Điểm dừng tiếp theo của chúng tôi là Mô hình Singleton, tấm vé của chúng tôi để tạo ra các đối tượng độc nhất vô nhị mà chỉ có một thể hiện duy nhất. Trên thực tế, đối với nhiều loại đối tượng này, nếu chúng ta khởi tạo nhiều đối tượng, chúng ta sẽ gặp đủ loại vấn đề như hành vi chương trình không chính xác, sử dụng quá mức tài nguyên hoặc kết quả không nhất quán. Bạn biết đấy, giống như trong Java, tôi có thể làm điều đó với một biến tĩnh. Singleton Pattern cũng cung cấp cho chúng ta một điểm truy cập toàn cục, giống như một biến toàn cục, nhưng không có nhược điểm.  
  
được tạo khi ứng dụng của bạn bắt đầu. Như bạn sẽ thấy, với Singleton Pattern, chúng ta chỉ có thể tạo các đối tượng của mình khi chúng cần thiết. và các phương thức cũng như công cụ sửa đổi quyền truy cập, nó không phải. Tuy nhiên, trong cả hai trường hợp, thật thú vị khi thấy cách hoạt động của Singleton, và nghe đơn giản như vậy, mã Singleton rất khó để làm đúng.  
  
Tại sao bạn không bắt đầu bằng cách cho chúng tôi biết một chút về bản thân bạn. Tôi dựa trên Mô hình Singleton, đảm bảo rằng tại bất kỳ thời điểm nào chỉ có một trường hợp duy nhất của tôi. Giả sử bạn có một đối tượng chứa cài đặt đăng ký. Bạn không muốn nhiều bản sao của đối tượng đó và các giá trị của nó chạy xung quanh - điều đó sẽ dẫn đến hỗn loạn.  
  
Bằng cách sử dụng một đối tượng như tôi, bạn có thể đảm bảo rằng mọi đối tượng trong ứng dụng của bạn đang sử dụng cùng một tài nguyên chung. Tôi thường được sử dụng để quản lý nhóm tài nguyên, như nhóm kết nối hoặc nhóm luồng. Sự thật mà nói ... à, điều này là mang tính cá nhân nhưng ... tôi không có công cụ xây dựng công cộng. Trên thực tế, tôi có thể đã giúp đỡ các đối tượng khác khi bạn yêu cầu.  
  
Không có bất ngờ lớn ở đó. Chúng tôi cũng đang ngăn không cho bất kỳ lớp nào khác tự tạo một phiên bản mới. Như bạn đã thấy, chúng ta có thể triển khai điều này để Singleton được tạo theo cách lười biếng, điều này đặc biệt quan trọng đối với các đối tượng sử dụng nhiều tài nguyên. Phương thức getInstance là tĩnh, có nghĩa là nó là một phương thức lớp, vì vậy bạn có thể truy cập phương thức này một cách thuận tiện từ bất kỳ đâu trong mã của mình bằng cách sử dụng Singleton.getInstance .  
  
Nói cách khác, khi chúng tôi đã đặt biến uniqueInstance thành một phiên bản của Singleton, chúng tôi không cần phải đồng bộ hóa phương thức này nữa. Sau lần đầu tiên, đồng bộ hóa hoàn toàn không cần thiết! Chúng ta có thể cải thiện đa luồng không? Đối với hầu hết các ứng dụng Java, rõ ràng chúng ta cần đảm bảo rằng Singleton hoạt động khi có nhiều luồng. Chà, chúng ta có một vài lựa chọn ... 1. Chỉ cần lưu ý rằng việc đồng bộ hóa một phương pháp có thể làm giảm hiệu suất xuống hệ số 100, vì vậy nếu một phần lưu lượng truy cập cao trong mã của bạn bắt đầu sử dụng getInstance , bạn có thể phải xem xét lại.  
  
Sử dụng «khóa kiểm tra hai lần» để giảm việc sử dụng đồng bộ hóa trong getInstance Với khóa kiểm tra hai lần, trước tiên chúng tôi kiểm tra xem một phiên bản có được tạo hay không, và nếu không, THÌ chúng tôi sẽ đồng bộ hóa. Bằng cách này, chúng tôi chỉ đồng bộ hóa lần đầu tiên thông qua những gì chúng tôi muốn. Tuy nhiên, do cách khởi tạo tĩnh được xử lý trong Java, điều này có thể rất lộn xộn, đặc biệt nếu có nhiều lớp tham gia. Trừ khi có nhu cầu bắt buộc phải triển khai «singleton» của bạn theo cách này, tốt hơn hết là bạn nên ở trong thế giới đối tượng.  
  
Tôi nghe nói rằng có khả năng hai trình nạp lớp có thể kết thúc với phiên bản Singleton của riêng chúng. Bây giờ, nếu lớp đó là một Singleton, thì vì chúng ta có nhiều hơn một phiên bản của lớp, chúng ta cũng có nhiều hơn một trường hợp của Singleton. Vì vậy, nếu bạn đang sử dụng nhiều bộ nạp lớp và Singleton, Trước Java 1.2, một lỗi trong trình thu gom rác đã cho phép các Singleton bị thu thập sớm nếu không có tham chiếu toàn cầu về chúng. Nói cách khác, bạn có thể tạo một Singleton và nếu tham chiếu duy nhất đến Singleton nằm trong chính Singleton, nó sẽ được thu thập và phá hủy bởi bộ thu gom rác.  
  
Điều này dẫn đến các lỗi khó hiểu vì sau khi Singleton được «thu thập», lệnh gọi tiếp theo tới getInstance đã tạo ra một Singleton mới sáng bóng. Trong nhiều ứng dụng, điều này có thể gây ra hành vi khó hiểu vì trạng thái được đặt lại một cách bí ẩn về giá trị ban đầu hoặc những thứ như kết nối mạng được đặt lại. Nếu vì một lý do nào đó, bạn vẫn đang sử dụng JVM trước Java 1.2, thì hãy lưu ý vấn đề này, nếu không, bạn có thể ngủ ngon khi biết các Singleton của bạn sẽ không bị thu thập sớm. Ngoài ra, nhiều nhà phát triển đã quen thuộc với mẫu Singleton vì nó đang được sử dụng rộng rãi.  
  
Điều đó nói rằng, một số nhà phát triển cảm thấy cần phải trừu tượng hóa chức năng của Singleton. Bạn không thể mở rộng một lớp với một hàm tạo riêng. Nhưng sau đó, nó không thực sự là Singleton nữa, vì các lớp khác có thể khởi tạo nó. Việc triển khai Singleton dựa trên một biến tĩnh, vì vậy nếu bạn thực hiện một lớp con đơn giản, tất cả các lớp dẫn xuất của bạn sẽ chia sẻ cùng một biến thể hiện.  
  
Đây có lẽ không phải là những gì bạn nghĩ đến. Vì vậy, để lớp con hoạt động, việc triển khai đăng ký các loại là bắt buộc trong lớp cơ sở. Trước khi thực hiện một sơ đồ như vậy, bạn nên tự hỏi bản thân rằng bạn thực sự thu được gì từ phân lớp con của một Singleton. Giống như hầu hết các mẫu khác, Singleton không nhất thiết phải là một giải pháp có thể phù hợp với một thư viện.  
  
Ngoài ra, mã Singleton rất nhỏ để thêm vào bất kỳ lớp nào hiện có. Cuối cùng, nếu bạn đang sử dụng một số lượng lớn các Singleton trong ứng dụng của mình, bạn nên xem xét kỹ thiết kế của mình. Singleton có nghĩa là để được sử dụng một cách tiết kiệm. Có một số nhược điểm khi sử dụng các biến toàn cục theo cách này.  
  
Một biến toàn cục có thể cung cấp biến sau, nhưng không cung cấp biến trước. Các biến toàn cục cũng có xu hướng khuyến khích các nhà phát triển gây ô nhiễm không gian tên với nhiều tham chiếu toàn cục đến các đối tượng nhỏ. Singletons không khuyến khích điều này theo cách tương tự, nhưng dù sao thì cũng có thể bị lạm dụng.