Design Patterns

<https://www.geeksforgeeks.org/design-patterns-set-1-introduction/>

<https://www.geeksforgeeks.org/design-patterns-understand-the-importance-with-real-life-examples/>

A design pattern provides a general reusable solution for the common problems occurs in software design. The patterns typically show relationships and interactions between classes or objects. The idea is to speed up the development process by providing well tested, proven development/design paradigm. Design patterns are programming language independent strategies for solving a common problem. That means a design pattern represents an idea, not a particular implementation. By using the design patterns you can make your code more flexible, reusable and maintainable.

It’s not mandatory to implement design patterns in your project always. Design patterns are not meant for project development. Design patterns are meant for common problem-solving. Whenever there is a need, you have to implement a suitable pattern to avoid such problems in the future. To find out which pattern to use. You just have to try to understand the design patterns and it’s purposes. Only by then you will be able to pick the right one.

**Goal:**  
Understand the purpose and usage of each design patterns. So, you will be able to pick and implement the correct pattern as needed.

**Example:**  
For example, in many real-world situations, we want to create only one instance of a class. For example, there can be only one active president of the country at a time regardless of personal identity. This pattern is called a Singleton pattern. Other software examples could be a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.  
   
**Types of Design Patterns**  
There are mainly three types of design patterns:

1. **Creational**  
   These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Creational design patterns are the Factory Method, Abstract Factory, Builder, Singleton, Object Pool, and Prototype.

Use case of creational design pattern-  
1) Suppose a developer wants to create a simple DBConnection class to connect to a database and wants to access the database at multiple locations from code, generally what developer will do is create an instance of DBConnection class and use it for doing database operations wherever required. Which results in creating multiple connections from the database as each instance of DBConnection class will have a separate connection to the database. In order to deal with it, we create DBConnection class as a singleton class, so that only one instance of DBConnection is created and a single connection is established. Because we can manage DB Connection via one instance so we can control load balance, unnecessary connections, etc.

2) Suppose you want to create multiple instances of similar kind and want to achieve loose coupling then you can go for Factory pattern. A class implementing factory design pattern works as a bridge between multiple classes. Consider an example of using multiple database servers like SQL Server and Oracle. If you are developing an application using SQL Server database as back end, but in future need to change database to oracle, you will need to modify all your code, so as factory design patterns maintain loose coupling and easy implementation we should go for factory for achieving loose coupling and creation of similar kind of object.

1. **Structural**  
   These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.

Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy.

Use Case Of Structural Design Pattern-

1) When 2 interfaces are not compatible with each other and want to make establish a relationship between them through an adapter its called adapter design pattern. Adapter pattern converts the interface of a class into another interface or classes the client expects that is adapter lets classes works together that could not otherwise because of incompatibility. so in these type of incompatible scenarios, we can go for the adapter pattern.

1. **Behavioral**  
   Behavioral patterns are about identifying common communication patterns between objects and realize these patterns.

Behavioral patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor

Use Case of Behavioral Design Pattern-

1) Template pattern defines the skeleton of an algorithm in an operation deferring some steps to sub-classes, Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure. say for an example in your project you want the behavior of the module can be extended, such that we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. However, No one is allowed to make source code changes to it. it means you can add but can’t modify the structure in those scenarios a developer can approach template design pattern.

# Singleton Design Pattern (Creational) | Introduction

<https://www.geeksforgeeks.org/singleton-design-pattern-introduction/>

Singleton is a part of **Gang of Four design pattern** and it is categorized under **creational** design patterns. In this article, we are going to take a deeper look into the usage of the Singleton pattern. It is one of the most simple design patterns in terms of the modelling but on the other hand, this is one of the most controversial patterns in terms of complexity of usage.

Singleton pattern is a design pattern which restricts a class to instantiate its multiple objects. It is nothing but a way of defining a class. Class is defined in such a way that only one instance of the class is created in the complete execution of a program or project. It is used where only a single instance of a class is required to control the action throughout the execution. A singleton class shouldn’t have multiple instances in any case and at any cost. Singleton classes are used for logging, driver objects, caching and thread pool, database connections.

**Implementation of Singleton class**

An implementation of singleton class should have following properties:

1. **It should have only one instance :** This is done by providing an instance of the class from within the class. Outer classes or subclasses should be prevented to create the instance. This is done by making the constructor private in java so that no class can access the constructor and hence cannot instantiate it.
2. **Instance should be globally accessible :** Instance of singleton class should be globally accessible so that each class can use it. In Java, it is done by making the access-specifier of instance public.

//A singleton class should have public visibility

//so that complete application can use

public class GFG {

  //static instance of class globally accessible

  public static GFG instance = new GFG();

  private GFG() {

    // private constructor so that class

    //cannot be instantiated from outside

    //this class

  }

}

**Initialization Types of Singleton**

Singleton class can be instantiated by two methods:

1. **Early initialization :** In this method, class is initialized whether it is to be used or not. The main advantage of this method is its simplicity. You initiate the class at the time of class loading. Its drawback is that class is always initialized whether it is being used or not.
2. **Lazy initialization :** In this method, class in initialized only when it is required. It can save you from instantiating the class when you don’t need it. Generally, lazy initialization is used when we create a singleton class.

**Examples of Singleton class**

1. **java.lang.Runtime :** Java provides a class Runtime in its lang package which is singleton in nature. Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime() method.  
   An application cannot instantiate this class so multiple objects can’t be created for this class. Hence Runtime is a singleton class.
2. **java.awt.Desktop :** The Desktop class allows a Java application to launch associated applications registered on the native desktop to handle a URI or a file.  
   Supported operations include:
   * launching the user-default browser to show a specified URI;  
     launching the user-default mail client with an optional mailto URI;
   * launching a registered application to open, edit or print a specified file.
   * This class provides methods corresponding to these operations. The methods look for the associated application registered on the current platform, and launch it to handle a URI or file. If there is no associated application or the associated application fails to be launched, an exception is thrown.
   * Each operation is an action type represented by the Desktop.Action class.

This class also cannot be instantiated from application. Hence it is also a singleton class.

**Applications of Singleton classes**

There is a lot of applications of singleton pattern like cache-memory, database connection, drivers, logging. Some major of them are :-

1. **Hardware interface access:** The use of singleton depends on the requirements. Singleton classes are also used to prevent concurrent access of class. Practically singleton can be used in case external hardware resource usage limitation required e.g. Hardware printers where the print spooler can be made a singleton to avoid multiple concurrent accesses and creating deadlock.
2. **Logger :** Singleton classes are used in log file generations. Log files are created by the logger class object. Suppose an application where the logging utility has to produce one log file based on the messages received from the users. If there is multiple client application using this logging utility class they might create multiple instances of this class and it can potentially cause issues during concurrent access to the same logger file. We can use the logger utility class as a singleton and provide a global point of reference so that each user can use this utility and no 2 users access it at the same time.
3. **Configuration File:** This is another potential candidate for Singleton pattern because this has a performance benefit as it prevents multiple users to repeatedly access and read the configuration file or properties file. It creates a single instance of the configuration file which can be accessed by multiple calls concurrently as it will provide static config data loaded into in-memory objects. The application only reads from the configuration file for the first time and thereafter from second call onwards the client applications read the data from in-memory objects.
4. **Cache:** We can use the cache as a singleton object as it can have a global point of reference and for all future calls to the cache object the client application will use the in-memory object.

**Important points**

* Singleton classes can have only one instance and that instance should be globally accessible.
* java.lang.Runtime and java.awt.Desktop are 2 singleton classes provided by JVM.
* Singleton Design pattern is a type of creational design pattern.
* Outer classes should be prevented to create instance of singleton class.

# Singleton Design Pattern | Implementation

<https://www.geeksforgeeks.org/singleton-design-pattern/?ref=lbp>

The singleton pattern is one of the simplest design patterns. Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

**Definition:**  
The singleton pattern is a design pattern that restricts the instantiation of a class to one object.  
Let’s see various design options for implementing such a class. If you have a good handle on static class variables and access modifiers this should not be a difficult task.

**Method 1: Classic Implementation**

// Classical Java implementation of singleton

// design pattern

class Singleton

{

    private static Singleton obj;

    // private constructor to force use of

    // getInstance() to create Singleton object

    private Singleton() {}

    public static Singleton getInstance()

    {

        if (obj==null)

            obj = new Singleton();

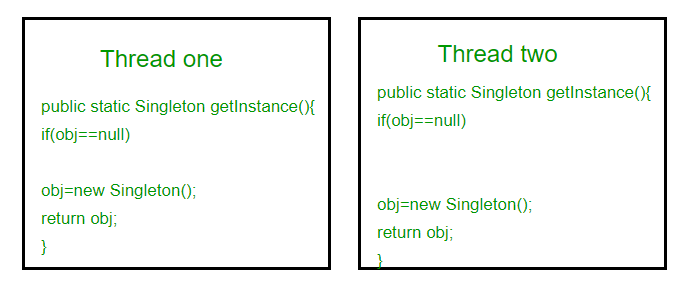
        return obj;

    }

}

Here we have declared getInstance() static so that we can call it without instantiating the class. The first time getInstance() is called it creates a new singleton object and after that it just returns the same object. Note that Singleton obj is not created until we need it and call getInstance() method. This is called lazy instantiation.

The main problem with above method is that it is not thread safe. Consider the following execution sequence.

[](https://media.geeksforgeeks.org/wp-content/uploads/singleton-1.png)

This execution sequence creates two objects for singleton. Therefore this classic implementation is not thread safe.

**Method 2: make getInstance() synchronized**

|  |
| --- |
| // Thread Synchronized Java implementation of  // singleton design pattern  class Singleton  {      private static Singleton obj;        private Singleton() {}        // Only one thread can execute this at a time      public static synchronized Singleton getInstance()      {          if (obj==null)              obj = new Singleton();          return obj;      }  } |

Here using synchronized makes sure that only one thread at a time can execute getInstance().  
The main disadvantage of this is method is that using synchronized every time while creating the singleton object is expensive and may decrease the performance of your program. However if performance of getInstance() is not critical for your application this method provides a clean and simple solution.

**Method 3: Eager Instantiation**

|  |
| --- |
| // Static initializer based Java implementation of  // singleton design pattern  class Singleton  {      private static Singleton obj = new Singleton();        private Singleton() {}        public static Singleton getInstance()      {          return obj;      }  } |

Here we have created instance of singleton in static initializer. JVM executes static initializer when the class is loaded and hence this is guaranteed to be thread safe. Use this method only when your singleton class is light and is used throughout the execution of your program.

**Method 4 (Best): Use “**[**Double Checked Locking**](https://en.wikipedia.org/wiki/Double-checked_locking)**”**

If you notice carefully once an object is created synchronization is no longer useful because now obj will not be null and any sequence of operations will lead to consistent results.  
So we will only acquire lock on the getInstance() once, when the obj is null. This way we only synchronize the first way through, just what we want.

// Double Checked Locking based Java implementation of

// singleton design pattern

class Singleton

{

    private volatile static Singleton obj;

    private Singleton() {}

    public static Singleton getInstance()

    {

        if (obj == null)

        {

            // To make thread safe

            synchronized (Singleton.class)

            {

                // check again as multiple threads

                // can reach above step

                if (obj==null)

                    obj = new Singleton();

            }

        }

        return obj;

    }

}

We have declared the obj [volatile](https://www.geeksforgeeks.org/volatile-keyword-in-java/) which ensures that multiple threads offer the obj variable correctly when it is being initialized to Singleton instance. This method drastically reduces the overhead of calling the synchronized method every time.

# Java Singleton Design Pattern Practices with Examples

<https://www.geeksforgeeks.org/java-singleton-design-pattern-practices-examples/>

In this article, we will see how we can create singleton classes. After reading this article you will be able to create your singleton class according to your use, simplicity and removed bottlenecks.  
There are many ways this can be done in Java. All these ways differs in their implementation of the pattern, but in the end, they all achieve the same end result of a single instance.

**Eager initialization:** This is the simplest method of creating a singleton class. In this, object of class is created when it is loaded to the memory by JVM. It is done by assigning the reference an instance directly.  
It can be used when program will always use instance of this class, or the cost of creating the instance is not too large in terms of resources and time.

// Java code to create singleton class by

// Eager Initialization

public class GFG

{

  // public instance initialized when loading the class

  private static final GFG instance = new GFG();

  private GFG()

  {

    // private constructor

  }

  public static GFG getInstance(){

        return instance;

    }

}

**Pros:**

1. Very simple to implement.

**Cons:**

1. May lead to resource wastage. Because instance of class is created always, whether it is required or not.
2. CPU time is also wasted in creation of instance if it is not required.
3. Exception handling is not possible.

**Using static block:** This is also a sub part of Eager initialization. The only difference is object is created in a static block so that we can have access on its creation, like exception handling. In this way also, object is created at the time of class loading.  
It can be used when there is a chance of exceptions in creating object with eager initialization.

// Java code to create singleton class

// Using Static block

public class GFG

{

  // public instance

  public static GFG instance;

  private GFG()

  {

    // private constructor

  }

static

  {

    // static block to initialize instance

    instance = new GFG();

  }

}

**Pros:**

1. Very simple to implement.
2. No need to implement getInstance() method. Instance can be accessed directly.
3. Exceptions can be handled in static block.

**Cons:**

1. May lead to resource wastage. Because instance of class is created always, whether it is required or not.
2. CPU time is also wasted in creation of instance if it is not required.

**Lazy initialization:** In this method, object is created only if it is needed. This may prevent resource wastage. An implementation of getInstance() method is required which return the instance. There is a null check that if object is not created then create, otherwise return previously created. To make sure that class cannot be instantiated in any other way, constructor is made final. As object is created with in a method, it ensures that object will not be created until and unless it is required. Instance is kept private so that no one can access it directly.  
It can be used in a single threaded environment because multiple threads can break singleton property because they can access get instance method simultaneously and create multiple objects.

//Java Code to create singleton class

// With Lazy initialization

public class GFG

{

  // private instance, so that it can be

  // accessed by only by getInstance() method

  private static GFG instance;

  private GFG()

  {

    // private constructor

  }

  //method to return instance of class

  public static GFG getInstance()

  {

    if (instance == null)

    {

      // if instance is null, initialize

      instance = new GFG();

    }

    return instance;

  }

}

**Pros:**

1. Object is created only if it is needed. It may overcome resource overcome and wastage of CPU time.
2. Exception handling is also possible in method.

**Cons:**

1. Every time a condition of null has to be checked.
2. instance can’t be accessed directly.
3. In multithreaded environment, it may break singleton property.

**Thread Safe Singleton:** A thread safe singleton in created so that singleton property is maintained even in multithreaded environment. To make a singleton class thread-safe, getInstance() method is made synchronized so that multiple threads can’t access it simultaneously.

|  |
| --- |
| // Java program to create Thread Safe  // Singleton class  public class GFG  {    // private instance, so that it can be    // accessed by only by getInstance() method    private static GFG instance;      private GFG()    {      // private constructor    }     //synchronized method to control simultaneous access    synchronized public static GFG getInstance()    {      if (instance == null)      {        // if instance is null, initialize        instance = new GFG();      }      return instance;    }  } |

**Pros:**

1. Lazy initialization is possible.
2. It is also thread safe.

**Cons:**

1. getInstance() method is synchronized so it causes slow performance as multiple threads can’t access it simultaneously.

**Lazy initialization with Double check locking:** In this mechanism, we overcome the overhead problem of synchronized code. In this method, getInstance is not synchronized but the block which creates instance is synchronized so that minimum number of threads have to wait and that’s only for first time.

|  |
| --- |
| // Java code to explain double check locking  public class GFG  {    // private instance, so that it can be    // accessed by only by getInstance() method    private static GFG instance;      private GFG()    {      // private constructor    }      public static GFG getInstance()    {      if (instance == null)      {        //synchronized block to remove overhead        synchronized (GFG.class)        {          if(instance==null)          {            // if instance is null, initialize            instance = new GFG();          }          }      }      return instance;    }  } |

**Pros:**

1. Lazy initialization is possible.
2. It is also thread safe.
3. Performance overhead gets reduced because of synchronized keyword.

**Cons:**

1. First time, it can affect performance.

As cons. of double check locking method is bearable so it can be used for high performance multi-threaded applications.

**Bill Pugh Singleton Implementation:** Prior to Java5, memory model had a lot of issues and above methods caused failure in certain scenarios in multithreaded environment. So, Bill Pugh suggested a concept of inner static classes to use for singleton.

// Java code for Bill Pugh Singleton Implementaion

public class GFG

{

  private GFG()

  {

    // private constructor

  }

  // Inner class to provide instance of class

  private static class BillPughSingleton

  {

    private static final GFG INSTANCE = new GFG();

  }

  public static GFG getInstance()

  {

    return BillPughSingleton.INSTANCE;

  }

}

When the singleton class is loaded, inner class is not loaded and hence doesn’t create object when loading the class. Inner class is created only when getInstance() method is called. So it may seem like eager initialization but it is lazy initialization.  
This is the most widely used approach as it doesn’t use synchronization.

**When to use What**

1. Eager initialization is easy to implement but it may cause resource and CPU time wastage. Use it only if cost of initializing a class is less in terms of resources or your program will always need the instance of class.
2. By using Static block in Eager initialization we can provide exception handling and also can control over instance.
3. Using synchronized we can create singleton class in multi-threading environment also but it can cause slow performance, so we can use Double check locking mechanism.
4. Bill Pugh implementation is most widely used approach for singleton classes. Most developers prefer it because of its simplicity and advantages.

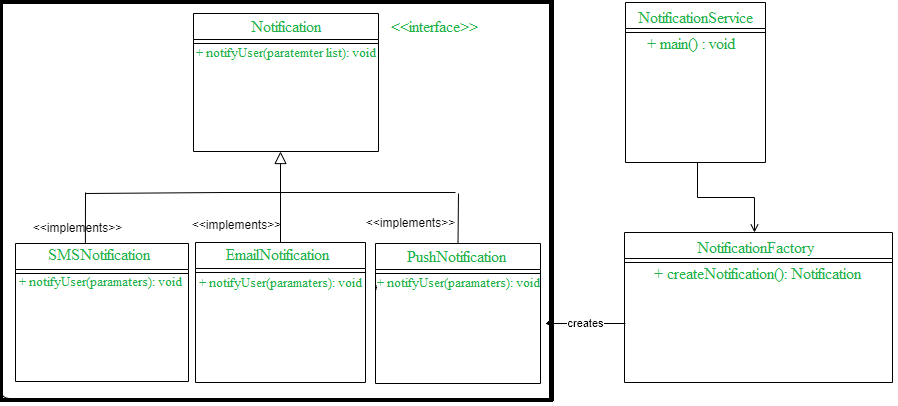
# Factory method design pattern (Creational) in Java

<https://www.geeksforgeeks.org/factory-method-design-pattern-in-java/>

It is a creational design pattern which talks about the creation of an object. The factory design pattern says that define an interface ( A java interface or an abstract class) and let the subclasses decide which object to instantiate. The factory method in the interface lets a class defer the instantiation to one or more concrete subclasses. Since this design patterns talk about instantiation of an object and so it comes under the category of creational design pattern. If we notice the name **Factory method**, that means there is a method which is a factory, and in general factories are involved with creational stuff and here with this an object is being created. It is one of the best ways to create an object where object creation logic is hidden to the client. Now Let’s look at the implementation.

**Implementation:**  
1. Define a factory method inside an interface.  
2. Let the subclass implements the above factory method and decide which object to create.

In Java constructors are not polymorphic, but by allowing subclass to create an object, we are adding polymorphic behavior to the instantiation. In short, we are trying to achieve Pseudo polymorphism by letting the subclass to decide what to create, and so this Factory method is also called as **Virtual constructor**. Let’s try to implement it with a real-time problem and some coding exercise.  
**Problem Statement :**  
Consider we want to implement a notification service through email, SMS, and push notification. Let’s try to implement this with the help of factory method design pattern. First we will design a UML class diagram for this.



In the above class diagram we have an interface called **NotificationService**, and three concrete classes are implementing NotificationService interface. A factory class NotificationFactory is created to get a NotificationService object. Let’s jump into the coding now.  
**Create Notification interface**

public interface Notification {

    void notifyUser();

}

Note- Above interface could be created as an abstract class as well.  
**Create all implementation classes**

SMSNotification.java

public class SMSNotification implements Notification {

    @Override

    public void notifyUser()

    {

        // TODO Auto-generated method stub

        System.out.println("Sending an SMS notification");

    }

}

EmailNotification.java

public class EmailNotification implements Notification {

    @Override

    public void notifyUser()

    {

        // TODO Auto-generated method stub

        System.out.println("Sending an e-mail notification");

    }

}

PushNotification.java

public class PushNotification implements Notification {

    @Override

    public void notifyUser()

    {

        // TODO Auto-generated method stub

        System.out.println("Sending a push notification");

    }

}

**Create a factory class NotificationFactory.java to instantiate concrete class.**

public class NotificationFactory {

    public Notification createNotification(String channel)

    {

        if (channel == null || channel.isEmpty())

            return null;

        if ("SMS".equals(channel)) {

            return new SMSNotification();

        }

        else if ("EMAIL".equals(channel)) {

            return new EmailNotification();

        }

        else if ("PUSH".equals(channel)) {

            return new PushNotification();

        }

        return null;

    }

}

Now let’s use factory class to create and get an object of concrete class by passing some information.

\

public class NotificationService {

    public static void main(String[] args)

    {

        NotificationFactory notificationFactory = new NotificationFactory();

        Notification notification = notificationFactory.createNotification("SMS");

        notification.notifyUser();

    }

}

**Output :** Sending an SMS notification

**Real-time examples**  
This design pattern has been widely used in JDK, such as  
1. getInstance() method of java.util.Calendar, NumberFormat, and ResourceBundle uses factory method design pattern.  
2. All the wrapper classes like Integer, Boolean etc, in Java uses this pattern to evaluate the values using valueOf() method.  
3. java.nio.charset.Charset.forName(), java.sql.DriverManager#getConnection(), java.net.URL.openConnection(), java.lang.Class.newInstance(), java.lang.Class.forName() are some of ther example where factory method design pattern has been used.

**Conclusion**  
So far we learned what is Factory method design pattern and how to implement it. I believe now we have a fair understanding of the advantage of this design mechanism.

# Prototype Design Pattern (Creational)

<https://www.geeksforgeeks.org/prototype-design-pattern/>

# Builder Design Pattern

<https://www.geeksforgeeks.org/builder-design-pattern/>

# Facade Design Pattern (Structural) | Introduction

<https://www.geeksforgeeks.org/facade-design-pattern-introduction/>

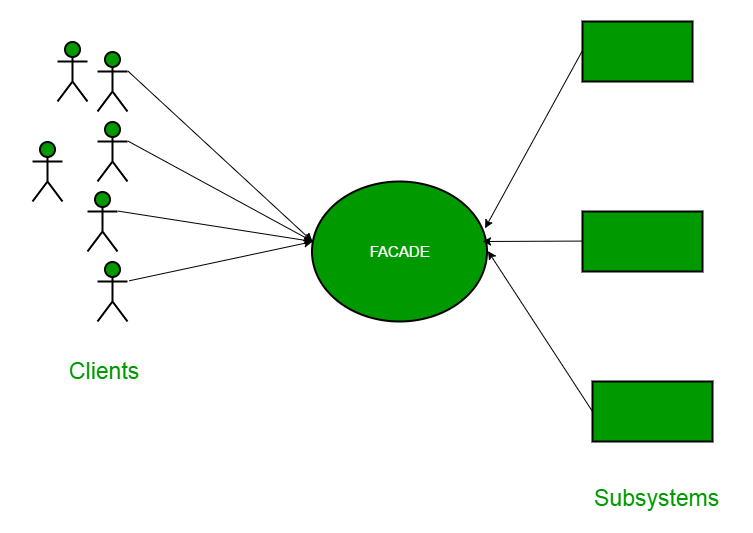
Facade is a part of Gang of Four design pattern and it is categorized under Structural design patterns. Before we dig into the details of it, let us discuss some examples which will be solved by this particular Pattern.

So, As the name suggests, it means the face of the building. The people walking past the road can only see this glass face of the building. They do not know anything about it, the wiring, the pipes and other complexities. It hides all the complexities of the building and displays a friendly face.

**More examples**

In Java, the interface JDBC can be called a facade because, we as users or clients create connection using the “java.sql.Connection” interface, the implementation of which we are not concerned about. The implementation is left to the vendor of driver.

Another good example can be the startup of a computer. When a computer starts up, it involves the work of cpu, memory, hard drive, etc. To make it easy to use for users, we can add a facade which wrap the complexity of the task, and provide one simple interface instead.  
Same goes for the **Facade Design Pattern**. It hides the complexities of the system and provides an interface to the client from where the client can access the system.

**Facade Design Pattern Diagram**

Now Let’s try and understand the facade pattern better using a simple example. Let’s consider a hotel. This hotel has a hotel keeper. There are a lot of restaurants inside hotel e.g. Veg restaurants, Non-Veg restaurants and Veg/Non Both restaurants.  
You, as client want access to different menus of different restaurants . You do not know what are the different menus they have. You just have access to hotel keeper who knows his hotel well. Whichever menu you want, you tell the hotel keeper and he takes it out of from the respective restaurants and hands it over to you. Here, the hotel keeper acts as the **facade**, as he hides the complexities of the system hotel.  
Let’s see how it works :

**Interface of Hotel**

package structural.facade;

public interface Hotel

{

    public Menus getMenus();

}

The hotel interface only returns Menus.  
Similarly, the Restaurant are of three types and can implement the hotel interface. Let’s have a look at the code for one of the Restaurants.

**NonVegRestaurant.java**

package structural.facade;

public class NonVegRestaurant implements Hotel

{

    public Menus getMenus()

    {

        NonVegMenu nv = new NonVegMenu();

        return nv;

    }

}

**VegRestaurant.java**

|  |
| --- |
| package structural.facade;    public class VegRestaurant implements Hotel  {      public Menus getMenus()      {          VegMenu v = new VegMenu();          return v;      }  } |

**VegNonBothRestaurant.java**

package structural.facade;

public class VegNonBothRestaurant implements Hotel

{

    public Menus getMenus()

    {

        Both b = new Both();

        return b;

    }

}

Now let’s consider the facade,

**HotelKeeper.java**

package structural.facade;

public class HotelKeeper

{

    public VegMenu getVegMenu()

    {

        VegRestaurant v = new VegRestaurant();

        VegMenu vegMenu = (VegMenu)v.getMenus();

        return vegMenu;

    }

    public NonVegMenu getNonVegMenu()

    {

        NonVegRestaurant v = new NonVegRestaurant();

        NonVegMenu NonvegMenu = (NonVegMenu)v.getMenus();

        return NonvegMenu;

    }

    public Both getVegNonMenu()

    {

        VegNonBothRestaurant v = new VegNonBothRestaurant();

        Both bothMenu = (Both)v.getMenus();

        return bothMenu;

    }

}

From this, It is clear that the complex implementation will be done by HotelKeeper himself. The client will just access the HotelKeeper and ask for either Veg, NonVeg or VegNon Both Restaurant menu.

**How will the client program access this façade?**

package structural.facade;

public class Client

{

    public static void main (String[] args)

    {

        HotelKeeper keeper = new HotelKeeper();

        VegMenu v = keeper.getVegMenu();

        NonVegMenu nv = keeper.getNonVegMenu();

        Both = keeper.getVegNonMenu();

    }

}

In this way the implementation is sent to the façade. The client is given just one interface and can access only that. This hides all the complexities.

**When Should this pattern be used?**

The facade pattern is appropriate when you have a **complex system**that you want to expose to clients in a simplified way, or you want to make an external communication layer over an existing system which is incompatible with the system. Facade deals with interfaces, not implementation. Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.

Adapter Pattern (**Structural**)

<https://www.geeksforgeeks.org/adapter-pattern/>

This pattern is easy to understand as the real world is full of adapters.   For example consider a USB to Ethernet adapter. We need this when we have an Ethernet interface on one end and USB on the other. Since they are incompatible with each other. we use an adapter that converts one to other. This example is pretty analogous to Object Oriented Adapters. In design, adapters are used when we have a class (Client) expecting some type of object and we have an object (Adaptee) offering the same features but exposing a different interface.

To use an adapter:

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request on the adaptee using the adaptee interface.
3. Client receive the results of the call and is unaware of adapter’s presence.

***Definition:***

The adapter pattern convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

**Class Diagram:**  


The client sees only the target interface and not the adapter. The adapter implements the target interface. Adapter delegates all requests to Adaptee.

**Example:**

Suppose you have a Bird class with fly() , and makeSound()methods. And also a ToyDuck class with squeak() method. Let’s assume that you are short on ToyDuck objects and you would like to use Bird objects in their place. Birds have some similar functionality but implement a different interface, so we can’t use them directly. So we will use adapter pattern. Here our client would be ToyDuck and adaptee would be Bird.

Below is Java implementation of it.

// Java implementation of Adapter pattern

interface Bird

{

    // birds implement Bird interface that allows

    // them to fly and make sounds adaptee interface

    public void fly();

    public void makeSound();

}

class Sparrow implements Bird

{

    // a concrete implementation of bird

    public void fly()

    {

        System.out.println("Flying");

    }

    public void makeSound()

    {

        System.out.println("Chirp Chirp");

    }

}

interface ToyDuck

{

    // target interface

    // toyducks dont fly they just make

    // squeaking sound

    public void squeak();

}

class PlasticToyDuck implements ToyDuck

{

    public void squeak()

    {

        System.out.println("Squeak");

    }

}

class BirdAdapter implements ToyDuck

{

    // You need to implement the interface your

    // client expects to use.

    Bird bird;

    public BirdAdapter(Bird bird)

    {

        // we need reference to the object we

        // are adapting

        this.bird = bird;

    }

    public void squeak()

    {

        // translate the methods appropriately

        bird.makeSound();

    }

}

class Main

{

    public static void main(String args[])

    {

        Sparrow sparrow = new Sparrow();

        ToyDuck toyDuck = new PlasticToyDuck();

        // Wrap a bird in a birdAdapter so that it

        // behaves like toy duck

        ToyDuck birdAdapter = new BirdAdapter(sparrow);

        System.out.println("Sparrow...");

        sparrow.fly();

        sparrow.makeSound();

        System.out.println("ToyDuck...");

        toyDuck.squeak();

        // toy duck behaving like a bird

        System.out.println("BirdAdapter...");

        birdAdapter.squeak();

    }

}

Output:

Sparrow...

Flying

Chirp Chirp

ToyDuck...

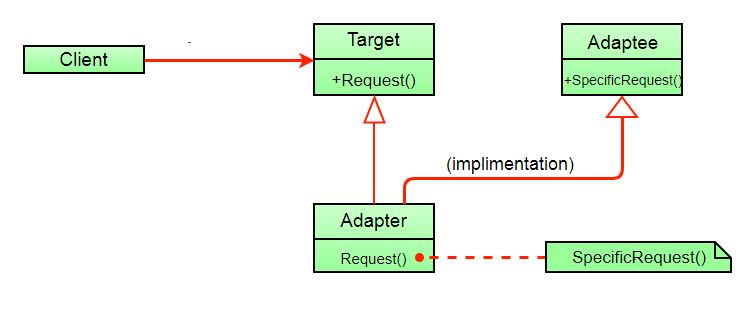
Squeak

BirdAdapter...

Chirp Chirp

**Explanation :**  
Suppose we have a bird that can makeSound(), and we have a plastic toy duck that can squeak(). Now suppose our client changes the requirement and he wants the toyDuck to makeSound than ?  
Simple solution is that we will just change the implementation class to the new adapter class and tell the client to pass the instance of the bird(which wants to squeak()) to that class.  
**Before :** ToyDuck toyDuck = new PlasticToyDuck();  
**After :** ToyDuck toyDuck = new BirdAdapter(sparrow);  
You can see that by changing just one line the toyDuck can now do Chirp Chirp !!

**Object Adapter Vs Class Adapter**  
The adapter pattern we have implemented above is called Object Adapter Pattern because the adapter holds an instance of adaptee. There is also another type called Class Adapter Pattern which use inheritance instead of composition but you require multiple inheritance to implement it.  
Class diagram of Class Adapter Pattern:



Here instead of having an adaptee object inside adapter (composition) to make use of its functionality adapter inherits the adaptee.

Since multiple inheritance is not supported by many languages including java and is associated with many problems we have not shown implementation using class adapter pattern.

**Advantages:**

* Helps achieve reusability and flexibility.
* Client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.

**Disadvantages:**

* All requests are forwarded, so there is a slight increase in the overhead.
* Sometimes many adaptations are required along an adapter chain to reach the type which is required.

# Bridge Design Pattern (Structural)

<https://www.geeksforgeeks.org/bridge-design-pattern/>

# Composite Design Pattern (Structural)

<https://www.geeksforgeeks.org/composite-design-pattern/>

# Decorator Pattern (Structural) | Set 1 (Background)

<https://www.geeksforgeeks.org/decorator-pattern/>

# Decorator Pattern (Structural) | Set 2 (Introduction and Design)

<https://www.geeksforgeeks.org/the-decorator-pattern-set-2-introduction-and-design/>

# Decorator Pattern (Structural) | Set 3 (Coding the Design)

<https://www.geeksforgeeks.org/decorator-pattern-set-3-coding-the-design/>

# Proxy Design Pattern (Structural)

<https://www.geeksforgeeks.org/proxy-design-pattern/>

# Interpreter Design Pattern (Behavioral)

<https://www.geeksforgeeks.org/interpreter-design-pattern/>

# Strategy Pattern (Behavioral)| Set 1 (Introduction)

<https://www.geeksforgeeks.org/strategy-pattern-set-1/>

# Strategy Pattern (Behavioral)| Set 2 (Implementation)

<https://www.geeksforgeeks.org/strategy-pattern-set-2/>

# State Design Pattern (Behavioral)

<https://www.geeksforgeeks.org/state-design-pattern/>

# Command Pattern (Behavioral)

<https://www.geeksforgeeks.org/command-pattern/>

# Chain of Responsibility Design Pattern (Behavioral)

<https://www.geeksforgeeks.org/chain-responsibility-design-pattern/>

# Memento design pattern (Behavioral)

<https://www.geeksforgeeks.org/memento-design-pattern/>

# Template Method Design Pattern (Behavioral)

<https://www.geeksforgeeks.org/template-method-design-pattern/>

# Observer Pattern (Behavioral)| Set 1 (Introduction)

<https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/>

# Observer Pattern | Set 2 (Implementation)

<https://www.geeksforgeeks.org/observer-pattern-set-2-implementation/>

# Visitor design pattern (Behavioral)

<https://www.geeksforgeeks.org/visitor-design-pattern/>