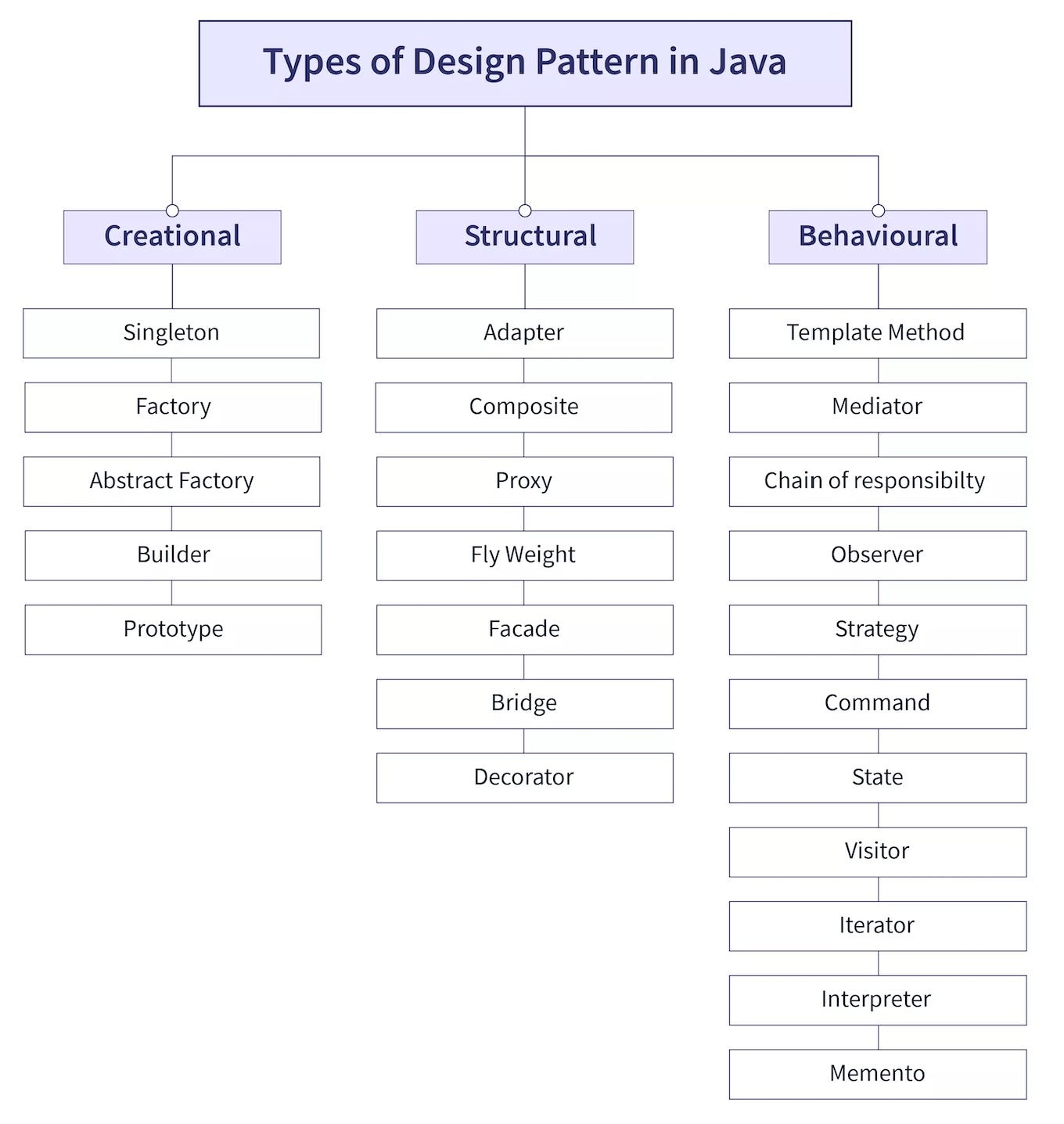
**Design Patterns**



**Creational**

**Singleton Design Pattern**

Singleton says that a class that has only one instance provide for global point of access.

Advantage is Save the memory because object is not created only single instance will reuse at each request.

There are two form for Creation of Singleton Pattern.

Early (Create at load time) and Lazy (Create when required)

Rule for Creation Singleton Design Pattern

1. Create Private Static instance.
2. Create Private Constructor.
3. Create Static method for access.

Early or Eager Instantiation

In simple word instance will create at the time of loading the class by JVM.

public class EagerSingleton {  
 private static final EagerSingleton *instance* = new EagerSingleton();  
  
 public EagerSingleton()  
 {  
 System.*out*.println("EagerSingleton Constructor Call");  
 }  
  
 public static EagerSingleton getInstance()  
 {  
 return *instance*;  
 }  
}

Lazy Instantiation

It means the initialization will until the first time it call then it will create object.

public class LazySingleton {  
 private static LazySingleton *instance*;  
  
 private LazySingleton()  
 {  
  
 }  
  
 public static LazySingleton getInstance()  
 {  
 if (*instance* == null)  
 {  
 return *instance* = new LazySingleton();  
 }else {  
 return *instance*;  
 }  
 }  
  
}

Now in in this approach while the multithread environment works then we need to add Synchronized keyword.

To make singleton class thread safe that’s why **we make getInstance() method is made synchronized so that multiple thread can’t access it simultaneously.**

public class LazySingleton {  
 private static LazySingleton *instance*;  
  
 private LazySingleton()  
 {  
  
 }  
  
 public static synchronized LazySingleton getInstance()  
 {  
 if (*instance* == null)  
 {  
 return *instance* = new LazySingleton();  
 }else {  
 return *instance*;  
 }  
 }  
  
}

Now we also create a double checking lazy singleton class for thread safe.

public class LazyDoubleCheckSingleton {  
 private static LazyDoubleCheckSingleton *instance*;  
  
 private LazyDoubleCheckSingleton()  
 {  
  
 }  
  
 public static LazyDoubleCheckSingleton getInstance() {  
 if (*instance* == null) {  
 synchronized (LazyDoubleCheckSingleton.class) {  
 if (*instance* == null) {  
 return *instance* = new LazyDoubleCheckSingleton();  
 }  
 }  
 }  
 return *instance*;  
 }  
  
}

Now Lazy Inner Class Singleton

In this approach the inner class cannot get loaded until the getInstance() method call for the first time so that this solution is thread safe we don’t need to add manually the Synchronized.

So this is the most efficient approach for singleton design pattern with Thread safe.

public class LazyInnerSingleton {  
 private LazyInnerSingleton(){}  
 private static class SingletonHelper  
 {  
 private static final LazyInnerSingleton *instance* = new LazyInnerSingleton();  
 }  
  
 public static LazyInnerSingleton getInstance()  
 {  
 return SingletonHelper.*instance*;  
 }  
}

How to Prevent from Clone method of singleton class

If we extend the clone class in singleton class the we should override clone method and throw CloneNotSupportException.

public class LazySingleton extends MyClone{  
 private static LazySingleton *instance*;  
  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 throw new CloneNotSupportedException("Do not Clone this singleton class");  
 }  
  
 private LazySingleton()  
 {  
  
 }  
  
 public static synchronized LazySingleton getInstance()  
 {  
 if (*instance* == null)  
 {  
 return *instance* = new LazySingleton();  
 }else {  
 return *instance*;  
 }  
 }  
  
}