**Creational Design Patterns**

**1) Abstract Factory**

* Abstract Factory Pattern is a super-factory creates other factories This factory is called as Factory of factories

**2) Factory**

* In Factory Pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface
* In Simple words, If we have a super class and n sub-classes, and based on the data provided, we have to return the object of the one of the sub-classes
* The basic principle behind this pattern is that, at run time, we get an object of similar type based on the parameter we pass
* If object creation code is spread in whole application, and if you need to change the process of object creation then you need to go in each and every to make necessary changes

**3) Prototype**

* Prototype Pattern is used for creating new objects (instances) by cloning (copying) other objects and this way we can improve the performance
* This pattern uses in java cloning to copy the object
* The pattern is used when creation of object is costly or complex. Example a object is to be created after a costly database operation. We can cache the object, return its clone on next request. Once we get the cloned object we can modify according to our needs.

**4) Builder**

* Builder pattern builds a complex object using simple objects and using a step by step approach.
* The process of constructing complex object should be generic so that the same process can be used to create different representations of the same complex object.

**5) Singleton**

* Singleton pattern is a design solution where an application wants to have one and only one instance of the class
* We can access the singleton instance Globally
* Singleton pattern ensures that there is only one instance of a class is created in the JVM
* Singleton patterns are used in logging, caches, thread pools, configuration settings, device driver objects
* To prevent your singleton from Reflection API, just throw an IllegalStateException from private constructor if anybody tries to create second instance
* To protect the singleton from de-serialization, you have to implement readResolve() method in your singleton class. It is called when the object is de-serialized.
* As synchronized method/block causing the performance issues, Bill Pugh came up with a different approach to implement singleton design pattern using private static inner class. This method is thread safe and doesn’t require synchronization.

Enum Singleton:

* As all enums are initialized only once at the time of class loading, they are singleton by default. Enum singletons are thread safe and they can’t be broken by the reflection. But only limitation is that they are eagerly instantiated.

//Private Static inner class which is loaded when getInstance() is called for the first time

**private** **static** **class** SingletonHelper

    {

**private** **static** **final** Singleton instance = **new** Singleton();

    }

Ex:

/\*\*  
 \* Hard-to-get-it-right implementation of a Singleton supporting lazy  
 \* instantiation.  
 \*/  
public class Singleton implements Cloneable, Serializable {  
 private static final long serialVersionUID = 6462568326558031828L;   
  
 /\*  
 \* Marking volatile is necessary for avoiding the threads from creating  
 \* their own copies, even with double checked locking.  
 \*/  
 private static volatile Singleton INSTANCE = null;  
   
 /\*  
 \* Guard against 'reflection' to create a new instance.  
 \*/  
 private Singleton() {  
 if (INSTANCE != null) {  
 throw new IllegalStateException(“Instance already created”);  
 }  
 }  
  
 /\*  
 \* Synchronizes only on the critical section, and uses double checked  
 \* locking to ensure that all get requests are not Synchronized once the  
 \* instance is created.  
 \*/  
 public static Singleton getInstance() {  
 if (INSTANCE == null) {  
 synchronized (Singleton.class) {  
 if (INSTANCE == null) {  
 INSTANCE = new Singleton();  
 }  
 }  
 }  
 return INSTANCE;  
 }  
  
 /\*\*  
 \* Yeah, why dont you try cloning me?  
 \*/  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 throw new CloneNotSupportedException();  
  
 }

// To protect the singleton from de-serialization, you have to implement readResolve() method in your singleton class

readResolve() method : returns the existing instance

protected Object readResolve() throws ObjectStreamException{

return INSTANCE;

}  
}

**Structural Design Patterns**

**1) Adapter**

* Adapter pattern works as a bridge between two incompatible interfaces
* This pattern involves a single class called adapter which is responsible for communication between two independent or incompatible interfaces.
* An adapter helps two incompatible interfaces to work together

**2) Decorator**

* Decorator Pattern allows to add new functionality to an existing object without altering its structure and this pattern acts as a wrapper to the existing class.
* This pattern dynamically changes the functionality of an object at runtime without impacting the existing functionality of the objects. In short this pattern adds additional functionalities to the object by wrapping it.
* Decorator is an object that adds features to another object.

**3) Proxy**

* Proxy provides a surrogate or placeholder for another object to control access to it.
* Proxy means 'in place of' or 'Representing' or 'on behalf of'.
* Proxy is the object that is being called by the client to access the real object behind the scene.
* In Proxy pattern, a class represents functionality of another class.

**4) Facade**

* Make a complex system simpler by providing a general interface
* Minimize the communication and dependency between subsystems

**Behavioral Design Patterns**

**1) Command**

* Command Pattern is used to encapsulate a request as an object[command] and pass to an invoker, wherein the invoker does not know how to service the request, but uses the encapsulated command to perform an action.

**2) Template**

* Template Pattern defines a sequence of steps of an algorithm
* The subclasses are allowed to override the steps but not allowed to change the sequence
* The key to the template method pattern is that we put the general logic in the abstract parent class and let the child class define the specifics

**3) Strategy**

* Strategy pattern is used when we have multiple algorithms for a specific task and client decides the actual implementation to be used at runtime
* It attempts to solve the issue where you need to provide multiple solutions for the same problem so that one can be selected at runtime

**4) Observer**

* Observer Pattern defines one to many relationships between Observable and observers
* When Observable changes the state, then all Observers are notified.
* Observer can be used in event Handling
* Example: Updates stocks in Stock market

**J2EE Patterns**

MVC

**Service Locator**

* The Service Locator Design Pattern is used when we want locate various services(JMS Service, EJB Service) using JNDI lookup.
* Considering high cost of looking up JNDI for a service, Service Locator pattern makes use of caching technique.

**Business Delegate**

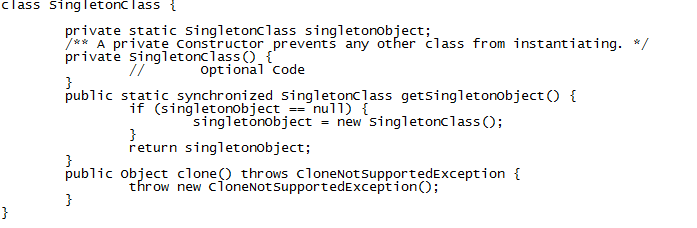
* It is used to reduce remote lookup functionality to business tier code in presentation tier code.
* We want to hide clients from the complexity of remote communication with business service components then we can use Business Delegate Pattern
* This reason for this approach is that a too strong coupling between the two ways [presentation and business layers] can cause various problems

**Front Controller**

* The Front Controller pattern provides a centralized request handling mechanism so that all requests will be handled by a single handler.
* Front Controller is a centralized entry point for handling requests
* Front Controller will do the authentication/ authorization /logging of tracking of request and then pass the requests to corresponding handlers.

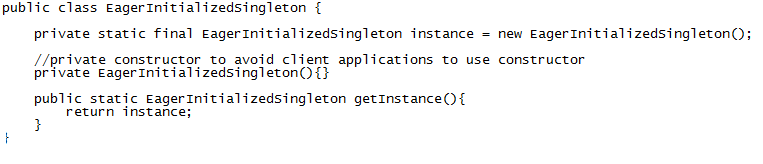
1. Tell me the code for Singleton Design pattern and necessary steps to follow

* Create a default Private constructor
* Create a Method for getting the reference to Singleton Object
* Make the Access method Synchronized to prevent Thread Problems.
* Override the Object clone method to prevent cloning



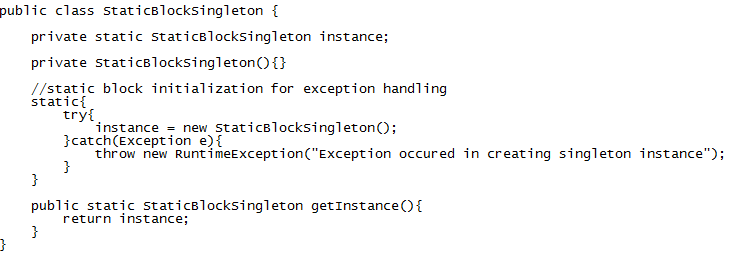
1. Tell me the code for Singleton Design pattern using Eager initialization

* In eager initialization, the instance of Singleton Class is created at the time of class loading
* Drawback is that instance will be creating at class loading even though client application might not be using it



1. Tell me the code for Singleton Design pattern using Static block

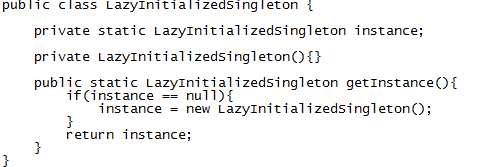
* Static block initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for exception handling



* Both eager initialization and static block initialization creates the instance even before it’s being used and that is not the best practice to use.

1. Tell me the code for Singleton Design pattern using Lazy initialization

* Lazy initialization method to implement Singleton pattern creates the instance in the global access method. Here is the sample code for creating Singleton class with this approach



* The above implementation works fine in case of single threaded environment but when it comes to multithreaded systems, it can cause issues if multiple threads are inside the if loop at the same time. It will destroy the singleton pattern and both threads will get the different instances of singleton class.

1. How to make Singleton Design pattern as thread safe and tell me the code snippet

* The easier way to create a thread-safe singleton class is to make the global access method synchronized, so that only one thread can execute this method at a time. General implementation of this approach is like the below class

**Circuit Breaker**

* **Circuit Breaker:**Every request passes through the circuit breaker component which verifies from the Health Checker component the overall health of dependent services for that API.
* If the overall health status is not healthy then the circuit breaker component breaks the circuit i.e. instead of passing the request to request processor it returns 5xx response to the client.
* The client will retry the request after a delay.
* **Implementation:** We can implement this pattern Netflix Hystrix library
* Hystrix is a **latency** and **fault tolerance** library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.

**S.O.L.I.D Principles:**

**Single Responsibility Principle (SRP):**

* The single responsibility principle states that **every Java class must perform a single functionality.**
* Implementation of multiple functionalities in a single class mashup the code and if any modification is required it may affect the whole class

|  |
| --- |
| * Better maintainability and separation of concerns. |

**Open-Closed Principle (OCP):**

* The open-closed principle states that according to new requirements **the module should be open for extension but closed for modification.**
* We should be able to add new functionality without modifying existing code.

|  |
| --- |
| * New features can be added without modifying existing code. |

**Liskov Substitution Principle (LSP):**

* It applies to inheritance in such a way that the **derived classes must be completely substitutable for their base classes**.
* In other words, if class A is a subtype of class B, then we should be able to replace B with A without interrupting the behavior of the program.
* Subclasses should be **replaceable** for their base class without breaking functionality.

|  |
| --- |
| * Prevents unexpected behaviors. |

**Interface Segregation Principle (ISP):**

* The principle states that the larger interfaces split into smaller ones. Because the implementation classes use only the methods that are required. We should not force the client to use the methods that they do not want to use.
* Clients should not be forced to implement **unnecessary methods**.

|  |
| --- |
| * Avoids bloated interfaces. |

**Dependency Inversion Principle (DIP):**

* The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations.
* High-level modules should not depend on the low-level module but both should depend on abstraction. Because the abstraction does not depend on detail, but the detail depends on abstraction.
* Depends on abstraction, but not concretions.
* Improves flexibility and testability.

**Domain Driven Design Pattern (DDD):**

* Stop adding functionality to the monolithic application.
* Split the frontend from the backend
* It helps in structuring **microservices** by dividing them into **independent business domains** to ensure **scalability, maintainability, and clarity**
* Extract monolithic modules into micro services. Decompose and decouple the monolith into a series of micro services.
* Prevents **tight coupling** between microservices.
* Each **microservice is independent** and focuses on a single domain.