[Python Design Patterns 2](#_Toc104104179)

[1. Định nghĩa Design pattern 2](#_Toc1950514466)

[2. Tại sao cần sử dụng Design pattern 2](#_Toc1828715026)

[3. Phân loại Design Patterns 2](#_Toc20196937)

[3.1. Nhóm Creational (nhóm khởi tạo) 5](#_Toc1367789887)

[3.2. Nhóm Structural (nhóm cấu trúc) 6](#_Toc1269150494)

[3.3. Nhóm Behavioral (nhóm hành vi/ tương tác) 7](#_Toc233860252)

[4. Singleton Pattern 9](#_Toc693079683)

[4.1. Eager initialization 10](#_Toc1094031999)

[4.2. Static block initialization 10](#_Toc1206189951)

[4.3. Lazy Initialization 11](#_Toc319235819)

[4.4. Thread Safe Singleton 11](#_Toc166143782)

[4.5. Double Check Locking Singleton 12](#_Toc785341141)

[4.6. Bill Pugh Singleton Implementation 13](#_Toc1371489602)

[4.7. Phá vỡ cấu trúc Singleton Pattern bằng Reflection. 13](#_Toc1466142042)

[4.8. Enum Singleton 14](#_Toc1919949547)

[4.9. Serialization and Singleton 15](#_Toc1855900742)

[5. Factory Pattern 17](#_Toc1037700244)

[6. Tham khảo 17](#_Toc555866580)

**Python Design Patterns**

# Định nghĩa Design pattern

* Design Patterns (mẫu thiết kế) là một kỹ thuật trong lập trình hướng đối tượng. Nó cung cấp các “mẫu thiết kế”, giải pháp để giải quyết các vấn đề chung, thường gặp trong lập trình. Các vấn đề mà bạn gặp phải có thể bạn sẽ tự nghĩ ra cách giải quyết nhưng có thể nó chưa phải là tối ưu. Design Pattern giúp bạn giải quyết vấn đề một cách tối ưu nhất, cung cấp cho bạn các giải pháp trong lập trình OOP.
* Design Patterns không phải là ngôn ngữ cụ thể nào cả. Nó có thể thực hiện được ở phần lớn các ngôn ngữ lập trình, chẳng hạn như Java, C#, thậm chí là Javascript hay bất kỳ ngôn ngữ lập trình nào khác.

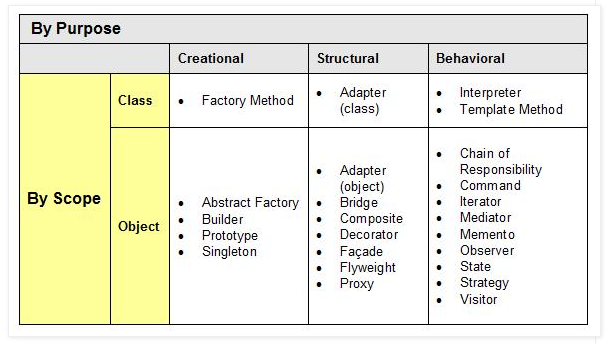
# Tại sao cần sử dụng Design pattern

* Design Pattern giúp bạn tái sử dụng mã lệnh và dẽ dàng mở rộng.
* Nó là tập hơn những giải pháp đã được tối ưu hóa, đã được kiểm chứng để giải quyết các vấn đề trong software engineering. Vậy khi bạn gặp bất kỳ khó khăn gì, design patterns là kim chỉ nam giúp bạn giải quyết vấn đề thay vì tự tìm kiếm giải pháp cho một vấn đề đã được chứng minh.
* Design pattern cung cấp giải pháp ở dạng tổng quát, giúp tăng tốc độ phát triển phần mềm bằng cách đưa ra các mô hình test, mô hình phát triển đã qua kiểm nghiệm.
* Dùng lại các design pattern giúp tránh được các vấn đề tiềm ẩn có thể gây ra những lỗi lớn, dễ dàng nâng cấp, bảo trì về sau.
* Giúp cho các lập trình viên có thể hiểu code của người khác 1 cách nhanh chóng (có thể hiểu là tính communicate). Mọi thành viên trong team có thể dễ dàng trao đổi với nhau để cùng xây dựng dự án mà không mất quá nhiều thời gian.

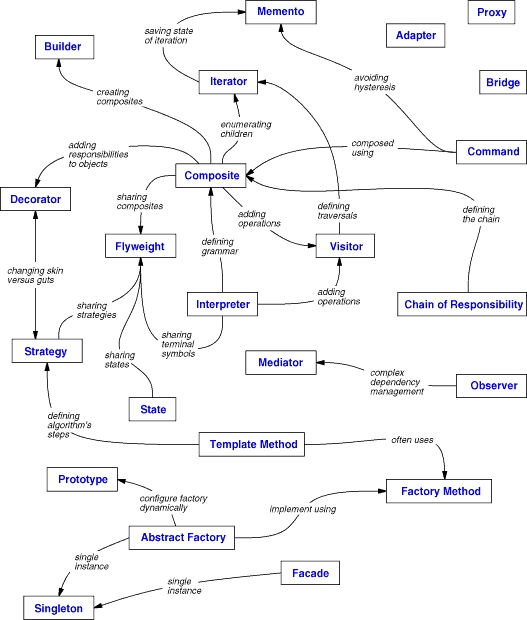
# Phân loại Design Patterns

Năm 1994, bốn tác giả **Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides** đã cho xuất bản một cuốn sách với tiêu đề **Design Patterns – Elements of Reusable Object-Oriented Software**, đây là khởi nguồn của khái niệm design pattern trong lập trình phần mềm.

Hệ thống các mẫu Design pattern hiện có **23** **mẫu** được định nghĩa trong cuốn “**Design patterns Elements of Reusable Object Oriented Software**” và được chia thành **3 nhóm**:



* **Creational Pattern** (nhóm khởi tạo – 5 mẫu) gồm: Factory Method, Abstract Factory, Builder, Prototype, Singleton. Những Design pattern loại này cung cấp một giải pháp để tạo ra các object và che giấu được logic của việc tạo ra nó, thay vì tạo ra object một cách trực tiếp bằng cách sử dụng method **new**. Điều này giúp cho chương trình trở nên mềm dẻo hơn trong việc quyết định object nào cần được tạo ra trong những tình huống được đưa ra.
* **Structural Pattern** (nhóm cấu trúc – 7 mẫu) gồm: Adapter, Bridge, Composite, Decorator, Facade, Flyweight và Proxy. Những Design pattern loại này liên quan tới **class** và các thành phần của **object**. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.
* **Behavioral Pattern** (nhóm tương tác/ hành vi – 11 mẫu) gồm: Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy và Visitor. Nhóm này dùng trong thực hiện các hành vi của đối tượng, sự giao tiếp giữa các **object** với nhau.

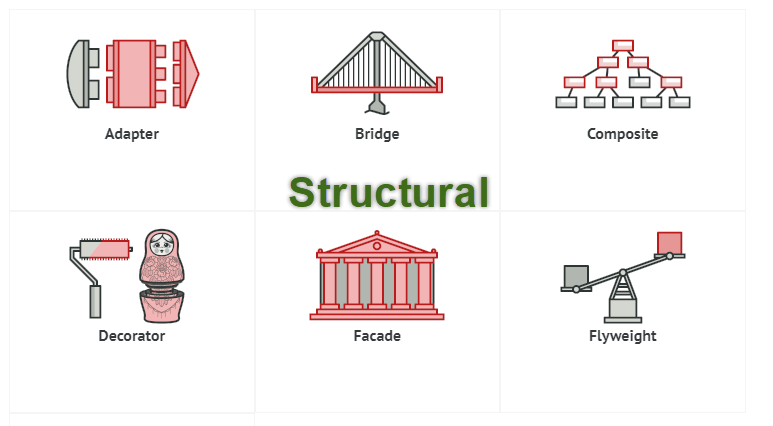


## Nhóm Creational (nhóm khởi tạo)



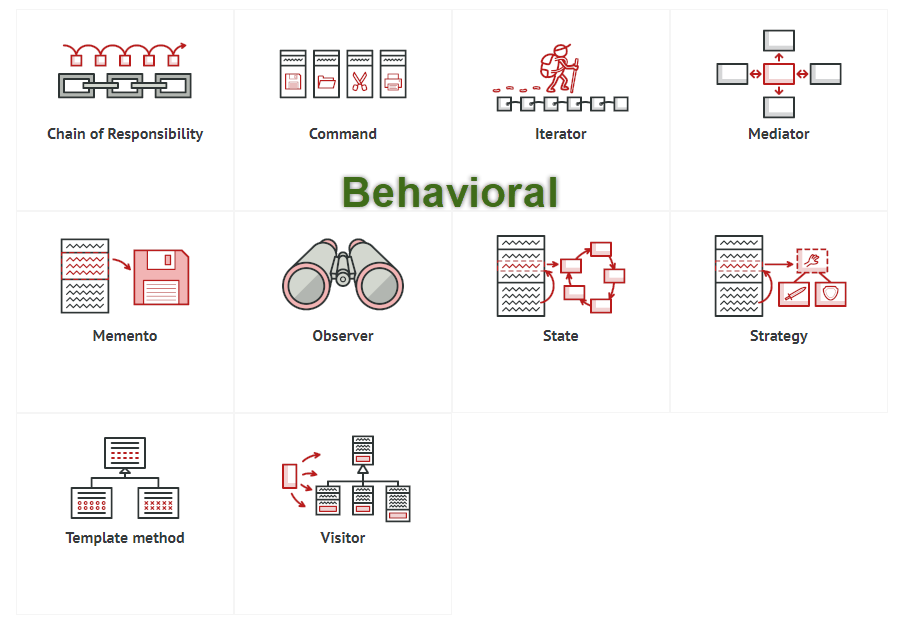
* **Singleton**:
  + Đảm bảo 1 class chỉ có 1 instance và cung cấp 1 điểm truy xuất toàn cục đến nó.
  + Tần suất sử dụng: cao trung bình.
* **Abstract Factory**:
  + Cung cấp một interface cho việc tạo lập các đối tượng (có liên hệ với nhau) mà không cần qui định lớp khi hay xác định lớp cụ thể (concrete) tạo mỗi đối tượng.
  + Tần suất sử dụng: cao.
* **Factory Method**:
  + Định nghĩa Interface để sinh ra đối tượng nhưng để cho lớp con quyết định lớp nào được dùng để sinh ra đối tượng Factory method cho phép một lớp chuyển quá trình khởi tạo đối tượng cho lớp con.
  + Tần suất sử dụng: cao.
* **Builder**:
  + Tách rời việc xây dựng (construction) một đối tượng phức tạp khỏi biểu diễn của nó sao cho cùng một tiến trình xây dựng có thể tạo được các biểu diễn khác nhau.
  + Tần suất sử dụng: trung bình thấp.
* **Prototype**:
  + Qui định loại của các đối tượng cần tạo bằng cách dùng một đối tượng mẫu, tạo mới nhờ vào sao chép đối tượng mẫu này.
  + Tần suất sử dụng: trung bình.

## Nhóm Structural (nhóm cấu trúc)



* **Adapter**:
  + Do vấn đề tương thích, thay đổi interface của một lớp thành một interface khác phù hợp với yêu cầu người sử dụng lớp.
  + Tần suất sử dụng: cao trung bình.
* **Bridge**:
  + Tách rời ngữ nghĩa của một vấn đề khỏi việc cài đặt, mục đích để cả hai bộ phận (ngữ nghĩa và cài đặt) có thể thay đổi độc lập nhau.
  + Tần suất sử dụng: trung bình.
* **Composite**:
  + Tổ chức các đối tượng theo cấu trúc phân cấp dạng cây. Tất cả các đối tượng trong cấu trúc được thao tác theo một cách thuần nhất như nhau.  
    Tạo quan hệ thứ bậc bao gộp giữa các đối tượng. Client có thể xem đối tượng bao gộp và bị bao gộp như nhau -> khả năng tổng quát hoá trong code của client -> dễ phát triển, nâng cấp, bảo trì.
  + Tần suất sử dụng: cao trung bình.
* **Decorator**:
  + Gán thêm trách nhiệm cho đối tượng (mở rộng chức năng) vào lúc chạy (dynamically).
  + Tần suất sử dụng:trung bình.
* **Facade**:
  + Cung cấp một interface thuần nhất cho một tập hợp các interface trong một “hệ thống con” (subsystem). Nó định nghĩa 1 interface cao hơn các interface có sẵn để làm cho hệ thống con dễ sử dụng hơn.
  + Tần suất sử dụng: cao.
* **Flyweight:**
  + Sử dụng việc chia sẻ để thao tác hiệu quả trên một số lượng lớn đối tượng “cở nhỏ” (chẳng hạn paragraph, dòng, cột, ký tự…).
  + Tần suất sử dụng: thấp.
* **Proxy:**
  + Cung cấp đối tượng đại diện cho một đối tượng khác để hỗ trợ hoặc kiểm soát quá trình truy xuất đối tượng đó. Đối tượng thay thế gọi là proxy.
  + Tần suất sử dụng: cao trung bình.

## Nhóm Behavioral (nhóm hành vi/ tương tác)



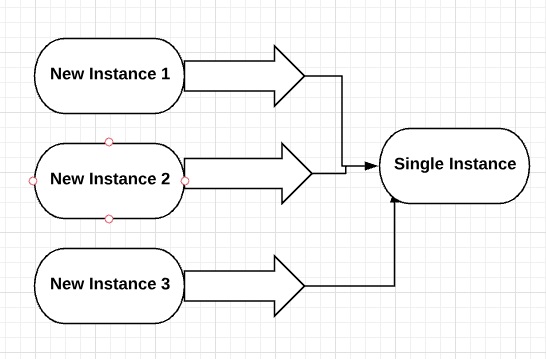
* **Chain of Responsibility:**
  + Khắc phục việc ghép cặp giữa bộ gởi và bộ nhận thông điệp. Các đối tượng nhận thông điệp được kết nối thành một chuỗi và thông điệp được chuyển dọc theo chuỗi nầy đến khi gặp được đối tượng xử lý nó. Tránh việc gắn kết cứng giữa phần tử gởi request với phần tử nhận và xử lý request bằng cách cho phép hơn 1 đối tượng có có cơ hội xử lý request. Liên kết các đối tượng nhận request thành 1 dây chuyền rồi gửi request xuyên qua từng đối tượng xử lý đến khi gặp đối tượng xử lý cụ thể.
  + Tần suất sử dụng: trung bình thấp.
* **Command:**
  + Mỗi yêu cầu (thực hiện một thao tác nào đó) được bao bọc thành một đối tượng. Các yêu cầu sẽ được lưu trữ và gởi đi như các đối tượng.Đóng gói request vào trong một Object, nhờ đó có thể nthông số hoá chương trình nhận request và thực hiện các thao tác trên request: sắp xếp, log, undo…
  + Tần suất sử dụng: cao trung bình.
* **Interpreter:**
  + Hỗ trợ việc định nghĩa biểu diễn văn phạm và bộ thông dịch cho một ngôn ngữ.
  + Tần suất sử dụng: thấp.
* **Iterator:**
  + Truy xuất các phần tử của đối tượng dạng tập hợp tuần tự (list, array, …) mà không phụ thuộc vào biểu diễn bên trong của các phần tử.
  + Tần suất sử dụng: cao.
* **Mediator:**
  + Định nghĩa một đối tượng để bao bọc việc giao tiếp giữa một số đối tượng với nhau.
  + Tần suất sử dụng: trung bình thấp.
* **Memento:**
  + Hiệu chỉnh và trả lại như cũ trạng thái bên trong của đối tượng mà vẫn không vi phạm việc bao bọc dữ liệu.
  + Tần suất sử dụng: thấp.
* **Observer:**
  + Định nghĩa sự phụ thuộc một-nhiều giữa các đối tượng sao cho khi một đối tượng thay đổi trạng thái thì tất cả các đối tượng phụ thuộc nó cũng thay đổi theo.
  + Tần suất sử dụng: cao.
* **State:**
  + Cho phép một đối tượng thay đổi hành vi khi trạng thái bên trong của nó thay đổi, ta có cảm giác như class của đối tượng bị thay đổi.
  + Tần suất sử dụng: trung bình.
* **Strategy:**
  + Bao bọc một họ các thuật toán bằng các lớp đối tượng để thuật toán có thể thay đổi độc lập đối với chương trình sử dụng thuật toán.Cung cấp một họ giải thuật cho phép client chọn lựa linh động một giải thuật cụ thể khi sử dụng.
  + Tần suất sử dụng: cao trung bình.
* **Template method:**
  + Định nghĩa phần khung của một thuật toán, tức là một thuật toán tổng quát gọi đến một số phương thức chưa được cài đặt trong lớp cơ sở; việc cài đặt các phương thức được ủy nhiệm cho các lớp kế thừa.
  + Tần suất sử dụng: trung bình.
* **Visitor:**
  + Cho phép định nghĩa thêm phép toán mới tác động lên các phần tử của một cấu trúc đối tượng mà không cần thay đổi các lớp định nghĩa cấu trúc đó.
  + Tần suất sử dụng: thấp.

Ở 1 số bài viết có giới thiệu tới 32 mẫu design. Tuy nhiên những design đã liệt kê bên trên là các mẫu nguyên thủy nhất và hay được sử dụng nhiều nhất.

# Singleton Pattern

**Singleton pattern là gì?**

* Singleton đảm bảo chỉ duy nhất một thể hiện hay đối tượng (instance) được tạo ra.
* Lớp này cung cấp một method để truy xuất đến đối tượng duy nhất mọi lúc mọi nơi trong chương trình mà không cần khởi tạo đối tượng của lớp.



**Sử dụng khi nào?**

Khi chúng ta muốn:

* Vì class dùng Singleton chỉ tồn tại 1 Instance (thể hiện) nên nó thường được dùng cho các trường hợp giải quyết các bài toán cần truy cập vào các ứng dụng như: Shared resource, Logger, Configuration, Caching, Thread pool, …
* Một số design pattern khác cũng sử dụng Singleton để triển khai: Abstract Factory, Builder, Prototype, Facade…
* Việc quản lý việc truy cập tốt hơn vì chỉ có một thể hiện duy nhất.
* Có thể quản lý số lượng thể hiện của một lớp trong giới hạn chỉ định.

**Cách implement:** Có rất nhiều cách để implement Singleton Pattern. Nhưng dù cho việc implement bằng cách nào đi nữa cũng dựa vào nguyên tắc dưới đây cơ bản dưới đây

* **private constructor** để hạn chế truy cập từ class bên ngoài.
* Đặt **private static final variable** đảm bảo biến chỉ được khởi tạo trong class.
* Có một method **public static** để **return instance** được khởi tạo ở trên.

Những cách nào để implement Singleton Pattern:

## Eager initialization

**Singleton** Class được khởi tạo ngay khi được gọi đến. Đây là cách dễ nhất nhưng nó có một nhược điểm mặc dù instance đã được khởi tạo mà có thể sẽ không dùng tới.

|  |
| --- |
| package com.gpcoder.patterns.creational.singleton;    public class EagerInitializedSingleton {        private static final EagerInitializedSingleton INSTANCE = new EagerInitializedSingleton();        // Private constructor to avoid client applications to use constructor      private EagerInitializedSingleton() {        }        public static EagerInitializedSingleton getInstance() {          return INSTANCE;      }  } |

## Static block initialization

Cách làm tương tự như **Eager initialization** chỉ khác phần **static block** cung cấp thêm lựa chọn cho việc handle exception hay các xử lý khác.

|  |
| --- |
| package com.gpcoder.patterns.creational.singleton;    public class StaticBlockSingleton {        private static final StaticBlockSingleton INSTANCE;        private StaticBlockSingleton() {      }        // Static block initialization for exception handling      static {          try {              INSTANCE = new StaticBlockSingleton();          } catch (Exception e) {              throw new RuntimeException("Exception occured in creating singleton instance");          }      }        public static StaticBlockSingleton getInstance() {          return INSTANCE;      }  } |

## Lazy Initialization

Là một cách làm mang tính mở rộng hơn so với 2 cách làm trên và hoạt động tốt trong môi trường đơn luồng (single-thread).

|  |
| --- |
| package com.gpcoder.patterns.creational.singleton;    public class LazyInitializedSingleton {        private static LazyInitializedSingleton instance;        private LazyInitializedSingleton() {      }        public static LazyInitializedSingleton getInstance() {          if (instance == null) {              instance = new LazyInitializedSingleton();          }          return instance;      }  } |

Cách này đã khắc phục được nhược điểm của cách **Eager initialization**, chỉ khi nào **getInstance()** được gọi thì instance mới được khởi tạo. Tuy nhiên, cách này chỉ sử dụng tốt trong trường hợp đơn luồng (single-thread), trường hợp nếu có nhiều luồng (multi-thread) cùng chạy và cùng gọi hàm getInstance() tại cùng một thời điểm thì có thể có nhiều hơn 1 thể hiện của instance. Để khắc phục nhược điểm này chúng ta sử dụng **Thread Safe Singleton**.

Một nhược điểm nữa của **Lazy Initialization** cần quan tâm là: đối với thao tác create instance quá chậm thì người dùng có phải chờ lâu cho lần sử dụng đầu tiên.

## Thread Safe Singleton

Cách đơn giản nhất là chúng ta gọi phương thức **synchronized** của hàm **getInstance()** và như vậy hệ thống đảm bảo rằng tại cùng một thời điểm chỉ có thể có 1 luồng có thể truy cập vào hàm getInstance() và đảm bảo rằng chỉ có duy nhất 1 thể hiện của class.

|  |
| --- |
| package com.gpcoder.patterns.creational.singleton;    public class ThreadSafeLazyInitializedSingleton {        private static volatile ThreadSafeLazyInitializedSingleton instance;        private ThreadSafeLazyInitializedSingleton() {      }        public static synchronized ThreadSafeLazyInitializedSingleton getInstance() {          if (instance == null) {              instance = new ThreadSafeLazyInitializedSingleton();          }          return instance;      }  } |

Cách này có nhược điểm là một phương thức **synchronized** sẽ chạy rất chậm và tốn hiệu năng, bất kỳ Thread nào gọi đến đều phải chờ nếu có một Thread khác đang sử dụng. Có những tác vụ xử lý trước và sau khi tạo thể hiện không cần thiết phải block. Vì vậy chúng ta cần cải tiến nó đi 1 chút với **Double Check Locking Singleton**.

## Double Check Locking Singleton

Để implement theo cách này, chúng ta sẽ kiểm tra sự tồn tại thể hiện của lớp, với sự hổ trợ của đồng bộ hóa, hai lần trước khi khởi tạo. Phải khai báo **volatile** cho instance để tránh lớp làm việc không chính xác do quá trình tối ưu hóa của trình biên dịch.

|  |
| --- |
| **package** com.gpcoder.patterns.creational.singleton;    **public** **class** DoubleCheckLockingSingleton {    **private** **static** **volatile** DoubleCheckLockingSingleton instance;    **private** DoubleCheckLockingSingleton() {      }    **public** **static** DoubleCheckLockingSingleton getInstance() {          // Do something before get instance ...  **if** (instance == **null**) {              // Do the task too long before create instance ...              // Block so other threads cannot come into while initialize  **synchronized** (DoubleCheckLockingSingleton.**class**) {                  // Re-check again. Maybe another thread has initialized before  **if** (instance == **null**) {                      instance = **new** DoubleCheckLockingSingleton();                  }              }          }          // Do something after get instance ...  **return** instance;      }  } |

## Bill Pugh Singleton Implementation

Với cách làm này bạn sẽ tạo ra static nested class với vai trò 1 Helper khi muốn tách biệt chức năng cho 1 class function rõ ràng hơn. Đây là cách thường hay được sử dụng và có hiệu suất tốt (theo các chuyên gia đánh giá 🙂 ).

|  |
| --- |
| **package** com.gpcoder.patterns.creational.singleton;    **public** **class** BillPughSingleton {    **private** BillPughSingleton() {      }    **public** **static** BillPughSingleton getInstance() {  **return** SingletonHelper.INSTANCE;      }    **private** **static** **class** SingletonHelper {  **private** **static** **final** BillPughSingleton INSTANCE = **new** BillPughSingleton();      }  } |

## Phá vỡ cấu trúc Singleton Pattern bằng Reflection.

Reflection có thể được dùng để phá vỡ Pattern của Eager Initialization ở trên. Ví dụ

|  |
| --- |
| **package** com.gpcoder.patterns.creational.singleton;    **import** java.lang.reflect.Constructor;  **import** java.lang.reflect.InvocationTargetException;    **public** **class** ReflectionBreakSingleton {    **public** **static** **void** main(String[] args)  **throws** InstantiationException, IllegalAccessException, InvocationTargetException {            EagerInitializedSingleton instanceOne = EagerInitializedSingleton.getInstance();          EagerInitializedSingleton instanceTwo = **null**;            Constructor<?>[] constructors = EagerInitializedSingleton.**class**.getDeclaredConstructors();  **for** (Constructor<?> constructor : constructors) {              constructor.setAccessible(**true**);              instanceTwo = (EagerInitializedSingleton) constructor.newInstance();          }            System.out.println(instanceOne.hashCode());          System.out.println(instanceTwo.hashCode());      }  } |

Output của chương trình:

1 2018699554

2 1311053135

Do đó, khi cần sử dụng Eager Initialization chúng ta nên implement theo Bill Pugh Singleton để không bị break bởi Reflection và cũng đạt được hiệu suất tốt hơn.

## Enum Singleton

Khi dùng enum thì các params chỉ được khởi tạo 1 lần duy nhất, đây cũng là cách giúp bạn tạo ra Singleton instance.

|  |
| --- |
| **package** com.gpcoder.patterns.creational.singleton;    /\*\*   \* Singleton implementation using enum initialization   \*/  **public** **enum** EnumSingleton {        INSTANCE;  } |

Lưu ý:

- Enum có thể sử dụng như một Singleton, nhưng nó có nhược điểm là không thể extends từ một lớp được, nên khi sử dụng cần xem xét vấn đề này.

- Hàm constructor của enum là lazy, nghĩa là khi được sử dụng mới chạy hàm khởi tạo và nó chỉ chạy duy nhất một lần. Nếu muốn sử dụng như một eager singleton thì cần gọi thực thi trong một static block khi start chương trình.

So sánh giữa 2 cách sử dụng enum initialization và static block initialization method, enum có một điểm rất mạnh khi giải quyết về vấn đề Serialization/ Deserialization.

## Serialization and Singleton

Đôi khi trong các hệ thống phân tán (distributed system), chúng ta cần implement interface Serializable trong lớp Singleton để chúng ta có thể lưu trữ trạng thái của nó trong file hệ thống và truy xuất lại nó sau.

|  |
| --- |
| **package** com.gpcoder.patterns.creational.singleton;    **import** java.io.ObjectStreamException;  **import** java.io.Serializable;    **public** **class** SerializedSingleton **implements** Serializable {    **private** **static** **final** **long** serialVersionUID = 1741825395699241705L;    **private** SerializedSingleton() {      }    **private** **static** **class** SingletonHelper {  **private** **static** **final** SerializedSingleton instance = **new** SerializedSingleton();      }    **public** **static** SerializedSingleton getInstance() {  **return** SingletonHelper.instance;      }        /\*\*       \* Special hook provided by serialization where developer can control what object needs to sent.       \* However this method is invoked on the new object instance created by de serialization process.       \*       \* @return       \* @throws ObjectStreamException       \*/  //    private Object readResolve() throws ObjectStreamException {  //        return SingletonHelper.instance;  //    }    } |

|  |
| --- |
| **package** com.gpcoder.patterns.creational.singleton;    **import** java.io.FileInputStream;  **import** java.io.FileNotFoundException;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.ObjectInput;  **import** java.io.ObjectInputStream;  **import** java.io.ObjectOutput;  **import** java.io.ObjectOutputStream;    **public** **class** SingletonSerializedTest {    **public** **static** **void** main(String[] args) **throws** FileNotFoundException, IOException, ClassNotFoundException {            SerializedSingleton serializedSingleton1 = SerializedSingleton.getInstance();          EnumSingleton enumSingleton1 = EnumSingleton.INSTANCE;            ObjectOutput out = **new** ObjectOutputStream(**new** FileOutputStream("SingletonSerializedTest.txt"));          out.writeObject(serializedSingleton1);          out.writeObject(enumSingleton1);          out.close();            // De-serialize from file to object          ObjectInput in = **new** ObjectInputStream(**new** FileInputStream("SingletonSerializedTest.txt"));          SerializedSingleton serializedSingleton2 = (SerializedSingleton) in.readObject();          EnumSingleton enumSingleton2 = (EnumSingleton) in.readObject();          in.close();            System.out.println("serializedSingleton1 hashCode=" + serializedSingleton1.hashCode());          System.out.println("serializedSingleton2 hashCode=" + serializedSingleton2.hashCode());          System.out.println("enumSingleton1 hashCode=" + enumSingleton1.hashCode());          System.out.println("enumSingleton2 hashCode=" + enumSingleton2.hashCode());      }  } |

Kết quả:

|  |
| --- |
| serializedSingleton1 hashCode=1028566121  serializedSingleton2 hashCode=1747585824  enumSingleton1 hashCode=1118140819  enumSingleton2 hashCode=1118140819 |

Như trong ví dụ trên, Deserialize đối tượng của SerializedSingleton khác với đối tượng gốc. Tuy nhiên vấn đề này không xảy ra khi sử dụng enum.

Thực tế thì vẫn có cách khắc phục khi sử dụng class SerializedSingleton là implement một phương thức readResolve(). Nhưng khi chúng ta thật sự gặp vấn đề và cần sử dụng Serialize/ Deserialize, thì nên sử dụng enum sẽ đơn giản hơn.

Singleton trong Python:

|  |
| --- |
| **class** Singleton:  \_\_instance = **None** @staticmethod  **def** getInstance():  **if** Singleton.\_\_instance == **None**:  Singleton()  **return** Singleton.\_\_instance   **""" Python khong co private constructor"""  def** \_\_init\_\_(self):  **if** Singleton.\_\_instance != **None**:  **raise** Exception(**"This class is singleton"**)  **else**:  Singleton.\_\_instance = self  s2 = Singleton() print(s2)  s = Singleton.getInstance() print(s)  s2 = Singleton.getInstance() print(s2) |

# Factory Pattern

**Factory pattern là gì?**

Factory Method Design Pattern hay gọi ngắn gọn là Factory Pattern là một trong những Pattern thuộc nhóm Creational Design Pattern. Nhiệm vụ của Factory Pattern là quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đổi tượng một cách linh hoạt hơn.

Factory Pattern đúng nghĩa là một nhà máy, và nhà máy này sẽ “sản xuất” các đối tượng theo yêu cầu của chúng ta.

Trong Factory Pattern, chúng ta tạo đối tượng mà không để lộ logic tạo đối tượng ở phía người dùng và tham chiếu đến đối tượng mới được tạo ra bằng cách sử dụng một interface chung.

Factory Pattern được sử dụng khi có một class cha (super-class) với nhiều class con (sub-class), dựa trên đầu vào và phải trả về 1 trong những class con đó.

# Tham khảo

Top các cuốn sách hay về Design Pattern:

* Design Patterns: Elements of Reusable Object-Oriented Software.
* Head First Design Patterns.
* Pattern Hatching: Design Patterns Applied.
* Refactoring to Patterns.
* Patterns of Enterprise Application Architecture.

Các link tham khảo:

<https://www.tutorialspoint.com/python_design_patterns/index.htm>

<https://www.tutorialspoint.com/design_pattern/index.htm>

<https://gpcoder.com/4164-gioi-thieu-design-patterns/>