**Design Patterns**

1. Singleton Design Pattern

Singleton Pattern says that just **"define a class that has only one instance and provides a global point of access to it".**

In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.

There are two forms of singleton design pattern

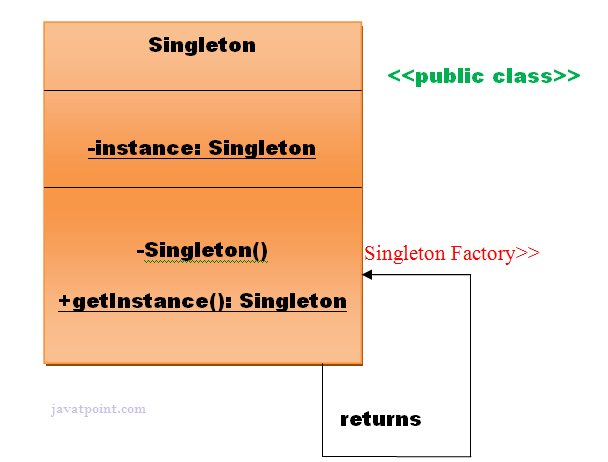
* **Early Instantiation:** creation of instance at load time.
* **Lazy Instantiation:** creation of instance when required.

Advantage of Singleton design pattern

* Saves memory because object is not created at each request. Only single instance is reused again and again.

Usage of Singleton design pattern

* Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

Class Diagram

**Examples**

### **Understanding early Instantiation of Singleton Pattern**

In such case, we create the instance of the class at the time of declaring the static data member, so instance of the class is created at the time of classloading.

Let's see the example of singleton design pattern using early instantiation.

*File: A.java*

1. **class** A{
2. **private** **static** A obj=**new** A();//Early, instance will be created at load time
3. **private** A(){}
4. **public** **static** A getA(){
5. **return** obj;
6. }
7. **public** **void** doSomething(){
8. //write your code
9. }
10. }

### **Understanding lazy Instantiation of Singleton Pattern**

In such case, we create the instance of the class in synchronized method or synchronized block, so instance of the class is created when required.

Let's see the simple example of singleton design pattern using lazy instantiation.

*File: A.java*

1. **class** A{
2. **private** **static** A obj;
3. **private** A(){}
4. **public** **static** A getA(){
5. **if** (obj == **null**){
6. **synchronized**(Singleton.**class**){
7. **if** (obj == **null**){
8. obj = **new** Singleton();//instance will be created at request time
9. } } }
10. **return** obj;
11. }
12. **public** **void** doSomething(){
13. //write your code
14. }
15. }

2) Adapter Design Pattern

An Adapter Pattern says that just **"converts the interface of a class into another interface that a client wants".**

In other words, to provide the interface according to client requirement while using the services of a class with a different interface.

The Adapter Pattern is also known as **Wrapper.**

#### Advantage of Adapter Pattern

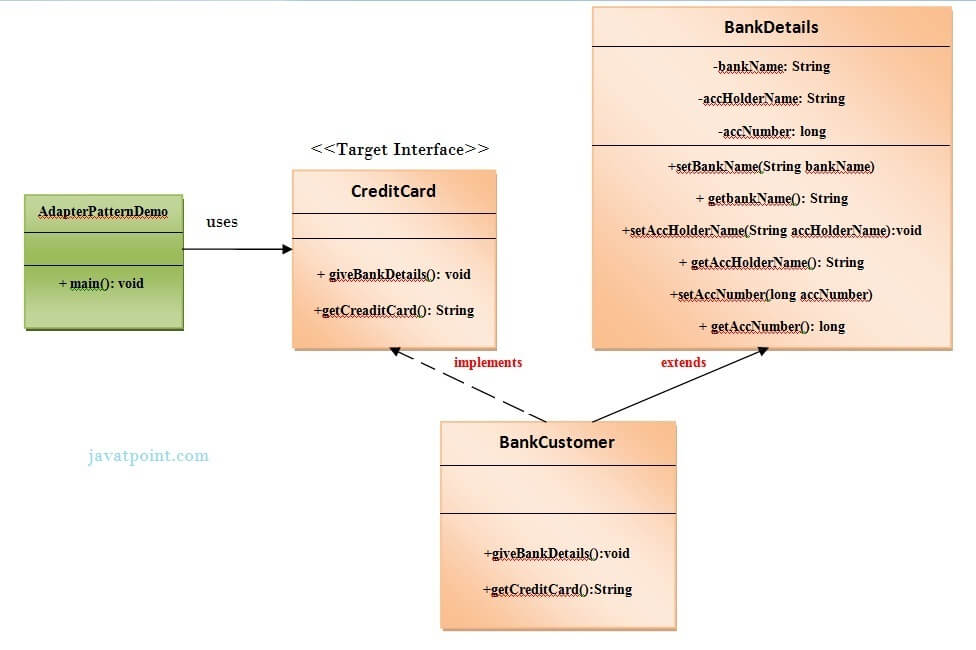
* It allows two or more previously incompatible objects to interact.
* It allows reusability of existing functionality.

#### Usage of Adapter pattern:

It is used:

* When an object needs to utilize an existing class with an incompatible interface.
* When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.
* When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.

Class Diagram



* 1. Factory Method Design Pattern

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

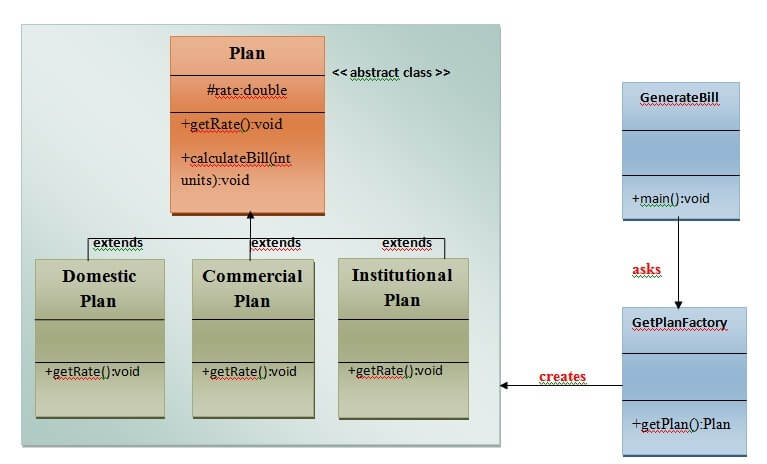
The Factory Method Pattern is also known as **Virtual Constructor.**

Advantage of Factory Design Pattern

* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

Usage of Factory Design Pattern

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.



* 1. Interpreter Design Pattern

An Interpreter Pattern says that **"to define a representation of grammar of a given language, along with an interpreter that uses this representation to interpret sentences in the language".**

Basically the Interpreter pattern has limited area where it can be applied. We can discuss the Interpreter pattern only in terms of formal grammars but in this area there are better solutions that is why it is not frequently used.

This pattern can applied for parsing the expressions defined in simple grammars and sometimes in simple rule engines.

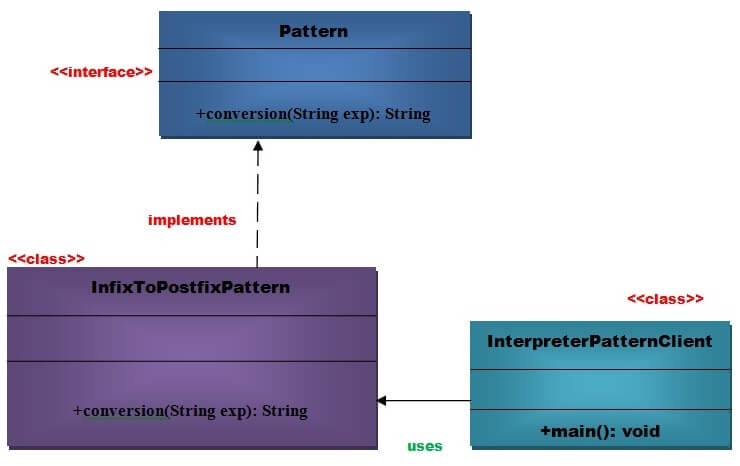
Advantage of Interpreter Pattern

* It is easier to change and extend the grammar.
* Implementing the grammar is straightforward.

Usage of Interpreter pattern:

It is used:

* When the grammar of the language is not complicated.
* When the efficiency is not a priority.



* 1. Command Pattern

A Command Pattern says that "**encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command**".

It is also known as **Action or Transaction.**

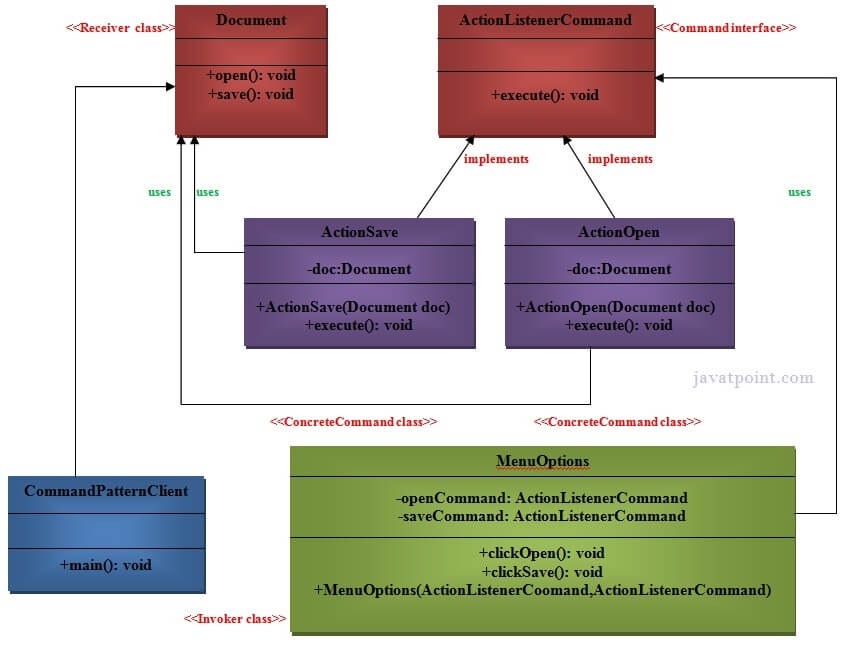
Advantage of command pattern

* It separates the object that invokes the operation from the object that actually performs the operation.
* It makes easy to add new commands, because existing classes remain unchanged.

Usage of command pattern:

It is used:

* When you need parameterize objects according to an action perform.
* When you need to create and execute requests at different times.
* When you need to support rollback, logging or transaction functionality.



* 1. Strategy Design Pattern

In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

Implementation

We are going to create a *Strategy* interface defining an action and concrete strategy classes implementing the *Strategy* interface. *Context* is a class which uses a Strategy.

*StrategyPatternDemo*, our demo class, will use *Context* and strategy objects to demonstrate change in Context behaviour based on strategy it deploys or uses.

