**Design Patterns : A design pattern** provides a general reusable solution for the common problems that occur in software design.The pattern shows 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 their purposes.

**Goal:**  
Understand the purpose and usage of each design pattern. 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 will 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](https://www.geeksforgeeks.org/coupling-in-java/) 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.

**2.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 establish a relationship between them through an adapter it’s called an adapter design pattern. Adapter pattern converts the interface of a class into another interface or class that the client expects, i.e 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.

**3. 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. For example in your project you want the behavior of the module to be able to extend, 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, i.e you can add but can’t modify the structure in those scenarios a developer can approach template design pattern.

**Singleton Design Pattern :** 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.

**Method 1: Classic Implementation :**

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)

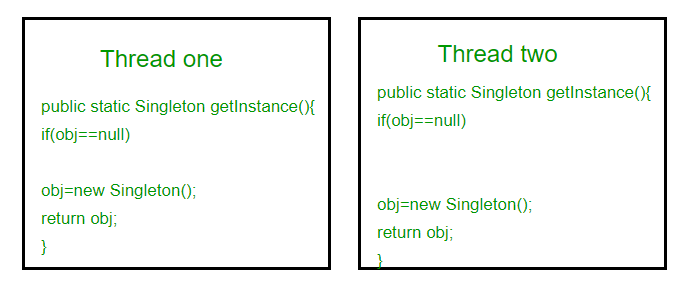
            obj = new Singleton();

        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.



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

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

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**

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" \t "https://www.geeksforgeeks.org/singleton-design-pattern/_blank)”**  
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.

class Singleton{

    private static volatile Singleton obj  = null;

    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.

**Singleton Class in Java :** In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time.

**To design a singleton class:**

1. Make constructor as private.
2. Write a static method that has return type object of this singleton class. Here, the concept of [Lazy initialization](https://en.wikipedia.org/wiki/Lazy_initialization) is used to write this static method.

class Singleton {

    // static variable single\_instance of type Singleton

    private static Singleton single\_instance = null;

    // variable of type String

    public String s;

    // private constructor restricted to this class itself

    private Singleton()     {

        s = "Hello I am a string part of Singleton class";

    }

      // static method to create instance of Singleton class

    public static Singleton getInstance() {

        if (single\_instance == null)

            single\_instance = new Singleton();

        return single\_instance;

    }

}

  // Driver Class

class Main {

    public static void main(String args[]) {

        // instantiating Singleton class with variable x

        Singleton x = Singleton.getInstance();

        // instantiating Singleton class with variable y

        Singleton y = Singleton.getInstance();

        // instantiating Singleton class with variable z

        Singleton z = Singleton.getInstance();

        // changing variable of instance x

        x.s = (x.s).toUpperCase();

        System.out.println("String from x is " + x.s);

        System.out.println("String from y is " + y.s);

        System.out.println("String from z is " + z.s);

        System.out.println("\n");

        // changing variable of instance z

        z.s = (z.s).toLowerCase();

        System.out.println("String from x is " + x.s);

        System.out.println("String from y is " + y.s);

        System.out.println("String from z is " + z.s);

    }

}

**Singleton** is a part of Gang of Four design pattern and it is categorized under**creational design patterns**.

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.

**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 :**

java.lang.Runtime : Java provides a class Runtime in its lang package which is singleton in nature.

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.

Applications of Singleton classes

There is a lot of applications of singleton pattern like cache-memory, database connection, drivers, logging.

**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.

**how we can implement singletons in Java.**

> create a private constructor that restricts to create an object outside of the class

> create a private attribute that refers to the singleton object.

> create a public static method that allows us to create and access the object we created. Inside the method, we will create a condition that restricts us from creating more than one object.

<https://codepumpkin.com/preventing-cloning-in-singleton-design-pattern/>

**Template Method Design Pattern :** Template method design pattern is to define an algorithm as a skeleton of operations and leave the details to be implemented by the child classes. The overall structure and sequence of the algorithm is preserved by the parent class.

Template means Preset format like HTML templates which has a fixed preset format. Similarly in the template method pattern, we have a preset structure method called template method which consists of steps. This steps can be an abstract method which will be implemented by its subclasses.

This pattern comes under **behavior pattern category**. This design pattern is used popularly in framework development. This helps to avoid code duplication also.

* **AbstractClass**contains the templateMethod() which should be made final so that it cannot be overridden. This template method makes use of other operations available in order to run the algorithm but is decoupled for the actual implementation of these methods. All operations used by this template method are made abstract, so their implementation is deferred to subclasses.
* **ConcreteClass**implements all the operations required by the templateMethod that were defined as abstract in the parent class. There can be many different ConcreteClasses.



We are going to create a Game abstract class defining operations with a template method set to be final so that it cannot be overridden. Cricket and Football are concrete classes that extend Game and override its methods.

TemplatePatternDemo, our demo class, will use Game to demonstrate use of template pattern.

## **Step 1 :** Create an abstract class with a template method being final.

**Game.java:**

public abstract class Game {

abstract void initialize();

abstract void startPlay();

abstract void endPlay();

**//template method**

public final void play(){

//initialize the game

initialize();

//start game

startPlay();

//end game

endPlay();

}

}

**Step 2 :**Create concrete classes extending the above class.

**Cricket.java**

public class Cricket extends Game {

@Override

void endPlay() {

System.out.println("Cricket Game Finished!");

}

@Override

void initialize() {

System.out.println("Cricket Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Cricket Game Started. Enjoy the game!");

}

}

**Football.java**

public class Football extends Game {

@Override

void endPlay() {

System.out.println("Football Game Finished!");

}

@Override

void initialize() {

System.out.println("Football Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Football Game Started. Enjoy the game!");

}

}

**Step 3 :**Use the Game's template method play() to demonstrate a defined way of playing game.

**TemplatePatternDemo.java**

public class TemplatePatternDemo {

public static void main(String[] args) {

Game game = new Cricket();

game.play();

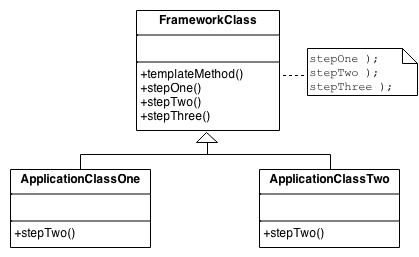
System.out.println();

game = new Football();

game.play();

}

}



**When to use template method :**The template method is used in frameworks, where each implements the invariant parts of a domain’s architecture, leaving “placeholders” for customization options.

The template method is used for the following reasons :

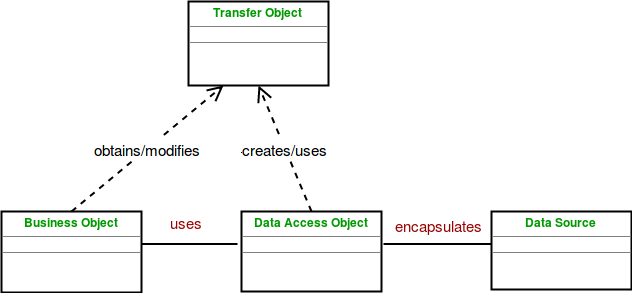
* Let subclasses implement varying behavior (through method overriding)
* Avoid duplication in the code, the general workflow structure is implemented once in the abstract class’s algorithm, and necessary variations are implemented in the subclasses.
* Control at what points subclassing is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change.

**Data Access Object Pattern :** Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services.

Java DAO design pattern is a part of layered architecture and single Responsibility problem where dao should only handle the DB related stuff.

It can not be classified under java design pattern i.e Creational, Structural or Behavioral

Following are the participants in Data Access Object Pattern.



**BusinessObject :** The BusinessObject represents the data client. It is the object that requires access to the data source to obtain and store data. A BusinessObject may be implemented as a session bean, entity bean or some other Java object in addition to a servlet or helper bean that accesses the data source.

**DataAccessObject :** The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source.

Data Access Object Interface - This interface defines the standard operations to be performed on a model object(s).

Data Access Object concrete class - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.

**DataSource :** This represents a data source implementation. A data source could be a database such as an RDBMS, OODBMS, XML repository, flat file system, and so forth. A data source can also be another system service or some kind of repository.**Model Object or Value Object -** This object is simple POJO containing get/set methods to store data retrieved using DAO class.

**TransferObject :** This represents a Transfer Object used as a data carrier. The DataAccessObject may use a Transfer Object to return data to the client. The DataAccessObject may also receive the data from the client in a Transfer Object to update the data in the data source.

**Let’s see an example of Data Access Object Pattern.**

class **Developer**{

    private String name;

    private int DeveloperId;

    Developer(String name, int DeveloperId)    {

        this.name = name;

        this.DeveloperId = DeveloperId;

}

// setter & getter

}

**interface DeveloperDao**

{

    public List<Developer> getAllDevelopers();

    public Developer getDeveloper(int DeveloperId);

    public void updateDeveloper(Developer Developer);

    public void deleteDeveloper(Developer Developer);

}

class **DeveloperDaoImpl** implements DeveloperDao {

// implement of interface method

….….

}

class **DaoPatternDemo**{

    public static void main(String[] args)     {

        DeveloperDao developerDao = new DeveloperDaoImpl();

// call the implementation method

}

**Advantages :**

* The advantage of using data access objects is the relatively simple and rigorous separation between two important parts of an application that can but should not know anything of each other, and which can be expected to evolve frequently and independently.
* if we need to change the underlying persistence mechanism we only have to change the DAO layer, and not all the places in the domain logic where the DAO layer is used from.

**Disadvantages :**

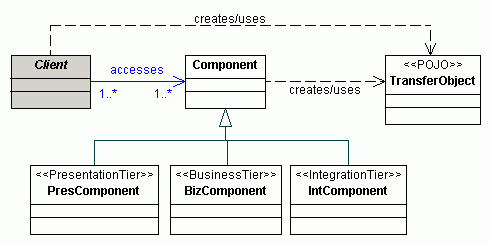
* Potential disadvantages of using DAO is leaky abstraction, code duplication, and abstraction inversion.

**Transfer Object Pattern :** It is one of the **[Java EE design patterns](https://www.dineshonjava.com/core-j2ee-patterns-best-design-practices/)**. We need ****Transfer Object**** when we need to pass the data across various attributes in a packet to the server. ****Value Object**** is another name for transfer object. The ****transfer object**** is just a class of POJO which has a method of the getter and setter. It is serializable which means we can transfer it through the network.

It is used when we want to pass data with multiple attributes in one shot from client to server. Transfer Object is a simple [POJO](https://www.geeksforgeeks.org/pojo-vs-java-beans/) class having getter/setter methods and is serialized so that it can be transferred over the network. Server Side business class normally fetches data from the database and fills the POJO and sends it to the client or passes it by value. For clients, the transfer object is read-only. The client can create its own transfer object and pass it to the server to update values in thedatabase in one shot.

The transfer object does not acquire any behavior. The class known as server-side business is generally responsible for fetching data and filling the POJO then, sending it to the respective client or passing it the value. Client objects, the object is in read-only mode. The clients are capable for creating their own transfer objects. They can even pass object to servers in order to update the values in databases in one go.

**UML Class Diagram of the Transfer Object :**



It requires certain components to implement the transfer object design pattern. Each of these components has certain tasks to perform and certain problems to cater. These components include client, business object, and transfer object. The client object is responsible for representing the client of the enterprise bean.

* The **client**can be the end user of the app.Either requests or sends the Transfer Object to Business Object
* The **business object** creates a transfer object and returns it to the respective client, when or if requested. The business object receives the data from the client in the form of a transfer object. It also uses this data in order to perform an update.Fills the Transfer Object with data
* The **transfer object** is, in fact, a Javascript. The class known as Transfer object is able to provide a constructor which may accept all the needed attributes in order to create the transfer object. The constructor also accepts the entire entity bean attribute values that a Transfer object holds. Simple POJO having methods to set/get attributes only

**Sample Implementation of the Transfer Object :Approach:**

Step 1: Create a Transfer Object

Step 2: Create a Business Object.

Step 3: Use the StudentBO to demonstrate Transfer Object Design Pattern

Step 4: Verify the output.

**Step 1:** Let’s create Transfer Object. : create a pojo class with getter & setter method

public class EmployeeVO {

private String name;

private int empNo;

public EmployeeVO(String name, int empNo) {

super();

this.name = name;

this.empNo = empNo;

}// getter & setter method

### **Step 2:** Let’s create Business Object.

public class EmployeeBO {

List employees;

public EmployeeBO(){

employees = new ArrayList<>();

EmployeeVO employee1 = new EmployeeVO("Dinesh",0);

EmployeeVO employee2 = new EmployeeVO("Arnav",1);

employees.add(employee1);

employees.add(employee2);

}

public void deleteEmployee(EmployeeVO employee) {

employees.remove(employee.getEmpNo());

System.out.println("Employee: Roll No " + employee.getEmpNo() + ", deleted from database");

}

### **Step 3:** Let’s create a demo class and use the EmployeeBO to demonstrate Transfer Object Design Pattern.

public class TransferObjectPatternDemo {

public static void main(String[] args) {

EmployeeBO EmployeeBusinessObject = new EmployeeBO();

//print all Employees

for (EmployeeVO employee : EmployeeBusinessObject.getAllEmployees()) {

System.out.println("Employee: [Emp No : " + employee.getEmpNo() + ", Name : " + employee.getName() + " ]");

}

//update Employee

EmployeeVO employee = EmployeeBusinessObject.getAllEmployees().get(0);

employee.setName("Anamika");

EmployeeBusinessObject.updateEmployee(employee);

//get the Employee

employee = EmployeeBusinessObject.getEmployee(0);

System.out.println("Employee: [Emp No : " + employee.getEmpNo() + ", Name : " + employee.getName() + " ]");

}

}

**Factory Method Pattern design :** 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.

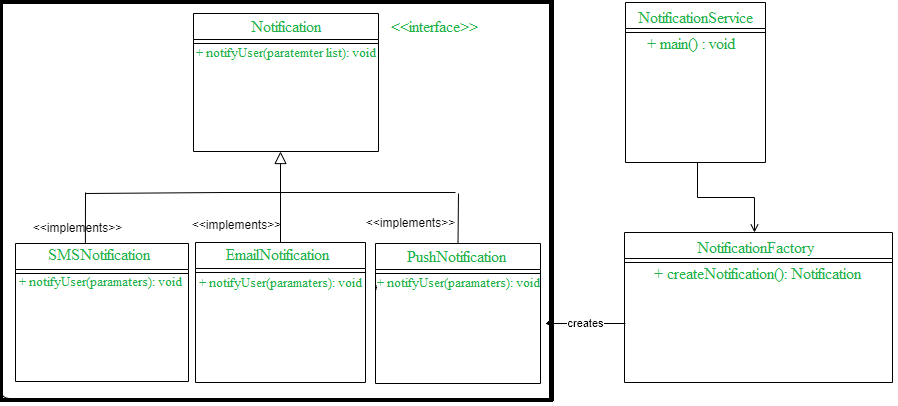
This design patterns talk about **instantiation of an object** and so it comes under the category of creational design pattern.  It is one of the best ways to create an object where object creation logic is hidden to the client.

**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.

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

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.



we have an interface called **Notification**, and three concrete classes are implementing Notification interface. A factory class **NotificationFactory** is created to get a Notification object.

**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();

    }

}

**Real-time examples**This design pattern has been widely used in JDK, such as 

* getInstance() method of java.util.Calendar, NumberFormat, and ResourceBundle uses factory method design pattern.
* All the wrapper classes like Integer, Boolean etc, in Java uses this pattern to evaluate the values using valueOf() method.
* 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.

**Abstract factory method design pattern :**

**Building design pattern:** The **[builder pattern](https://en.wikipedia.org/wiki/Builder_pattern" \o "Builder Pattern)** is an way to construct complex objects. This should be used only when you want to build different immutable objects using same object.

The **only big difference** between this builder pattern and abstract factory pattern is that, builder provides you more control over the object creation process . In one sentence, abstract factory pattern is the answer to "WHAT" and the builder pattern to "HOW".

A **builder pattern** is more like fluent interface. A fluent interface is normally implemented by using method cascading (or method chaining).

Builder pattern aims to “Separate the construction of a complex object from its representation so that the same construction process can create different representations.”

**Builder pattern** helps us in creating immutable classes with large set of state attributes.

Let’s discuss a common problem in our application. In any user management module, primary entity is User, let’s say. once a user object is fully created, you will not want to change it’s state. Now, let’s assume, our User object has following 5 attributes i.e. firstName, lastName, age, phone and address.

In normal practice, if you want to make a immutable User class, then you must pass all five information as parameters to constructor.

public User (String firstName, String lastName, int age, String phone, String address){

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.phone = phone;

this.address = address;

}

Now what if only firstName and lastName are mandatory and rest 3 fields are optional. Problem !! We need more constructors.

Now let’s introduce our sixth attribute i.e. salary. Now it is problem.

One way it to create more constructors, and another is to loose the immutability and introduce setter methods.

Builder pattern will help you to consume additional attributes while retaining the immutability of Use class.

public class User

{

//All final attributes

private final String firstName; // required

private final String lastName; // required

private final int age; // optional

private final String phone; // optional

private final String address; // optional

private User(UserBuilder builder) {

this.firstName = builder.firstName;

this.lastName = builder.lastName;

this.age = builder.age;

this.phone = builder.phone;

this.address = builder.address;

}

//All getter, and NO setter to provde immutability

@Override

public String toString() {

return "User: "+this.firstName+", "+this.lastName+", "+this.age+", "+this.phone+", "+this.address;

}

public static class UserBuilder {

private final String firstName;

private final String lastName;

private int age;

private String phone;

private String address;

public UserBuilder(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

public UserBuilder age(int age) {

this.age = age;

return this;

}

public UserBuilder phone(String phone) {

this.phone = phone;

return this;

}

public UserBuilder address(String address) {

this.address = address;

return this;

}

//Return the finally consrcuted User object

public User build() {

User user = new User(this);

validateUserObject(user);

return user;

}

private void validateUserObject(User user) {

//Do some basic validations to check

//if user object does not break any assumption of system

}

}

}

public static void main(String[] args) {

User user1 = new User.UserBuilder("Lokesh", "Gupta")

.age(30)

.phone("1234567")

.address("Fake address 1234")

.build();

System.out.println(user1);

User user2 = new User.UserBuilder("Jack", "Reacher")

.age(40)

.phone("5655")

//no address

.build();

System.out.println(user2);

User user3 = new User.UserBuilder("Super", "Man")

//No age

//No phone

//no address

.build();

System.out.println(user3);

}

Output:

User: Lokesh, Gupta, 30, 1234567, Fake address 1234

User: Jack, Reacher, 40, 5655, null

User: Super, Man, 0, null, null

Please note that above created user object does not have any setter method, so it’s state can not be changed once it has been built. This provides the desired immutability.

**Existing implementations in JDK**

* All implementations of [java.lang.Appendable](https://docs.oracle.com/javase/7/docs/api/java/lang/Appendable.html" \o "Appendable) are infact good example of use of Builder pattern in java. e.g.
* [java.lang.](https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html" \l "append(java.lang.CharSequence)" \o "StringBuilder)**[StringBuilder](https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html" \l "append(java.lang.CharSequence)" \o "StringBuilder)**[#append()](https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html" \l "append(java.lang.CharSequence)" \o "StringBuilder) [Unsynchronized class]
* [java.lang.](https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/StringBuffer.html" \l "append(java.lang.CharSequence)" \o "StringBuffer)**[StringBuffer](https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/StringBuffer.html" \l "append(java.lang.CharSequence)" \o "StringBuffer)**[#append()](https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/StringBuffer.html" \l "append(java.lang.CharSequence)" \o "StringBuffer) [Synchronized class]
* [java.nio.ByteBuffer#put()](https://docs.oracle.com/javase/6/docs/api/java/nio/ByteBuffer.html" \l "put(java.nio.ByteBuffer)" \o "ByteBuffer) (also on CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)