[**Sử dụng Singleton Pattern trong C#**](http://www.yoda.arachsys.com/csharp/singleton.html)

Posted On Tuesday, May 27, 2008 at at [8:18 PM](http://nthoai.blogspot.com/2008/05/su-dung-singleton-trong-csharp.html) by Van Nguyen

- Trong loạt bài dịch về [Spring.NET](http://nthoai.blogspot.com/search/label/Spring.NET) tôi đã có cơ hội chia sẽ với các bạn một mẫu Design Pattern khá hấp dẫn là Singleton. Sau bài viết đó, tôi cũng muốn viết thêm một ít về mẫu thiết kế này. Thật may khi tôi vô tình tìm được một [bài viết rất hay](http://www.yoda.arachsys.com/csharp/singleton.html) về chủ đề này. Trong bài, tác giả đề cập đến khá nhiều vấn đề lí thú khác có liên quan đến lập trình, đặc biệt là tính [lazy-load](http://en.wikipedia.org/wiki/Lazy_initialization) và [thread-safe](http://en.wikipedia.org/wiki/Thread-safety) của đối tượng. Vì vậy, tôi quyết định nấu nướng nó, 1 là để chia sẻ với những ai quan tâm, 2 là để nâng cao khả năng đọc hiểu của tôi , và tất nhiên là khả năng viết lách nữa hê hê .  
  
- Trước khi đọc bài này, các bạn nên nắm trước và hiểu một số thuật ngữ, từ khóa và kĩ thuật trong C# như : *base class, sub class, constructor, member, seal, private, public, protected, static, instance, reference, property, performance, thread, multi-thread, lock, lazy-load*. Đa số từ khóa trên theo tôi nghĩ các bạn học lập trình đều đã biết đến, cá nhân tôi cũng phải tự tìm hiểu trong quá trình làm việc khi lần đầu tiên gặp những kĩ thuật như lock và lazy-load . Cho nên trước khi viết, tôi xin giới thiệu sơ qua hai kĩ thuật này. Ngoài ra, trong bài viết có khi tôi sử dụng chữ “**instance**” và “**đối tượng**”, hai từ này đều có ý nghĩa vai trò như nhau. Tôi nghĩ có một số từ tôi sẽ giữ nguyên tên tiếng Anh mà không dịch ra tiếng Việt như: thread, multi-thread, performance, …. vì khi được dịch ra đọc thấy nó rất chuối , hơn nữa những từ tiếng Anh này cũng khá quen thuộc với chúng ta, ai làm lập trình cũng có thể hiểu được. Và cuối cùng, trong bài dịch tôi có thêm vào một số ý kiến của mình, các bạn sẽ dễ dàng phân biệt các ý kiến của tôi với phần dịch khi thấy icon sau " "

**Singleton Pattern**

- Khi lập trình [multi-thread](http://en.wikipedia.org/wiki/Thread_%28computer_science%29), chắc chắn các bạn sẽ gặp phải vấn đề đau đầu gọi là **đồng bộ hóa tiến trình**. Lúc trước tôi học về vấn đề này trong trường đại học ở môn “Hệ điều hành nâng cao”. He he lúc trước lười lắm nên khi đi thi không được điểm cao, cả lớp toàn 9 với 10 riêng tôi được 7. Vì đua đòi với chúng bạn, tôi quyết định thi lại môn này với hi vọng nâng điểm lên. Xui thay đến lần thi thứ 2 tôi bị bệnh khá nặng, thế là phải bỏ thi. Đến học kì sau thi lại thì chỉ được 5 điểm, chả hiểu. Thi lần nữa để hi vọng nâng điểm thì lại được 7. Trời ko thương người hiền lành mà, nhưng cũng nhờ vậy mà tôi nắm được khá rõ kĩ thuật lập trình thread, lỡ đứa nào có hỏi cũng hãnh diện nói :”Tao đã từng lập trình MultiThread, biết viết chương trình chat qua mạng nhé”. Viết xàm hơi nhiều rồi, giờ viết thiệt : Lock là một kĩ thuật trong lập trình multi-thread, gọi là “Đồng bộ hóa tiến trình”, hiểu đơn giản là kĩ thuật đó giúp *hai hoặc nhiều thread* trong chương trình có thể làm việc nhịp nhàng với nhau. Giả sử trong chương trình của bạn có một biến đếm *toàn cục*, nhiều thread trong chương trình sẽ thay phiên nhau truy xuất giá trị biến đó và tăng giá trị nó lên 1 đơn vị. Nếu có 2 thread cùng truy xuất giá trị một lúc và cùng tăng điểm đồng thời thì sẽ có trường hợp biến đếm chỉ tăng được 1 đơn vị, vì lần tăng của thread này sẽ đè giá trị mà thread kia vừa cập nhật cho biến đếm. Do đó đám đồ đệ của anh Bill Gate mới nghĩ ra cách để giải quyết vấn đề trên cho trong .NET, và Lock chỉ là một trong rất nhiều kĩ thuật giúp ta đồng bộ hóa tiến trình. Tuy nhiên trong phạm vi bài này chúng ta chỉ nói chính về mấu thiết kế Singleton nên tôi hi vọng với những giải thích ngắn gọn trên, các bạn đã có được khái niệm cơ bản về lock, hiểu nó là gì và nó có thể làm gì cho chúng ta.  
  
- Kế tiếp, xin nói sơ qua về [lazy-load](http://en.wikipedia.org/wiki/Lazy_initialization) hay [lazy initialization](http://en.wikipedia.org/wiki/Lazy_initialization). Đó là một kĩ thuật giúp chúng ta delay lại qua trình tạo ra đối tượng, quá trình tính toán, … và chỉ thực thi quá trình đó đến khi nào nó thực sự cần đến. Về nguyên tắc, kĩ thuật này có thể đạt được bằng cách sử dụng một cờ để kiểm tra xem quá trình của chúng ta có diễn ra hay chưa, nếu chưa thì sẽ thực hiện còn nếu đã làm rồi thì chỉ cần trả về giá trị được lưu của lần thực thi trước đó hoặc đơn giản chẳng làm gì cả. Trong thực tế thì kĩ thuật này sẽ được áp dụng nhiều trong các framework [ORM](http://madgeek.com/Articles/ORMapping/EN/mapping.htm). Tôi xin nói một ví dụ: giả sử bạn có 1 database với 2 bảng lưu thông tin lớp học và học sinh với ràng buộc một lớp học có thể có nhiều học sinh. Tương ứng với 2 bảng trong cơ sở dữ liệu thì trong chương trình của bạn có 2 lớp là **Class** và **Student**. Trong lớp Class sẽ có một member **StudentList** thuộc kiểu **List**. Như vậy khi chương trình của bạn đọc dữ liệu từ database lên các bạn sẽ có một đối tượng \_class thuộc kiểu **Class** và danh sách các **Student** tương ứng. Thực sự lúc này đối tượng \_class này chỉ giữ *số lượng student* thôi chứ thực sự \_class.StudentList[i] vẫn chưa có giá trị. Đến khi nào bạn cần truy xuất thông tin một student nào đó trong mảng thì nó mới thực sự load dữ liệu tương ứng từ database. Nếu các bạn muốn tìm hiểu thêm hãy google với từ khóa *lazy-load* sẽ ra rất nhiều bài viết hay. Hiện nay, các framework ORM đều hỗ trợ tính năng này và các bạn chỉ việc download về , đọc manual và sử dụng cho đúng cách. Đến đây hi vọng các bạn đã nắm được lazy-load là gì và một số ứng dụng trong thực tế của lazy-load. Tôi sẽ dừng việc giới thiệu ở đây và sẽ trình bày vào vấn đề chính của chúng ta: **Sử dụng Singleton trong C#**   
  
- Mẫu thiết kế [Singleton](http://nthoai.blogspot.com/search/label/Singleton) là một trong những mẫu thông dụng và được sử dụng rộng rãi nhất trong kĩ thuật lập trình. Về nguyên tắc, một Singleton là một lớp chỉ cho phép một đối tượng tương ứng được tạo, và tất nhiên nó cũng phải có một public member để những đối tượng khác có thể truy xuất *instance Singleton* này. Thông thường, cách thiết kế lớp theo kiểu Singleton sẽ không sử dụng hàm khởi tạo có tham số, nếu không thì một yêu cầu khởi tạo đối tượng với các tham số khác nhau sẽ gây rắc rối. (Nếu cùng một instance được gọi bằng hàm getInstance với những tham số khác nhau thì ta dùng mẫu thiết kế [Factory Method](http://nthoai.blogspot.com/search/label/Factory%20Method) sẽ hợp lý hơn) Bài viết này chỉ đề cập đến những Singletons với hàm khởi tạo không tham số, hay hàm *getInstance()* không có tham số. Nói chung, khi người ta sử dụng Singleton vì tính lazy-load của nó, instance của lớp sẽ không cần được tạo đến khi nào nó thật sự cần thiết.  
  
- Có khá nhiều cách để thiết kế Singleton trong C#. Tác giả sẽ thể hiện từng cách từ đơn giản đến phức tạp, bắt đầu bằng cách thông dụng nhất- cách không có tính thread-safe (an toàn khi sử dụng với thread) đến những cách có tính thread-safe và hỗ trợ lazy-loaded; đơn giản cũng như performance cao. Trong những ví dụ code mẫu ở trong bài viết, tác giả sẽ sử dụng từ khóa private mặc định cho các member của lớp. Trong nhiều ngôn ngữ khác như Java, có thể sẽ khác đi đôi chút tuy nhiên từ khóa private vẫn nên được sử dụng.  
  
- Dù có nhiều cách thực hiện, tất cả những phương pháp tôi giới thiệu ở trong bài viết này đều có thể sử dụng một trong những điều sau để thiết kế Singleton:  
  
• Có sử dụng một hàm khởi tạo không tham số và được khai báo private, làm như vậy sẽ không cho những lớp khác khởi tạo nó. Ngoài ra, khai báo hàm khởi tạo private cũng không cho những lớp con kế thừa, vì nếu một lớp được thiết kế theo kiểu Singleton thì nó có thể có hai hoặc nhiều lớp con và nếu mỗi lớp con đều có thể tạo ra instance thì mẫu thiết kế Singleton sẽ không còn đúng. Tôi chưa hiểu rõ lắm ý tác giả là thế nào. Có thể nếu một lớp Singleton có nhiều lớp con thì trong một lúc hệ thống có thể có nhiều hơn 1 đối tượng cùng thuộc về base class là Singleton, tất nhiên trong số những đối tượng đó sẽ có những instance là kiểu của subclass và chỉ một instance duy là kiểu base class thôi (Tính đa hình - polimorphism của OOP). Dù sao đi nữa, tôi vẫn thích cách khai báo protected hơn. Nếu chúng ta dùng đúng cách thì không có gì phải lo ngại, cũng giống như điện vậy thôi, có thể gây hỏa hoạn hoặc chết người nhưng nhà nhà đều dùng điện, người người dùng điện .  
  
• Có sử dụng lớp sealed (một lớp được khai báo với từ khóa seal sẽ không cho phép tạo lớp con kế thừa nó). Điều này đôi khi không cần thiết theo như cách làm tác giả nói ở trên nhưng nó có thể giúp trình dịch [JIT (Just – In – Time)](http://en.wikipedia.org/wiki/Just-in-time_compilation) tối ưu hóa chương trình.  
  
• Có thể sử dụng một biến static để giữ instance duy nhất của lớp.  
  
• Có thể sử dụng một biến public static để giúp các lớp khác sử dụng biến *instance Singleton* duy nhất được nói ở cách 3.  
  
- Xin nhắc lại một lần nữa là trong tất cả các cách làm trên đều có sử dụng một public static property tên là **Instance** ( *đây chỉ là cách đặt tên cho property, bạn có thể đặt tên nào tùy thích nhưng tên* ***Instance*** *có vẽ như ngắn gọn và hay nhất*) với mục đích giúp chúng ta sử dụng instance duy nhất của lớp. Tuy nhiên nếu thích sử dụng method theo kiểu getInstance() thì vẫn có thể chuyển qua cách đấy dễ dàng mà không ảnh hưởng đến tính *thread-safety* hay *performance*.

**First version – not thread-safe.**

**public sealed class Singleton1**

**{**

**static Singleton1 instance = null;**

**Singleton1()**

**{**

**}**

**public static Singleton1 Instance**

**{**

**get**

**{**

**if (instance == null)**

**{**

**instance = new Singleton1();**

**}**

**return instance;**

**}**

**}**

**}**  
- Như đã giới thiệu trước đây, cách làm trên là không đạt được nguyên tắc thread-safe. Hai thread khác nhau có thể cùng một lúc chạy đến đoạn code trên, kiểm tra xem biến instance có bằng null hay không và thấy là nó null thật nên cùng tạo ra instance trong mỗi thread, như vậy là không đúng với ý tưởng thiết kế của Singleton. Chúng ta cũng dễ dàng thấy rằng biến instance trong đoạn code trên cũng có thể được khởi tạo trước khi biểu thức kiểm tra null được tính, nhưng vì các thread khác nhau đều có thể đọc bộ nhớ cùng một lúc, nếu không có một cơ chế kiểm soát để các thread khi truy xuất vùng nhớ tuần tự thì sẽ có trường hợp 2 hoặc nhiều thread đều kiểm tra và thấy biến instance đang là null và tạo ra hàng loạt các instance mới. Thiên hạ đại loạn.

**Second version – simple thread safety**

**public sealed class Singleton2**

**{**

**static Singleton2 instance = null;**

**static readonly object padlock = new object();**

**Singleton2()**

**{**

**}**

**public static Singleton2 Instance**

**{**

**get**

**{**

**lock (padlock)**

**{**

**if (instance == null)**

**{**

**instance = new Singleton2();**

**}**

**return instance;**

**}**

**}**

**}**

**}**  
  
- Cách làm thứ hai này là hoàn toàn thread-safe. Nếu bạn đã đọc phần giới thiệu ở trên của tôi về lock , bạn sẽ thấy trong đoạn code có sử dụng một shared object để lock phần code kiểm tra null của biến instance. Cách làm này sẽ giải quyết được sự xung đột có thể có giữa các thread mà chúng ta đã đề cập ở cách một. Với kĩ thuật này chúng ta sẽ chắc chắn chỉ có duy nhất một thread được phép tạo ra instance, duy nhất một thread được xử lý đoạn code trong phần lock ở một thời điểm, sau đó các thread khác mới được nhảy vào và thấy rằng đã tồn tại một instance khác null được khởi tạo trong thread đầu tiên. Tuy nhiên, cách làm này lại không có lợi về performance cho lắm vì mỗi lần biến Instance được sử dụng thì hàm lock sẽ được gọi và sẽ làm chương trình chạy chậm đi một ít.  
  
- Điểm thứ hai cần lưu ý là ta có thể sử dụng bất kì object nào để dùng cho kĩ thuật lock, có thể sử dụng ngay biến Singleton mà một số lập trình viên vẫn sử dụng. Tuy nhiên tác giả sử dụng một 1 biến private static khác để dùng gọi trong lock. Locking trên những đối tượng mà các lớp khác trong chương trình có thể thấy và sử dụng được là điều vô cùng nguy hiểm, thậm chí có thể gây ra hiện tượng **dead-lock**. Sử dụng một shared object là cách tôi thích dùng bất khi nào có thể và tạo ra những đối tượng chỉ để dùng nó cho locking. Thường thì những đối tượng này nên được khai báo private trong lớp mà nó được sử dụng để tránh vấn đề nói trên. Cách làm đơn giản này giúp chúng ta có thể viết những chương trình thread-safe dễ dàng hơn nhiều.

**Third version – attempted thread-safety using double-check locking**

**public sealed class Singleton3**

**{**

**static Singleton3 instance = null;**

**static readonly object padlock = new object();**

**Singleton3()**

**{**

**}**

**public static Singleton3 Instance**

**{**

**get**

**{**

**if (instance == null)**

**{**

**lock (padlock)**

**{**

**if (instance == null)**

**{**

**instance = new Singleton3();**

**}**

**}**

**}**

**return instance;**

**}**

**}**

**}**  
  
- Phương pháp trên sẽ bảo đảm tính thread-safe của chương trình mà không cần phải gọi hàm lock mỗi lần sử dụng biến Instance. Nhưng thật không may, làm như vậy sẽ dẫn đến 4 trở ngại sau:  
  
• Không áp dụng được cho Java. Điều này dường như hơi thừa khi bàn tới vì chúng ta đang đề cập đến Singleton cho .NET, tuy nhiên biết đâu sau này chúng ta cần áp dụng cho Java, lập trình viên C# cũng có thể trở thành lập trình viên Java cơ mà. Mô hình bộ nhớ trong Java không bảo đảm hàm khởi tạo được kết thúc trước khi địa chỉ vùng nhớ của đối tượng được gán cho biến instance. Mô hình vùng nhớ của Java đã được cải tiến ở version 1.5 (hiện tại có sự nâng cấp nào chưa thì các bạn cần kiểm tra lại), nhưng cách làm double-check locking vẫn chưa thể áp dụng được cho Java như C#.  
  
• Cách làm này tuy an toàn (trên lý thuyết) với .NET framework 2.0 nhưng tôi vẫn không thích những gì trên lý thuyết, đặc biệt khi có sự nghi ngờ về tính an toàn của hệ thống.  
  
• Ngoài ra cách này rất dễ sai, cách thiết kế trên buộc bạn phải theo chính xác từng bước như code mẫu.  
  
• Và cuối cùng cách này vẫn chưa phải là cách hiệu quả nhất.  
  
  
**Fourth version – not quite as lazy, but thread-safe without using locks**

**public sealed class Singleton4**

**{**

**static readonly Singleton4 instance = new Singleton4();**

**//Explicit static constructor to tell C# compiler**

**// not to mark type as beforefieldinit**

**static Singleton4()**

**{**

**}**

**private Singleton4()**

**{**

**}**

**public static Singleton4 Instance**

**{**

**get**

**{**

**return instance;**

**}**

**}**

**}**  
  
- Các bạn có thể dễ dàng thấy rằng cách làm trên đơn giản hơn 3 cách trước rất nhiều, nhưng liệu cách này có bảo đảm tính thread-safe và lazy-load như mong muốn? Tất nhiên, hàm khởi tạo static của 1 lớp C# được thiết kế để chạy khi bất kì instance nào của lớp được tạo hoặc là có một hàm static nào đó của lớp được gọi. Ngoài ra hàm khởi tạo static chỉ được gọi một lần duy nhất trong mỗi [AppDomain](http://www.odetocode.com/Articles/305.aspx). Tính chất này của C# sẽ giúp chúng ta kiểm tra xem một type có được khởi tạo chưa và nó sẽ tự động gọi hàm khởi tạo static bất cứ khi nào cần thiết, như vậy sẽ nhanh hơn là dùng những bước kiểm tra như những phương pháp đầu. Tuy nhiên, cách này vẫn vướng phải một số khiếm khuyết sau đây:  
  
• Cách làm này không bảo đảm tính lazy-load. Trong trường hợp đặc biệt, nếu bạn dùng một static member thay vì dùng biến public static property Instance, lần gọi đầu tiên đến những static member này sẽ gọi hàm khởi tạo static và tạo ra instance của lớp. Nhược điểm này sẽ được khắc phục ở cách kế tiếp.  
  
• Các hàm khởi tạo static nếu có gọi lẫn nhau sẽ bị lặp trong một vòng lặp vô tận. Thực sự thì tôi vẫn chưa thể reproduce được vấn đề này nên cũng không biết nhận định của tác giả là đúng hay sai, tuy nhiên tôi nghĩ chẳng ai lại đi thiết kế những constructor đệ quy gọi vòng vòng lẫn nhau như thế cả.  
  
• Tính lazy-load trong quá trình khởi tạo đối tượng chỉ được bảo đảm trong môi trường .NET khi lớp không bị xem là “[beforefieldinit](http://www.yoda.arachsys.com/csharp/beforefieldinit.html)”. Thật không may khi trong bộ biên dịch của C#, nó sẽ xem mọi lớp không có static constructor là beforefieldinit. Tác giả có một bài viết về điều này ở đây dành cho ai muốn tìm hiểu về chủ đề này.  
  
- Một điểm cần lưu ý nữa với cách làm này (chỉ có cách này) là khai báo biến Instance là public static readonly. Như vậy code của chúng ta sẽ đơn giản và gọn hơn. Tuy nhiên, rất nhiều người vẫn thích sử dụng cách Property vì có thể chúng ta sẽ cần tới Property trong một số trường hợp.  
  
 **Fifth version – fully lazy instantiation**

public sealed class Singleton  
{

Singleton()  
{  
}  
  
public static Singleton Instance  
{

get  
{

return Nested.instance;

}

}  
  
class Nested  
{

// Explicit static constructor to tell C# compiler  
// not to mark type as beforefieldinit  
static Nested()  
{  
}  
internal static readonly Singleton instance = new Singleton();

}

}

- Trong cách này, quá trình khởi tạo đối tượng được gọi ở lần truy xuất đầu tiên đến static member của lớp con, quá trình này chỉ xảy ra trong biến **Instance**. Như vậy cách làm này là hoàn toàn có tính lazy-load và đồng thời vẫn đảm bảo performance tốt như ở cách 4. Bạn cũng nên chú ý rằng mặc dù lớp con (lớp ***Nested***) sẽ không thể được truy xuất từ bên ngoài lớp Singleton của chúng ta. Kĩ thuật này sẽ không gây bất kì vấn đề gì ngoài việc đoạn code của chúng ta sẽ phức tạp hơn một chút để bảo đảm tính lazy-load. Đối với cá nhân tôi hơi bất ngờ khi xem đến cách thứ năm này, có một chút gì đó nghệ thuật trong lập trình phải không các bạn và điều đó thể hiện rõ ở đây.  
  
**Performance vs laziness**  
  
- Trong nhiều trường hợp, bạn không thật sự cần đến lazy-load trừ khi quá trình khởi tạo của lớp có làm nhiều việc gì đó mất nhiều thời gian. Nếu không đòi hỏi những yêu cầu phức tạp, chúng ta thấy rằng cách sử dụng hàm khởi tạo static sẽ làm ứng dụng chạy nhanh hơn. Nó sẽ tăng performance cho ứng dụng vì cách làm đó cho phép trình biên dịch JIT chỉ kiểm tra một lần duy nhất để xem instance của lớp có được tạo ra hay chưa. Nếu instance Singleton của bạn được truy xuất bên trong một vòng lặp lớn, bạn có thể sẽ thấy chậm hơn đôi chút, tuy nhiên chúng ta nên cân nhắc khi nào cần lazy-load để áp dụng cách thích hợp nhất. Theo tôi không có cách nào tuyệt hảo nhất, chúng ta nên biết nhiều cách để áp dụng đúng nơi đúng chỗ.  
 **Exception**- Thỉnh thoảng, một số xử lý của bạn trong hàm khởi tạo của Singleton có thể tung ra exception, và có thể làm ảnh hưởng đến toàn bộ chương trình. Rất có thể trong trường hợp gặp exception khi khởi tạo, ứng dụng của bạn sẽ thử khởi tạo đối tượng một lần nữa trong phần catch exception. Và một trong những cách có thể làm điều đó là nhờ vào kĩ thuật gọi là “[type initializers](http://www.west-wind.com/weblog/posts/187116.aspx)”, tuy nhiên nó vẫn có thể gây nhiều vấn đề khác. Những lần chạy chương trình khác nhau có thể sẽ xử lý vấn đề này theo các cách khác nhau, tuy nhiên tôi vẫn không biết lần nào chạy đúng như yêu cầu. Và cho dù được như vậy, code của bạn vẫn có thể bị lỗi ở lần chạy khác. Để tránh chuyện này, tôi đề nghị chúng ta sử dụng cách thứ hai.  
  
**A word on performance**- Nhiều người cho rằng sử dụng locking tốn rất nhiều resource và làm cho ứng dụng chạy chậm đi. Tác giả đã thực hiện một chương trình để test thử bằng cách truy xuất biến instance của Singleton trong vòng lặp 1 tỉ lần. Tuy không khoa học lắm nhưng trong thực tế bạn có thể muốn kiểm tra xem nó nhanh chậm thế nào. Kết quả cho thấy thực sự không quan trọng. Và kết quả test với phương pháp chậm nhất (cách 2) là mất 40s để hoàn thành xong vòng lặp 1 tỉ lần. Điều này có nghĩa là bạn có thể truy xuất 25 triệu lần biến instance trong một giây, cho nên những cố gắng nâng cao hiệu xuất ứng dụng bằng cách tối ưu hóa mẫu Singleton là không đáng kể trong một số trường hợp.  
  
- Theo ý kiến của Nguyễn Thoại thì cách dùng nào cũng được hết và tôi vẫn sử dụng thường xuyên cách đầu tiên. Bời vì tôi chắc rằng chương trình của tôi không sử dụng multi-thread nên chẳng có thread nào xung đột với nhau. Ngoài ra performance của ứng dụng còn phụ thuộc vào nhiều yếu tố khác nữa, chúng ta nên tìm hiểu xem nguyên nhân làm chậm ứng dụng và fix đúng chỗ. Dù sao đi nữa thì việc sử dụng cách nào là tùy ý thích của chúng ta, như tôi đã nói tôi ưu hóa chương trình khi có thể thì không lúc nào là thừa cả.  
  
**Tôi thường sử dụng Singleton ở đâu?**  
- Trong gần 3 năm làm lập trình, tôi rất thích sử dụng Singleton cho những lớp chỉ mang tính chất Utility, những lớp mà nhiều nơi có thể gọi nó. Một số lần tôi sử dụng Singleton cho một số form trong các ứng dụng desktop. Trong windows, nếu để ý các bạn sẽ thấy có một số cứa sổ chỉ hiện một lần cho dù bạn click vào shotcut của nó như thế nào. Điển hình là cửa sổ Find của notepad hay cửa sổ Run của Windowz. Form cũng là một lớp trong nhiều ngôn ngữ lập trình nên áp dụng Singleton cho nó là điều hoàn toàn bình thường, nếu bạn muốn một form chỉ xuất hiện một lần cho dù bạn gọi nó bằng cách click cả chục cái thì đấy là lúc bạn có thể áp dụng Singleton.