thành nhiều modun nhỏ nên nhiều người có thể làm chung dự án
* Vấn đề bảo trì cũng tương đối ok, dễ nâng cấp
* Dễ dàng debug trong quá trình xây dựng

**Nhược điểm**:

* Hệ thống sẽ chạy chậm hơn PHP thuần, tuy nhiên nó ko phải là vấn đề :D
* Xây dựng cầu kì và mất thời gian để xây dựng thư viện, cấu trúc

Xét về ưu và nhược thì rõ ràng nên xử dụng MVC phải không nào các bạn :D

## 3. Luồng xử lý trong mô hình MVC

Ok bây giờ mình sẽ đưa ra một ví dụ về luồng xử lý trong mô hình MVC nhé.

Giả sử bạn đang xem một bài tuts trên website freetuts.net có URL là https://freetuts.net/rewrite-url-trong-codeigniter-344.html thì hệ thống MVC sẽ xử lý như sau:

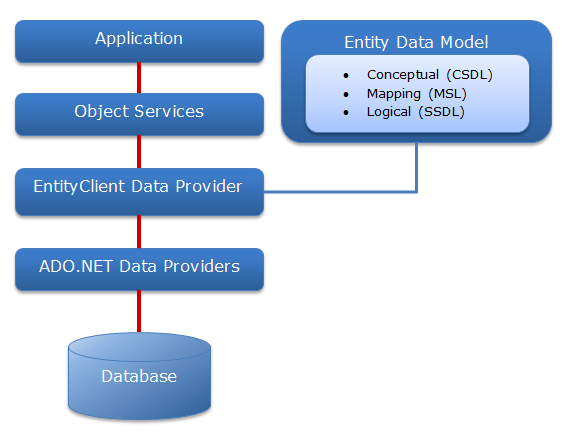
* **Bước 1**: Dựa vào yêu cầu của bạn là xem bài viết có id=344 nên controller sẽ gọi tới một hàm lấy dữ liệu theo id trong model
* **Bước 2**: Sau khi có dữ liệu controller sẽ gửi qua View, lúc này view có nhiệm vụ xử lý dữ liệu và convert thành nhữn đoạn mã HTML
* **Bước 3**: Sau khi view kết thúc thì controller sẽ gửi trả nội dung HTML của view về cho client nên bạn sẽ xem được nội dung của bài tus có id=344

<https://bachkhoa-aptech.edu.vn/huong-dan-su-dung-entity-framework-code-first-trong-asp-net-mvc.html>

# [Tổng quan kiến trúc của Entity Framework](https://yinyangit.wordpress.com/2011/11/24/tong-quan-ve-kien-truc-entity-framework/)

ADO.NET Entity Framework là một nền tảng được sử dụng để làm việc với database thông qua cơ chế ánh xạ Object/Relational Mapping (ORM). Nhờ đó, bạn có thể truy vấn, thao tác với database gián tiếp thông qua các đối tượng lập trình.

Kiến trúc của Entity Framework được minh họa như sau:

[](https://yinyangit.wordpress.com/2011/11/24/tong-quan-ve-kien-truc-entity-framework/entity-framework-architecture/)

Các nội dung dưới đây sẽ đi vào giải thích từng phần trong sơ đồ trên. Trong giới hạn của bài viết, tôi chỉ đi khái quát để bạn nắm được nội dung chính. Trong thực tế, bạn cũng không cần biết sâu hơn về kiến trúc bên dưới trừ những mục đích đặc biệt.

## Object Services

Đây là các class tự động sinh ra tương ứng với mô hình dữ liệu. Các class này bao gồm:

–      [ObjectContext](http://msdn.microsoft.com/en-us/library/system.data.objects.objectcontext%28v=vs.90%29.aspx) đại diện cho một database. ObjectContext có chức năng quản lý các kết nối, định nghĩa mô hình dữ liệu với metadata và thao tác với database. Lớp này cũng có thể thêm vào các phương thức đại diện cho các stored procedure trong database.

–      [ObjectSet<TEntity>](http://msdn.microsoft.com/en-us/library/dd412719.aspx) là một  một tập hợp các entity. Mỗi đối tượng này tương ứng với một table. Có thể lấy được các đối tượng này thông qua các property tương ứng của ObjectContext.

–      [EntityObject](http://msdn.microsoft.com/en-us/library/system.data.objects.dataclasses.entityobject%28v=vs.90%29.aspx), [ComplexObject](http://msdn.microsoft.com/en-us/library/system.data.objects.dataclasses.complexobject%28v=vs.90%29.aspx) là các lớp tương ứng cho một dòng dữ liệu của table trong database. Khác biệt chính giữa hai loại này là ComplexObject không chứa primary key.

–      [EntityCollection<TEntity>](http://msdn.microsoft.com/en-us/library/bb354106.aspx) và [EntityReference<TEntity>](http://msdn.microsoft.com/en-us/library/bb297956.aspx): là các đối tượng thể hiện mối quan hệ (relationship) giữa hai entity class. Mỗi đối tượng này có thể được truy xuất thông qua các property của entity class.

Ta có bảng ánh xạ tương đương sau giữa các đối tượng trong database và EF:

|  |  |
| --- | --- |
| **Database Object** | **Entity Framework Object** |
| Database | ObjectContext |
| Table, View | EntityObject, ComplexObject |
| Column | Property |
| Relationship | EntityCollection<TEntity>, EntityReference<TEntity> |

Ví dụ sau cho thấy cách tạo một đối tượng ObjectContext từ mô hình dữ liệu NorthwindEntities (tên của connection string được lưu trong app.config), sau đó sử dụng LINQ to Entities để lấy ra các sản phẩm (trong bảng Products) có tên bắt đầu bằng “G” và in ra màn:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | // …  ObjectContext context = new ObjectContext("name=NorthwindEntities");    ObjectSet<Product> products = context.CreateObjectSet<Product>();    var query = from p in products              where p.ProductName.StartsWith("g")              select p;    foreach (var item in query)  {      Console.WriteLine("{0,-4}{1}",item.ProductID, item.ProductName);  }  // … |

Output:

6 Grandma's Boysenberry Spread

15 Genen Shouyu

22 Gustaf's Knäckebröd

24 Guaraná Fantástica

26 Gumbär Gummibärchen

31 Gorgonzola Telino

33 Geitost

37 Gravad lax

44 Gula Malacca

56 Gnocchi di nonna Alice

69 Gudbrandsdalsost

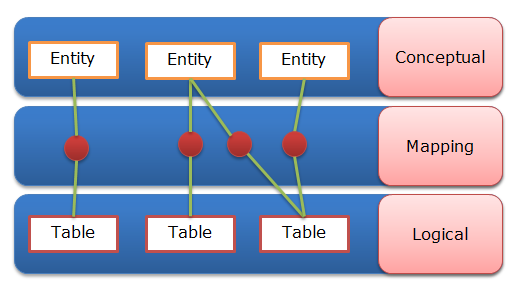
## Entity Data Model

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.

[](https://yinyangit.wordpress.com/2011/11/24/tong-quan-ve-kien-truc-entity-framework/entity-mapping-model/)

Tham khảo thêm: [CSDL, SSDL, and MSL Specifications](http://msdn.microsoft.com/en-us/library/bb399604.aspx)

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.

## EntityClient Data Provider

EntityClient là một data provider mới của ADO.NET dùng để truy xuất đến database. Được xây dựng bên trên các ADO.NET data provider cơ bản, EntityClient không truy xuất trực tiếp dữ liệu mà thông qua các data provider khác dựa vào các thông tin dữ liệu từ Entity Data Model.

EntityClient cũng bao gồm các lớp giống như các ADO.NET data provider khác và tên lớp được đặt với tiền tố Entity. Ví dụ bạn có thể tạo kết nối bằng [EntityConnection](http://msdn.microsoft.com/en-us/library/system.data.entityclient.entityconnection.aspx), tạo các câu truy vấn bằng [EntityCommand](http://msdn.microsoft.com/en-us/library/system.data.entityclient.entitycommand.aspx) và đọc kết quả bằng [EntityDataReader](http://msdn.microsoft.com/en-us/library/system.data.entityclient.entitydatareader.aspx).

Một điểm khác biệt với các data provider khác là EntityClient sử dụng Entity SQL để truy vấn dữ liệu. Các lệnh Entity SQL sẽ được chuyển thành một cấu trúc lệnh dạng cây (command tree) và chuyển xuống cho các data provider khác.

Ví dụ sau sử dụng EntityClient để tạo kết nối, tạo đối tượng EntityCommand và sử dụng tham số để lấy các sản phẩm có CategoryID là 1:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | int categoryID = 1;  using (EntityConnection con = new EntityConnection("Name=NorthwindEntities"))  {      con.Open();      EntityCommand cmd = con.CreateCommand();      cmd.CommandText = "SELECT VALUE p FROM NorthwindEntities.Products AS p WHERE p.CategoryID = @catId";      cmd.Parameters.AddWithValue("catId", categoryID);      EntityDataReader reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess);      while (reader.Read())      {          var name = reader["ProductName"];          var catId=reader["CategoryID"];          Console.WriteLine("{0,-3}{1}",catId,name);      }      reader.Close();  }  // ... |

Output:

CategoryID ProductName

1 Chai

1 Chang

1 Guaraná Fantástica

1 Sasquatch Ale

1 Steeleye Stout

1 Côte de Blaye

1 Chartreuse verte

1 Ipoh Coffee

1 Laughing Lumberjack Lager

1 Outback Lager

1 Rhönbräu Klosterbier

1 Lakkalikööri