Design Patterns

Contents

[Design pattern Types: 1](#_Toc508710081)

[Creational Design Patterns: 2](#_Toc508710082)

[Structural Design Patterns: 2](#_Toc508710083)

[Behavioural Design Patterns: 2](#_Toc508710084)

[Examples: 3](#_Toc508710085)

[Factory Pattern: 3](#_Toc508710086)

[Factory Design Pattern Advantages 6](#_Toc508710087)

[Factory Design Pattern Examples in JDK 6](#_Toc508710088)

[Abstract Factory Pattern: 6](#_Toc508710089)

[Singleton Pattern: 12](#_Toc508710090)

[Eager initialization 12](#_Toc508710091)

[Static block initialization 13](#_Toc508710092)

[Lazy Initialization 13](#_Toc508710093)

[Thread Safe Singleton 14](#_Toc508710094)

[Bill Pugh Singleton Implementation 15](#_Toc508710095)

[Using Reflection to destroy Singleton Pattern 16](#_Toc508710096)

[Enum Singleton 17](#_Toc508710097)

[Serialization and Singleton 18](#_Toc508710098)

[Note: 20](#_Toc508710099)

[Observer pattern: 20](#_Toc508710100)

[Facade Pattern: 20](#_Toc508710101)

[Facade Design Pattern Interface 21](#_Toc508710102)

[Facade Design Pattern Client Program 22](#_Toc508710103)

[Facade Design Pattern Important Points 23](#_Toc508710104)

# 

# Design pattern Types:

# 

|  |  |
| --- | --- |
| **S.N.** | **Pattern & Description** |
| 1 | **Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case. |
| 2 | **Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities. |
| 3 | **Behavioural Patterns** These design patterns are specifically concerned with communication between objects. |
| 4 | **J2EE Patterns** These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Centre. |

# Creational Design Patterns:

* Singleton Pattern
* Factory Pattern
* Abstract Factory Pattern
* Builder Pattern
* Prototype Pattern

# Structural Design Patterns:

* Adapter Pattern
* Composite Pattern
* Proxy Pattern
* Flyweight Pattern
* Facade Pattern
* Bridge Pattern
* Decorator Pattern

# Behavioural Design Patterns:

* Template Method Pattern
* Mediator Pattern
* Chain of Responsibility Pattern
* Observer Pattern
* Strategy Pattern
* Command Pattern
* State Pattern
* Visitor Pattern
* Interpreter Pattern
* Iterator Pattern
* Memento Pattern

# Examples:

## Factory Pattern:

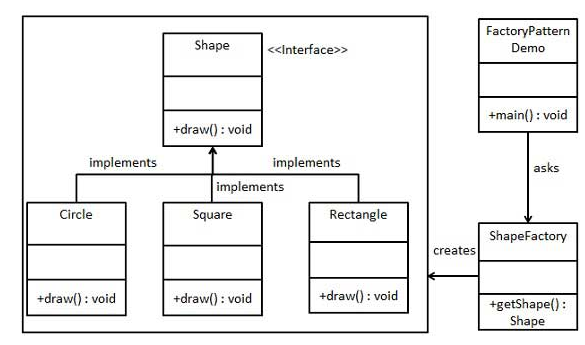
Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern take out the responsibility of instantiation of a class from client program to the factory class.

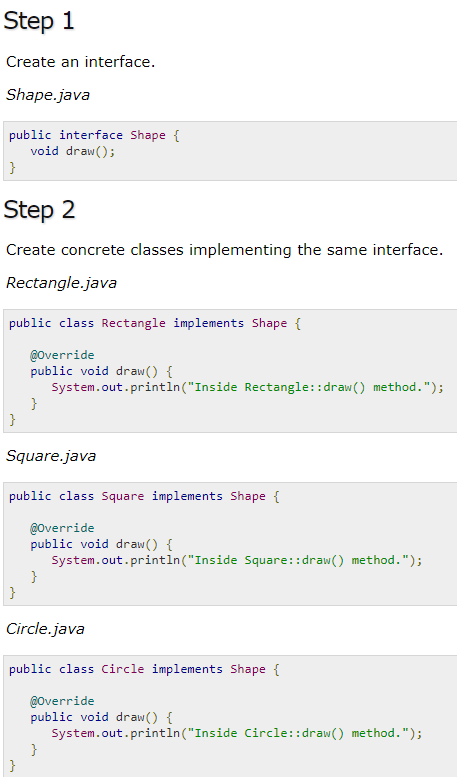
Or

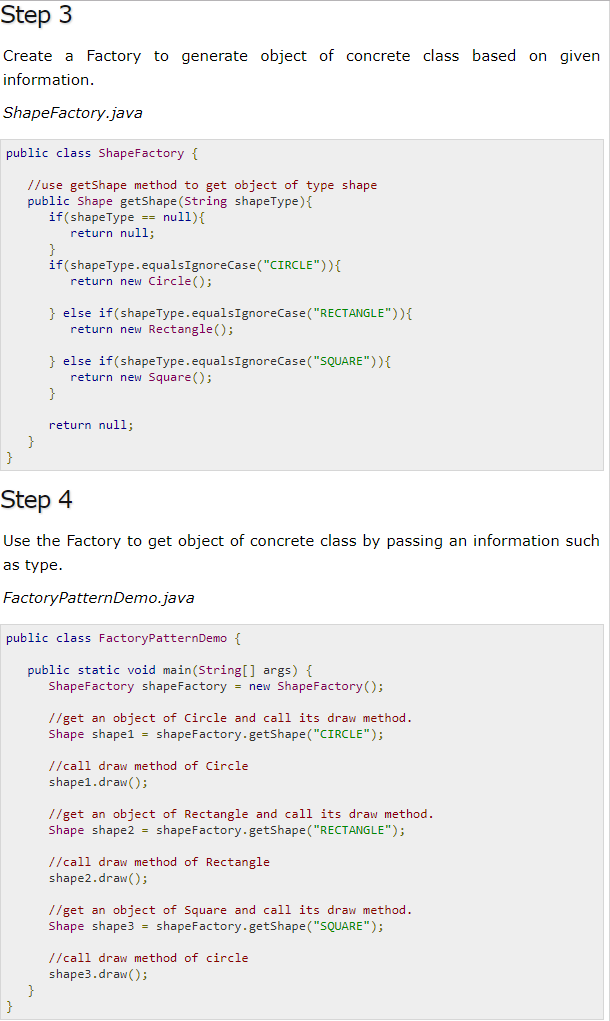
In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Note:

1. Super class in factory design pattern can be an interface, [**abstract class**](https://www.journaldev.com/1582/abstract-class-in-java) or a normal java class
2. We can keep Factory class Singleton or we can keep the method that returns the subclass as [static](https://www.journaldev.com/1365/static-keyword-in-java).







### Factory Design Pattern Advantages

1. Factory design pattern provides approach to code for interface rather than implementation.
2. Factory pattern removes the instantiation of actual implementation classes from client code. Factory pattern makes our code more robust, less coupled and easy to extend.
3. Factory pattern provides abstraction between implementation and client classes through inheritance.

### Factory Design Pattern Examples in JDK

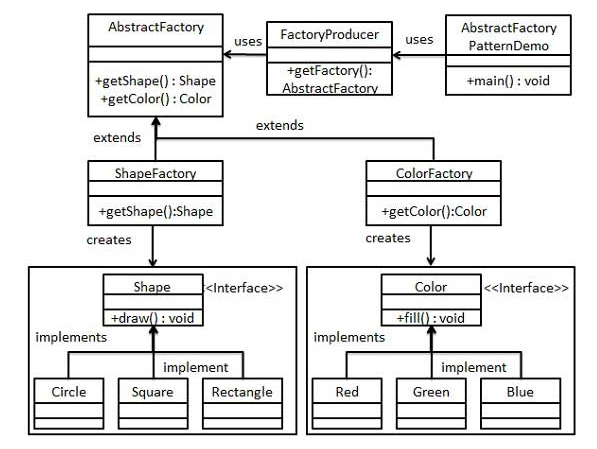
1. java.util.Calendar, Resource Bundle and Number Format getInstance() methods uses Factory pattern.
2. valueOf() method in wrapper classes like Boolean, Integer etc.

## Abstract Factory Pattern:

Abstract Factory pattern is like Factory pattern and its factory of factories. In factory design pattern, notice that we have a single Factory class that returns the different sub-classes based on the input provided and factory class uses if-else or switch statement to achieve this.

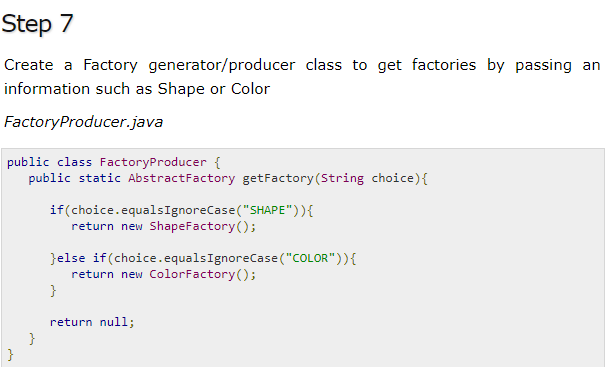
In Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class and then an Abstract Factory class that will return the sub-class based on the input factory class. **OR**

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.









## Singleton Pattern:

Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.

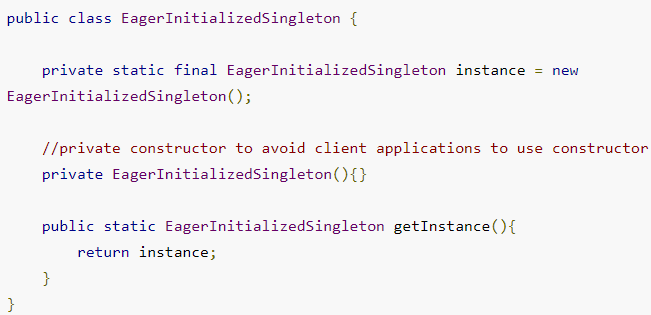
* Private constructor to restrict instantiation of the class from other classes.
* Private static variable of the same class that is the only instance of the class.
* Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

Singleton pattern implementation and design concerns with the implementation.

* Eager initialization
* Static block initialization
* Lazy Initialization
* Thread Safe Singleton
* Bill Pugh Singleton Implementation
* Using Reflection to destroy Singleton Pattern
* Enum Singleton
* Serialization and Singleton

### Eager initialization

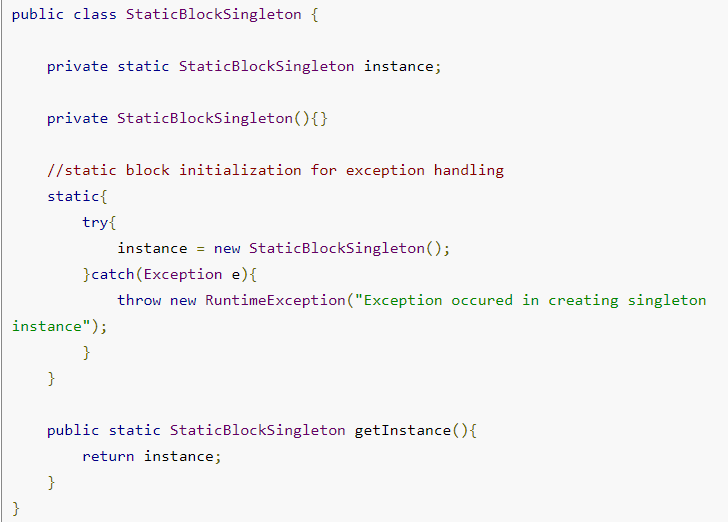
In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not be using it. Here is the implementation of static initialization singleton class.



If your singleton class is not using a lot of resources, this is the approach to use. But in most of the scenarios, Singleton classes are created for resources such as File System, Database connections etc and we should avoid the instantiation until unless client calls the getInstance method. Also this method doesn’t provide any options for exception handling.

### Static block initialization

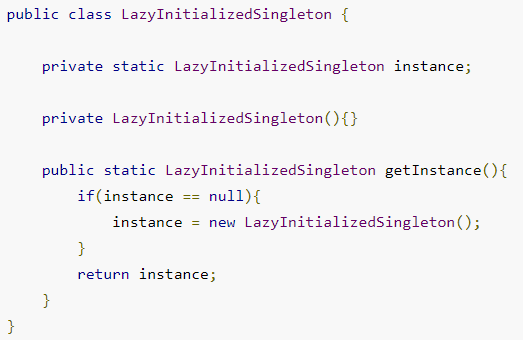
[Static block](https://www.journaldev.com/1365/static-keyword-in-java) initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for [exception handling](https://www.journaldev.com/1696/exception-handling-in-java).



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. So in further sections, we will learn how to create Singleton class that supports lazy initialization.

### Lazy Initialization

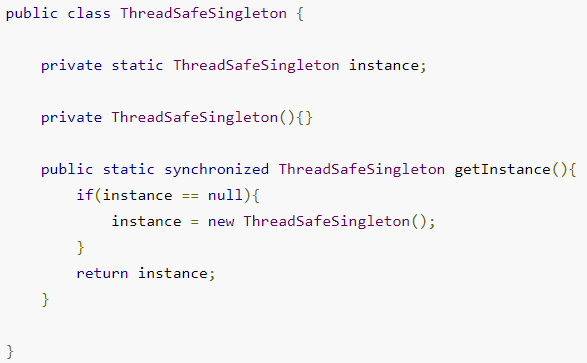
Lazy initialization method to implement Singleton pattern creates the instance in the global access method. Here is the sample code for creating Singleton class with this approach.



The above implementation works fine incase 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. In next section, we will see different ways to create a [thread-safe](https://www.journaldev.com/1061/thread-safety-in-java) singleton class.

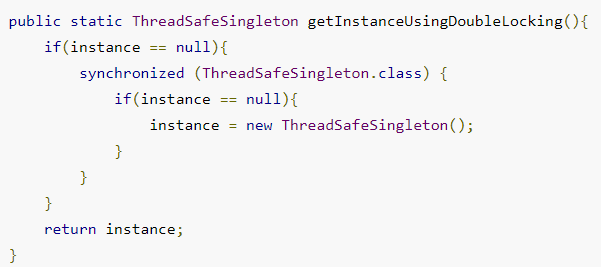
### Thread Safe Singleton

The easier way to create a thread-safe singleton class is to make the global access method [synchronized](https://www.journaldev.com/1061/thread-safety-in-java), so that only one thread can execute this method at a time. General implementation of this approach is like the below class.



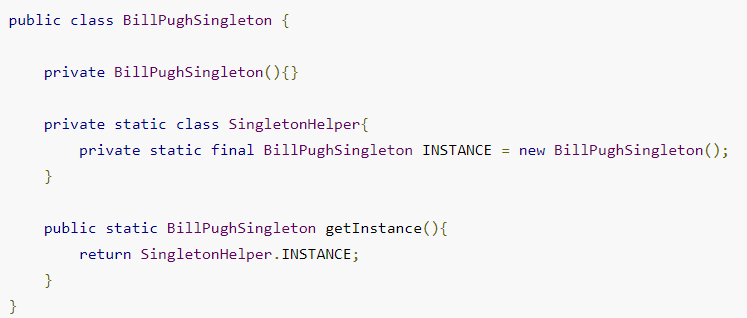
Above implementation works fine and provides thread-safety but it reduces the performance because of cost associated with the synchronized method, although we need it only for the first few threads who might create the separate instances To avoid this extra overhead every time, **double checked locking** principle is used. In this approach, the synchronized block is used inside the if condition with an additional check to ensure that only one instance of singleton class is created.

Below code snippet provides the double checked locking implementation.



### Bill Pugh Singleton Implementation

Prior to Java 5, java memory model had a lot of issues and above approaches used to fail in certain scenarios where too many threads try to get the instance of the Singleton class simultaneously. So Bill Pugh came up with a different approach to create the Singleton class using a [inner static helper class](https://www.journaldev.com/996/java-inner-class). The Bill Pugh Singleton implementation goes like this;



Notice the **private inner static class** that contains the instance of the singleton class. When the singleton class is loaded, Singleton Helper class is not loaded into memory and only when someone calls the getInstance method, this class gets loaded and creates the Singleton class instance.

This is the most widely used approach for Singleton class as it doesn’t require synchronization. I am using this approach in many of my projects and it’s easy to understand and implement also.

### Using Reflection to destroy Singleton Pattern

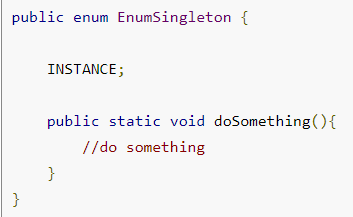
Reflection can be used to destroy all the above singleton implementation approaches. Let’s see this with an example class.



When you run the above test class, you will notice that hashCode of both the instances are not same that destroys the singleton pattern. Reflection is very powerful and used in a lot of frameworks like Spring and Hibernate, do check out [**Java Reflection Tutorial**](https://www.journaldev.com/1789/java-reflection-example-tutorial).

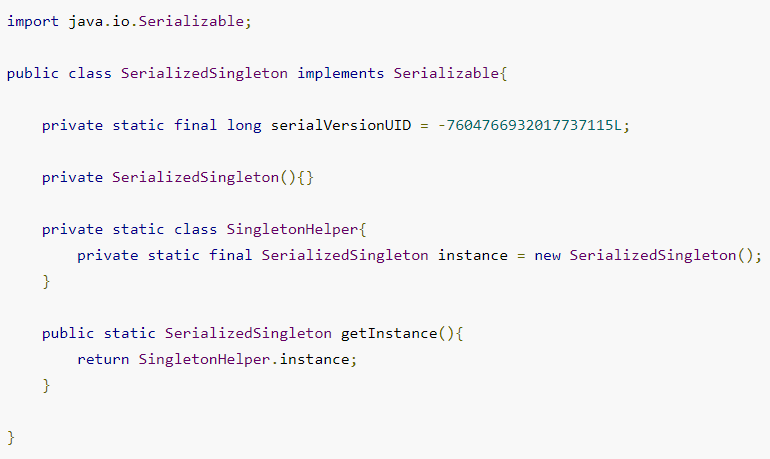
### Enum Singleton

To overcome this situation with Reflection, Joshua Bloch suggests the use of Enum to implement Singleton design pattern as Java ensures that any enum value is instantiated only once in a Java program. Since [Java Enum](https://www.journaldev.com/716/java-enum) values are globally accessible, so is the singleton. The drawback is that the enum type is somewhat inflexible; for example, it does not allow lazy initialization.



### Serialization and Singleton

Sometimes in distributed systems, we need to implement Serializable interface in Singleton class so that we can store it’s state in file system and retrieve it at later point of time. Here is a small singleton class that implements Serializable interface also.



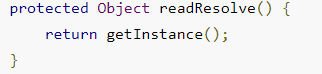
The problem with above serialized singleton class is that whenever we deserialized it, it will create a new instance of the class. Let’s see it with a simple program.



Output of the above program is:



So it destroys the singleton pattern, to overcome this scenario all we need to do it provide the implementation of readResolve() method.



After this you will notice that hashCode of both the instances are same in test program.

### Note:

* Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.
* The singleton class must provide a global access point to get the instance of the class.
* Singleton pattern is used for [logging](https://www.journaldev.com/977/logger-in-java-logging-example), drivers objects, caching and [thread pool](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice).
* Singleton design pattern is also used in other design patterns like [Abstract Factory](https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java), [Builder](https://www.journaldev.com/1425/builder-design-pattern-in-java), [Prototype](https://www.journaldev.com/1440/prototype-design-pattern-in-java), [Facade](https://www.journaldev.com/1557/facade-design-pattern-in-java) etc.
* Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

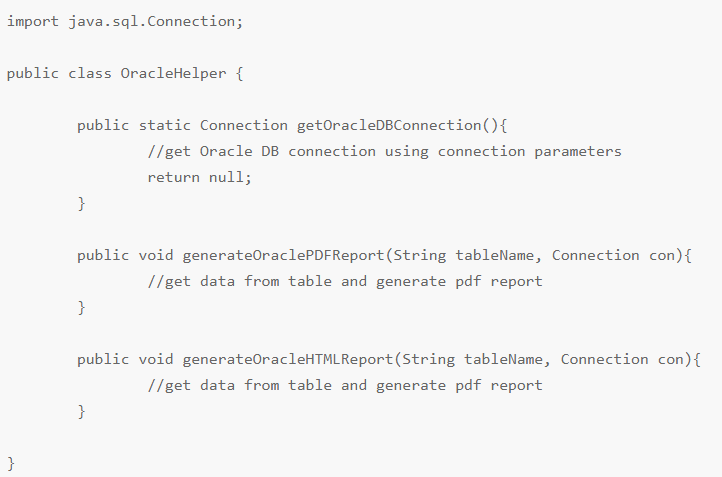
## Observer pattern:

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.

## Facade Pattern:

Facade Pattern is used to help client applications to easily interact with the system. Suppose we have an application with set of interfaces to use MySql/Oracle database and to generate different types of reports, such as HTML report, PDF report etc. So we will have different set of interfaces to work with different types of database. Now a client application can use these interfaces to get the required database connection and generate reports. But when the complexity increases or the interface behavior names are confusing, client application will find it difficult to manage it. So we can apply Facade pattern here and provide a wrapper interface on top of the existing interface to help client application.





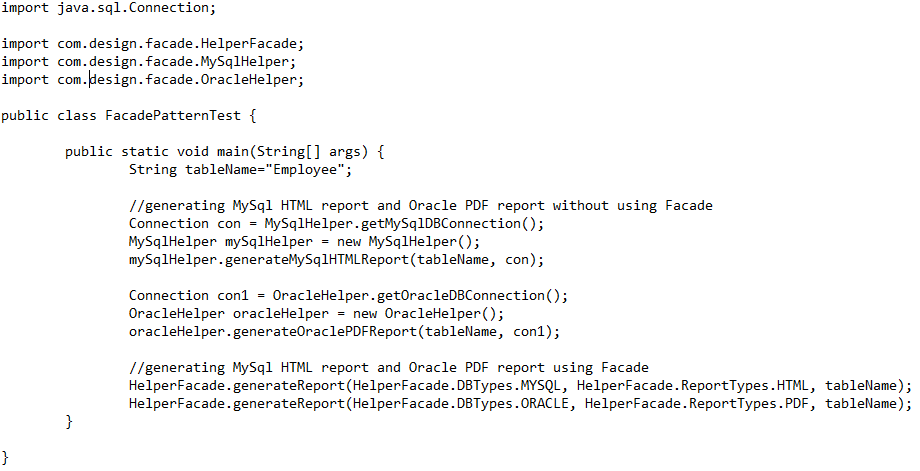
### Facade Design Pattern Interface

We can create a Facade pattern interface like below. Notice the use of [Java Enum](https://www.journaldev.com/716/java-enum) for type safety.



### Facade Design Pattern Client Program

Now let’s see client code without using Facade pattern and using Facade pattern interface.



As you can see that using Facade pattern interface is a lot easier and cleaner way to avoid having a lot of logic at client side. JDBC Driver Manager class to get the database connection is a wonderful example of facade design pattern.

### Facade Design Pattern Important Points

* Facade design pattern is more like a helper for client applications, it doesn’t hide subsystem interfaces from the client. Whether to use Facade or not is completely dependent on client code.
* Facade design pattern can be applied at any point of development, usually when the number of interfaces grow and system gets complex.
* Subsystem interfaces are not aware of Facade and they shouldn’t have any reference of the Facade interface.
* Facade design pattern should be applied for similar kind of interfaces, its purpose is to provide a single interface rather than multiple interfaces that does the similar kind of jobs.
* We can use [Factory pattern](https://www.journaldev.com/1392/factory-design-pattern-in-java) with Facade to provide better interface to client systems