* What is design pattern in java?

A design pattern provides a general reusable solution for a common problem occurs in software design. The patterns typically show relationships and interactions between classes and objects. Design pattern are programming languages independent strategies for solving a common problem. By using the design patterns, we can make our code more flexible, reusable and maintainable.

* What is Gang of Four (GOF)?

In 1994, four authors published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development. These authors are collectively known as Gang of fours(GOF). According to them the principles of object oriented design:

1. Program to an interface not its implementation
2. Favor object composition over inheritance.

* Types of design pattern?

There are mainly three types of design patterns:

1. Creational -> These design patterns are all about class instantiation or object creation.
2. Structural -> These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.
3. Behavioral -> Behavioral patterns are about identifying common communication patterns between objects and realize these patterns.

* What are the types of creational design pattern?

1. Singleton Pattern
2. Factory Method pattern
3. Abstract factory pattern
4. Prototype pattern
5. Builder pattern
6. Object pool pattern

* What are the types of Structural design pattern?

1. Adaptor Pattern
2. Bridge pattern
3. Composite pattern
4. Decorator pattern
5. Façade pattern
6. Flyweight pattern
7. Proxy pattern

* What are the types of Behavioral pattern?

1. Chain of responsibility pattern
2. Command pattern
3. Interpreter pattern
4. Iterator pattern
5. Mediator pattern
6. Memento Pattern
7. Observer pattern
8. State Pattern
9. Strategy pattern
10. Template pattern
11. Visitor pattern
12. Null Object

* What is Factory method pattern?

A Factory pattern or a Factory method pattern says that just define an interface or abstract class for creating an object but let the sub classes decide which class to instantiate. In other word, subclasses are responsible for creating the instance of the class.

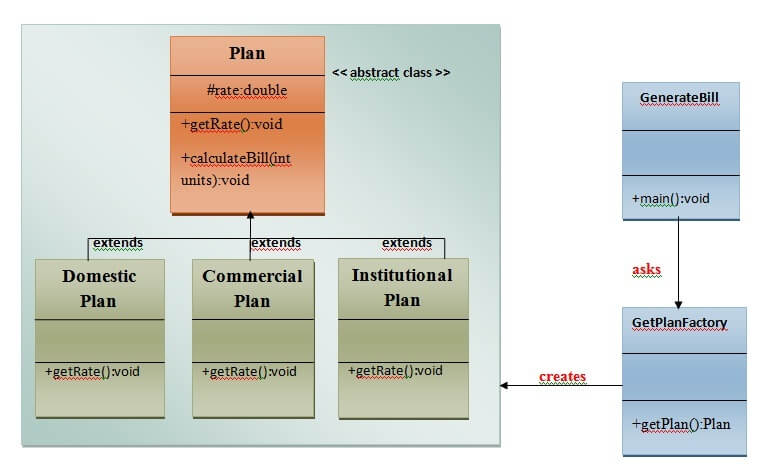
Advantage of factory design pattern:

1. Factory method pattern allows the subclasses to choose the type of objects to create.
2. It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code.

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.

Example:



1. **package** FactoryDesignPattern;
3. **import** java.io.\*;
5. **abstract** **class** Plan {
6. **protected** **double** rate;
7. **abstract** **void** getRate();
9. **public** **void** calculateBill(**int** units) {
10. System.out.println(units \* rate);
11. }
12. }
14. **class** DomesticPlan **extends** Plan {
15. // @override
16. **public** **void** getRate() {
17. rate = 3.50;
18. }
19. }
21. **class** CommercialPlan **extends** Plan {
22. // @override
23. **public** **void** getRate() {
24. rate = 7.50;
25. }
26. }
28. **class** InstitutionalPlan **extends** Plan {
29. // @override
30. **public** **void** getRate() {
31. rate = 5.50;
32. }
33. }
35. **class** GetPlanFactory {
36. // use getPlan method to get object of type Plan
37. **public** Plan getPlan(String planType) {
38. **if** (planType == **null**) {
39. **return** **null**;
40. }
41. **if** (planType.equalsIgnoreCase("DOMESTICPLAN")) {
42. **return** **new** DomesticPlan();
43. } **else** **if** (planType.equalsIgnoreCase("COMMERCIALPLAN")) {
44. **return** **new** CommercialPlan();
45. } **else** **if** (planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
46. **return** **new** InstitutionalPlan();
47. }
48. **return** **null**;
49. }
50. }
52. **class** GenerateBill {
53. **public** **static** **void** main(String args[]) **throws** IOException {
54. GetPlanFactory planFactory = **new** GetPlanFactory();
56. System.out.print("Enter the name of plan for which the bill will be generated: ");
57. BufferedReader br = **new** BufferedReader(**new** InputStreamReader(System.in));
59. String planName = br.readLine();
60. System.out.print("Enter the number of units for bill will be calculated: ");
61. **int** units = Integer.parseInt(br.readLine());
63. Plan p = planFactory.getPlan(planName);
64. // call getRate() method and calculateBill()method of DomesticPaln.
66. System.out.print("Bill amount for " + planName + " of  " + units + " units is: ");
67. p.getRate();
68. p.calculateBill(units);
69. }
70. }

* What is singleton design pattern?

At any point of time, in the application, there will be only one instance of the class will present.

Ways to achieve singleton design pattern.

1. **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.
2. **public** **class** GFG
3. {
4. // public instance initialized when loading the class
5. **private** **static** **final** GFG instance = **new** GFG();
7. **private** GFG()
8. {
9. // private constructor
10. }
11. **public** **static** GFG getInstance(){
12. **return** instance;
13. }
14. }

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.
4. **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.
5. // Java code to create singleton class
6. // Using Static block
7. **public** **class** GFG
8. {
9. // public instance
10. **public** **static** GFG instance;
12. **private** GFG()
13. {
14. // private constructor
15. }
17. {
18. // static block to initialize instance
19. instance = **new** GFG();
20. }
21. }

**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.
3. **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. 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.
4. //Java Code to create singleton class
5. // With Lazy initialization
6. **public** **class** GFG
7. {
8. // private instance, so that it can be
9. // accessed by only by getInstance() method
10. **private** **static** GFG instance;
12. **private** GFG() {}
14. //method to return instance of class
15. **public** **static** GFG getInstance()
16. {
17. **if** (instance == **null**)
18. {
19. // if instance is null, initialize
20. instance = **new** GFG();
21. }
22. **return** instance;
23. }
24. }

**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.
4. **Thread Safe Singleton:** To make a singleton class thread-safe, getInstance() method is made synchronized so that multiple threads can’t access it simultaneously.
5. // Java program to create Thread Safe
6. // Singleton class
7. **public** **class** GFG
8. {
9. // private instance, so that it can be
10. // accessed by only by getInstance() method
11. **private** **static** GFG instance;
13. **private** GFG()
14. {
15. // private constructor
16. }
18. //synchronized method to control simultaneous access
19. **synchronized** **public** **static** GFG getInstance()
20. {
21. **if** (instance == **null**)
22. {
23. // if instance is null, initialize
24. instance = **new** GFG();
25. }
26. **return** instance;
27. }
28. }

**Pros:**

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

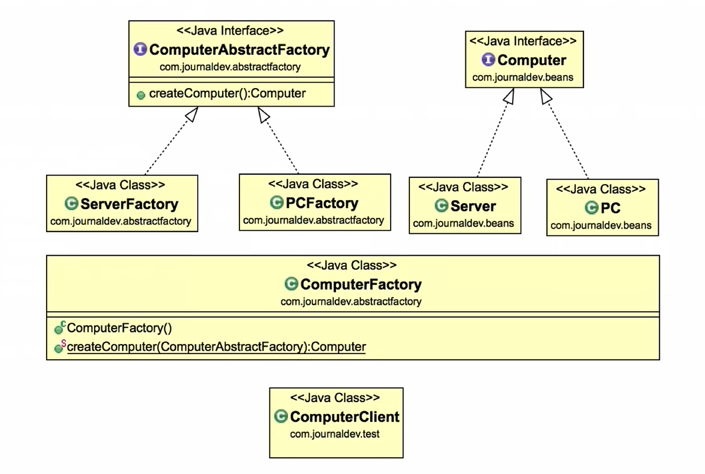
**Cons:**

1. getInstance() method is synchronized so it causes slow performance as multiple threads can’t access it simultaneously.
2. **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.
3. // Java code to explain double check locking
4. **public** **class** GFG
5. {
6. // private instance, so that it can be
7. // accessed by only by getInstance() method
8. **private** **static** GFG instance;
10. **private** GFG()
11. {
12. // private constructor
13. }
15. **public** **static** GFG getInstance()
16. {
17. **if** (instance == **null**)
18. {
19. //synchronized block to remove overhead
20. **synchronized** (GFG.**class**)
21. {
22. **if**(instance==**null**)
23. {
24. // if instance is null, initialize
25. instance = **new** GFG();
26. }
28. }
29. }
30. **return** instance;
31. }
32. }
33. **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.
34. **public** **class** GFG
35. {
37. **private** GFG() {}
39. // Inner class to provide instance of class
40. **private** **static** **class** BillPughSingleton
41. {
42. **private** **static** **final** GFG INSTANCE = **new** GFG();
43. }
45. **public** **static** GFG getInstance()
46. {
47. **return** BillPughSingleton.INSTANCE;
48. }
49. }

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.

* What is abstract factory design pattern?

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



1. **package** com.journaldev.design.model;
3. **public** **abstract** **class** Computer {
5. **public** **abstract** String getRAM();
6. **public** **abstract** String getHDD();
7. **public** **abstract** String getCPU();
9. @Override
10. **public** String toString(){
11. **return** "RAM= "+**this**.getRAM()+", HDD="+**this**.getHDD()+", CPU="+**this**.getCPU();
12. }
13. }
14. **public** **class** PC **extends** Computer {
16. **private** String ram;
17. **private** String hdd;
18. **private** String cpu;
20. **public** PC(String ram, String hdd, String cpu){
21. **this**.ram=ram;
22. **this**.hdd=hdd;
23. **this**.cpu=cpu;
24. }
25. @Override
26. **public** String getRAM() {
27. **return** **this**.ram;
28. }
30. @Override
31. **public** String getHDD() {
32. **return** **this**.hdd;
33. }
35. @Override
36. **public** String getCPU() {
37. **return** **this**.cpu;
38. }
40. }
41. **public** **class** Server **extends** Computer {
43. **private** String ram;
44. **private** String hdd;
45. **private** String cpu;
47. **public** Server(String ram, String hdd, String cpu){
48. **this**.ram=ram;
49. **this**.hdd=hdd;
50. **this**.cpu=cpu;
51. }
52. @Override
53. **public** String getRAM() {
54. **return** **this**.ram;
55. }
57. @Override
58. **public** String getHDD() {
59. **return** **this**.hdd;
60. }
62. @Override
63. **public** String getCPU() {
64. **return** **this**.cpu;
65. }
67. }
68. **public** **interface** ComputerAbstractFactory {
70. **public** Computer createComputer();
72. }
73. **public** **class** PCFactory **implements** ComputerAbstractFactory {
75. **private** String ram;
76. **private** String hdd;
77. **private** String cpu;
79. **public** PCFactory(String ram, String hdd, String cpu){
80. **this**.ram=ram;
81. **this**.hdd=hdd;
82. **this**.cpu=cpu;
83. }
84. @Override
85. **public** Computer createComputer() {
86. **return** **new** PC(ram,hdd,cpu);
87. }
89. }
90. **public** **class** ServerFactory **implements** ComputerAbstractFactory {
92. **private** String ram;
93. **private** String hdd;
94. **private** String cpu;
96. **public** ServerFactory(String ram, String hdd, String cpu){
97. **this**.ram=ram;
98. **this**.hdd=hdd;
99. **this**.cpu=cpu;
100. }
102. @Override
103. **public** Computer createComputer() {
104. **return** **new** Server(ram,hdd,cpu);
105. }
107. }

Now we will create a consumer class that will provide the entry point for the client classes to create sub-classes.

1. **public** **class** ComputerFactory {
3. **public** **static** Computer getComputer(ComputerAbstractFactory factory){
4. **return** factory.createComputer();
5. }
6. }

Test application:

1. **public** **class** TestDesignPatterns {
3. **public** **static** **void** main(String[] args) {
4. testAbstractFactory();
5. }
7. **private** **static** **void** testAbstractFactory() {
8. Computer pc = com.journaldev.design.abstractfactory.ComputerFactory.getComputer(**new** PCFactory("2 GB","500 GB","2.4 GHz"));
9. Computer server = com.journaldev.design.abstractfactory.ComputerFactory.getComputer(**new** ServerFactory("16 GB","1 TB","2.9 GHz"));
10. System.out.println("AbstractFactory PC Config::"+pc);
11. System.out.println("AbstractFactory Server Config::"+server);
12. }
13. }

**Abstract Factory Design Pattern Benefits**

* Abstract Factory design pattern provides approach to code for interface rather than implementation.
* Abstract Factory pattern is “factory of factories” and can be easily extended to accommodate more products, for example we can add another sub-class Laptop and a factory LaptopFactory.
* Abstract Factory pattern is robust and avoid conditional logic of Factory pattern.
* What is builder design pattern?

Builder pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

There are three major issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

* 1. Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side its hard to maintain the order of the argument.
  2. Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
  3. If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.

We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters. The problem with this approach is that the Object state will be inconsistent until unless all the attributes are set explicitly.

Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

Let’s see how we can implement builder design pattern in java.

1. First of all you need to create a static nested class and then copy all the arguments from the outer class to the Builder class. We should follow the naming convention and if the class name is Computer then builder class should be named as ComputerBuilder.
2. Java Builder class should have a public constructor with all the required attributes as parameters.
3. Java Builder class should have methods to set the optional parameters and it should return the same Builder object after setting the optional attribute.
4. The final step is to provide a build() method in the builder class that will return the Object needed by client program. For this we need to have a private constructor in the Class with Builder class as argument.

Here is the sample builder pattern example code where we have a Computer class and ComputerBuilder class to build it.

1. **public** **class** Computer {
3. //required parameters
4. **private** String HDD;
5. **private** String RAM;
7. //optional parameters
8. **private** **boolean** isGraphicsCardEnabled;
9. **private** **boolean** isBluetoothEnabled;

12. **public** String getHDD() {
13. **return** HDD;
14. }
16. **public** String getRAM() {
17. **return** RAM;
18. }
20. **public** **boolean** isGraphicsCardEnabled() {
21. **return** isGraphicsCardEnabled;
22. }
24. **public** **boolean** isBluetoothEnabled() {
25. **return** isBluetoothEnabled;
26. }
28. **private** Computer(ComputerBuilder builder) {
29. **this**.HDD=builder.HDD;
30. **this**.RAM=builder.RAM;
31. **this**.isGraphicsCardEnabled=builder.isGraphicsCardEnabled;
32. **this**.isBluetoothEnabled=builder.isBluetoothEnabled;
33. }
35. //Builder Class
36. **public** **static** **class** ComputerBuilder{
38. // required parameters
39. **private** String HDD;
40. **private** String RAM;
42. // optional parameters
43. **private** **boolean** isGraphicsCardEnabled;
44. **private** **boolean** isBluetoothEnabled;
46. **public** ComputerBuilder(String hdd, String ram){
47. **this**.HDD=hdd;
48. **this**.RAM=ram;
49. }
51. **public** ComputerBuilder setGraphicsCardEnabled(**boolean** isGraphicsCardEnabled) {
52. **this**.isGraphicsCardEnabled = isGraphicsCardEnabled;
53. **return** **this**;
54. }
56. **public** ComputerBuilder setBluetoothEnabled(**boolean** isBluetoothEnabled) {
57. **this**.isBluetoothEnabled = isBluetoothEnabled;
58. **return** **this**;
59. }
61. **public** Computer build(){
62. **return** **new** Computer(**this**);
63. }
65. }
67. }

Notice that Computer class has only getter methods and no public constructor. So the only way to get a Computer object is through the ComputerBuilder class.

Test code:

1. **public** **class** TestBuilderPattern {
3. **public** **static** **void** main(String[] args) {
4. //Using builder to get the object in a single line of code and
5. //without any inconsistent state or arguments management issues
6. Computer comp = **new** Computer.ComputerBuilder(
7. "500 GB", "2 GB").setBluetoothEnabled(**true**)
8. .setGraphicsCardEnabled(**true**).build();
9. }
10. }