**CREATIONAL**

**Singleton design pattern :**

1. Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.
2. The singleton class must provide a global access point to get the instance of the class.
3. 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).
4. 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.
5. Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

**Implementation :**

1. Private constructor to restrict instantiation of the class from other classes.
2. Private static variable of the same class that is the only instance of the class.
3. 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.

**Factory Pattern**

1. we create object without exposing the creation logic to the client and refer to newly created object using a common interface.
2. Factory pattern provides abstraction between implementation and client classes through inheritance.
3. java.util.Calendar, ResourceBundle and NumberFormat getInstance() methods uses Factory pattern
4. valueOf() method in wrapper classes like Boolean, Integer etc.
5. Now days, If we create any software then it would be in future the requirement change. So, that time we need to write whole logic again from scratch.  The main concern when we work with Software Architect how to create object of entity and pass to some other object without depending to others. One thing we need to consider, architecture should be liked this so it should be pluggable, So that in future more object can be added.

**WHAT FACTORY DESIGN PATTERN IS**

1. **The Factory Design Pattern** is commonly used design pattern where we need to create **Loosely Coupled System**.  Basically, it comes under **Creational Pattern** and it is used to create instance and reuse it. Factory Pattern is based on real time Factory concept. As we know Factory is used to manufacture something as per requirement and if new item is going to add to manufacture with Factory. Factory will also manufacture those items as well.**Factory class provides abstraction** between client and Car when create the instance of the Car [Honda, BMW etc].

**WHEN TO USE FACTORY DESIGN PATTERN**

1. It is used for creating objects to encapsulate the instantiation logic.
2. Client doesn’t know actual instantiation logic of entity.

<http://www.mukeshkumar.net/articles/designpattern/factory-design-pattern-real-world-example>

# **Abstract Factory Pattern**

1. Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories.
2. 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.
3. This Abstract factory class returns different subclasses based on the input provided and factory class uses if-else or switch statement to achieve this.

<https://stackoverflow.com/questions/328496/when-would-you-use-the-builder-pattern>

<https://betterprogramming.pub/understanding-the-builder-design-pattern-f4f56fa18c9>

Decorator pattern:

1. Decorator pattern allows a user to add new functionality to an existing object without altering its structure.
2. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

<https://stackoverflow.com/questions/2707401/understand-the-decorator-pattern-with-a-real-world-example>

Proxy Pattern:

1. Proxy pattern is used when we need to create a wrapper to cover the main object’s complexity from the client.
2. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.
3. Open/Closed Principle. You can introduce new proxies without changing the service or clients.

Uses =>

1. **Lazy initialization (virtual proxy). This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time. -** Instead of creating the object when the app launches, you can delay the object’s initialization to a time when it’s really needed.
2. **Access control (protection proxy). This is when you want only specific clients to be able to use the service object; for instance, when your objects are crucial parts of an operating system and clients are various launched applications (including malicious ones).-**  The proxy can pass the request to the service object only if the client’s credentials match some criteria.
3. **Local execution of a remote service (remote proxy). This is when the service object is located on a remote server. -** In this case, the proxy passes the client request over the network, handling all of the nasty details of working with the network.

[Proxy Design Pattern - GeeksforGeeks](https://www.geeksforgeeks.org/proxy-design-pattern/)

Façade :

1. Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

Strategy:

1. 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.

**Usage examples:** The Strategy pattern is very common in Java code. It’s often used in various frameworks to provide users a way to change the behavior of a class without extending it.

Java 8 brought the support of lambda functions, which can serve as simpler alternatives to the Strategy pattern.

Here some examples of Strategy in core Java libraries:

* [**java.util.Comparator#compare()**](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#compare-T-T-) called from Collections#sort().
* [**javax.servlet.http.HttpServlet**](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServlet.html): service() method, plus all of the doXXX() methods that accept HttpServletRequest and HttpServletResponse objects as arguments.
* [**javax.servlet.Filter#doFilter()**](http://docs.oracle.com/javaee/7/api/javax/servlet/Filter.html#doFilter-javax.servlet.ServletRequest-javax.servlet.ServletResponse-javax.servlet.FilterChain-)

State:

1. In State pattern a class behavior changes based on its state. This type of design pattern comes under behavior pattern. In State pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.
2. The main idea is that, at any given moment, there’s a finite number of states which a program can be in. Within any unique state, the program behaves differently, and the program can be switched from one state to another instantaneously. However, depending on a current state, the program may or may not switch to certain other states. These switching rules, called transitions, are also finite and predetermined.

Iterator:

* 1. This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

**Usage examples:** The pattern is very common in Java code. Many frameworks and libraries use it to provide a standard way for traversing their collections.

Here are some examples from core Java libraries:

* All implementations of **[java.util.Iterator](http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html)** (also **[java.util.Scanner](http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html)**).
* All implementations of **[java.util.Enumeration](http://docs.oracle.com/javase/8/docs/api/java/util/Enumeration.html)**

**Identification:** Iterator is easy to recognize by the navigation methods (such as next, previous and others). Client code that uses iterators might not have direct access to the collection being traversed.