* 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.

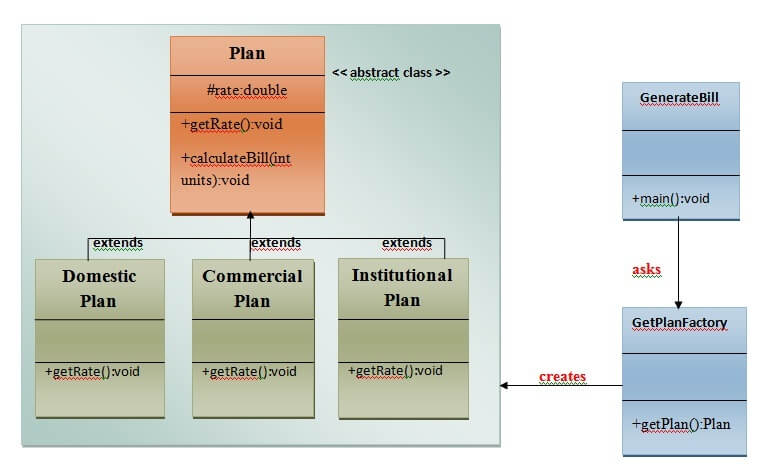
Advantage of factory design pattern:

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.

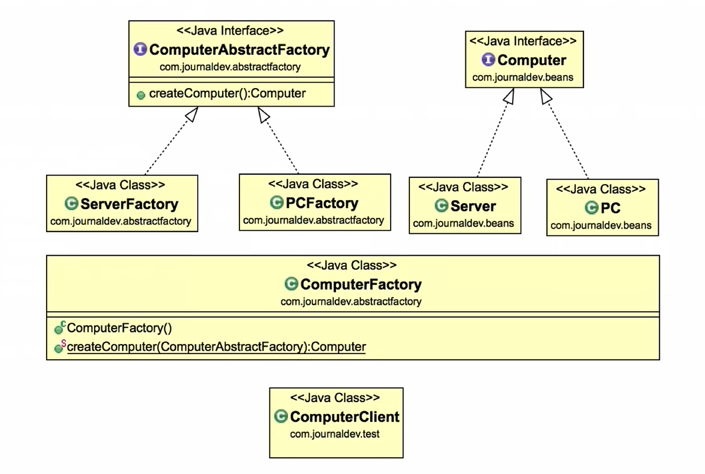
**Cons:**

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. }

Some of the builder design pattern in java JDK are StringBuilder, StringBuffer.

* What is prototype design pattern?

Prototype design pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing. Prototype pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. Prototype design pattern uses java cloning to copy the object.

Suppose we have an Object that loads data from database. Now we need to modify this data in our program multiple times, so it’s not a good idea to create the Object using new keyword and load all the data again from database.

The better approach would be to clone the existing object into a new object and then do the data manipulation.

Prototype design pattern mandates that the Object which you are copying should provide the copying feature.

1. **import** java.util.ArrayList;
2. **import** java.util.List;
4. **public** **class** Employees **implements** Cloneable{
6. **private** List<String> empList;
8. **public** Employees(){
9. empList = **new** ArrayList<String>();
10. }
12. **public** Employees(List<String> list){
13. **this**.empList=list;
14. }
15. **public** **void** loadData(){
16. //read all employees from database and put into the list
17. empList.add("Pankaj");
18. empList.add("Raj");
19. empList.add("David");
20. empList.add("Lisa");
21. }
23. **public** List<String> getEmpList() {
24. **return** empList;
25. }
27. @Override
28. **public** Object clone() **throws** CloneNotSupportedException{
29. List<String> temp = **new** ArrayList<String>();
30. **for**(String s : **this**.getEmpList()){
31. temp.add(s);
32. }
33. **return** **new** Employees(temp);
34. }
36. }

Driver code:

1. **package** com.journaldev.design.test;
3. **import** java.util.List;
5. **import** com.journaldev.design.prototype.Employees;
7. **public** **class** PrototypePatternTest {
9. **public** **static** **void** main(String[] args) **throws** CloneNotSupportedException {
10. Employees emps = **new** Employees();
11. emps.loadData();
13. //Use the clone method to get the Employee object
14. Employees empsNew = (Employees) emps.clone();
15. Employees empsNew1 = (Employees) emps.clone();
16. List<String> list = empsNew.getEmpList();
17. list.add("John");
18. List<String> list1 = empsNew1.getEmpList();
19. list1.remove("Pankaj");
21. System.out.println("emps List: "+emps.getEmpList());
22. System.out.println("empsNew List: "+list);
23. System.out.println("empsNew1 List: "+list1);
24. }
26. }

**Structural design pattern**

* What is adaptor design pattern?

Adapter design pattern is one of the structural design pattern and its used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an Adapter.

One of the great real life example of Adapter design pattern is mobile charger. Mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket.

We will try to implement multi-adapter using adapter design pattern in this tutorial.

So first of all we will have two classes – Volt (to measure volts) and Socket (producing constant volts of 120V).

1. **public** **class** Volt {
3. **private** **int** volts;
5. **public** Volt(**int** v){
6. **this**.volts=v;
7. }
9. **public** **int** getVolts() {
10. **return** volts;
11. }
13. **public** **void** setVolts(**int** volts) {
14. **this**.volts = volts;
15. }
17. }
18. **public** **class** Socket {
20. **public** Volt getVolt(){
21. **return** **new** Volt(120);
22. }
23. }

Now we want to build an adapter that can produce 3 volts, 12 volts and default 120 volts. So first of all we will create an adapter interface with these methods. The SocketAdapter interface can also be an abstract class.

1. **public** **interface** SocketAdapter {
3. **public** Volt get120Volt();
5. **public** Volt get12Volt();
7. **public** Volt get3Volt();
8. }

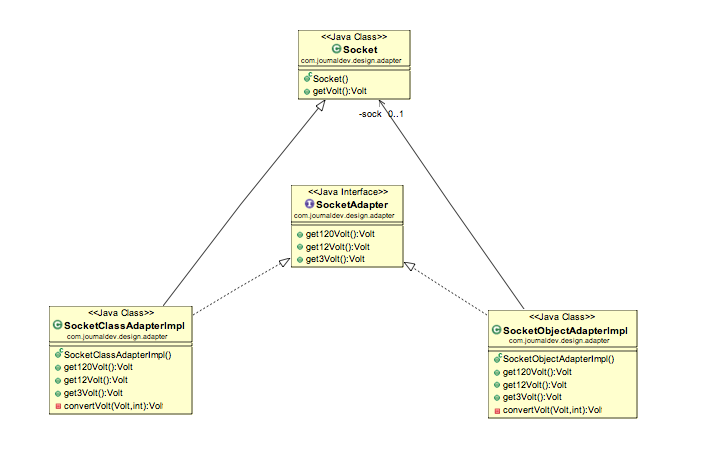
Two Way Adapter Pattern

While implementing Adapter pattern, there are two approaches – class adapter and object adapter – however both these approaches produce same result.

1. Class Adapter – This form uses java inheritance and extends the source interface, in our case Socket class.
2. Object Adapter – This form uses Java Composition and adapter contains the source object.

Here is the class adapter approach implementation of our adapter.

1. Using Inheritance for adaptor pattern.
2. **package** com.journaldev.design.adapter;
4. //Using inheritance for adapter pattern
5. **public** **class** SocketClassAdapterImpl **extends** Socket **implements** SocketAdapter{
7. @Override
8. **public** Volt get120Volt() {
9. **return** getVolt();
10. }
12. @Override
13. **public** Volt get12Volt() {
14. Volt v= getVolt();
15. **return** convertVolt(v,10);
16. }
18. @Override
19. **public** Volt get3Volt() {
20. Volt v= getVolt();
21. **return** convertVolt(v,40);
22. }
24. **private** Volt convertVolt(Volt v, **int** i) {
25. **return** **new** Volt(v.getVolts()/i);
26. }
28. }
29. Using composition for adaptor pattern
30. **public** **class** SocketObjectAdapterImpl **implements** SocketAdapter{
32. //Using Composition for adapter pattern
33. **private** Socket sock = **new** Socket();
35. @Override
36. **public** Volt get120Volt() {
37. **return** sock.getVolt();
38. }
40. @Override
41. **public** Volt get12Volt() {
42. Volt v= sock.getVolt();
43. **return** convertVolt(v,10);
44. }
46. @Override
47. **public** Volt get3Volt() {
48. Volt v= sock.getVolt();
49. **return** convertVolt(v,40);
50. }
52. **private** Volt convertVolt(Volt v, **int** i) {
53. **return** **new** Volt(v.getVolts()/i);
54. }
55. }



Test Program:

1. **import** com.journaldev.design.adapter.SocketAdapter;
2. **import** com.journaldev.design.adapter.SocketClassAdapterImpl;
3. **import** com.journaldev.design.adapter.SocketObjectAdapterImpl;
4. **import** com.journaldev.design.adapter.Volt;
6. **public** **class** AdapterPatternTest {
8. **public** **static** **void** main(String[] args) {
10. testClassAdapter();
11. testObjectAdapter();
12. }
14. **private** **static** **void** testObjectAdapter() {
15. SocketAdapter sockAdapter = **new** SocketObjectAdapterImpl();
16. Volt v3 = getVolt(sockAdapter,3);
17. Volt v12 = getVolt(sockAdapter,12);
18. Volt v120 = getVolt(sockAdapter,120);
19. System.out.println("v3 volts using Object Adapter="+v3.getVolts());
20. System.out.println("v12 volts using Object Adapter="+v12.getVolts());
21. System.out.println("v120 volts using Object Adapter="+v120.getVolts());
22. }
24. **private** **static** **void** testClassAdapter() {
25. SocketAdapter sockAdapter = **new** SocketClassAdapterImpl();
26. Volt v3 = getVolt(sockAdapter,3);
27. Volt v12 = getVolt(sockAdapter,12);
28. Volt v120 = getVolt(sockAdapter,120);
29. System.out.println("v3 volts using Class Adapter="+v3.getVolts());
30. System.out.println("v12 volts using Class Adapter="+v12.getVolts());
31. System.out.println("v120 volts using Class Adapter="+v120.getVolts());
32. }
34. **private** **static** Volt getVolt(SocketAdapter sockAdapter, **int** i) {
35. **switch** (i){
36. **case** 3: **return** sockAdapter.get3Volt();
37. **case** 12: **return** sockAdapter.get12Volt();
38. **case** 120: **return** sockAdapter.get120Volt();
39. **default**: **return** sockAdapter.get120Volt();
40. }
41. }
42. }

Some examples of adaptor design pattern in java JDK.

java.util.Arrays#asList()

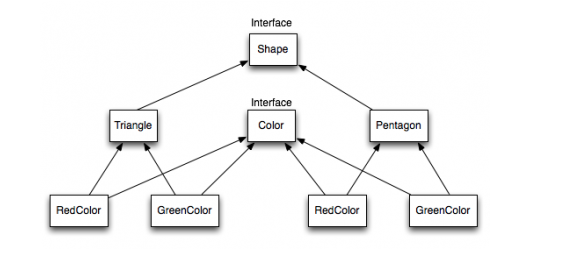
java.io.InputStreamReader(InputStream) (returns a Reader)

* What is Bridge design pattern in java?

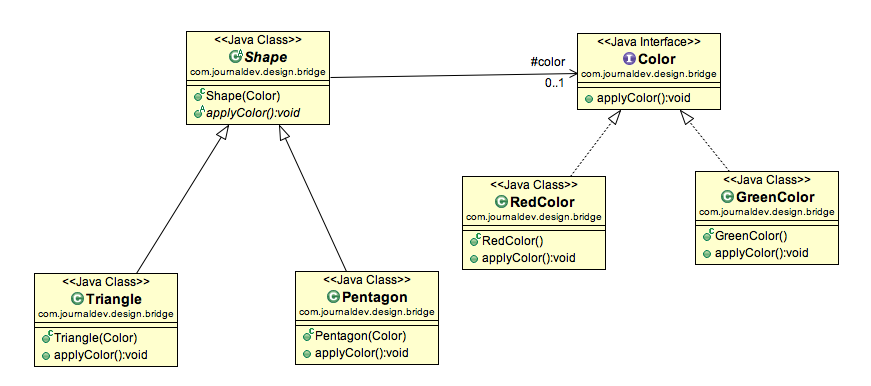
When we have interface hierarchies in both interfaces as well as implementations, then bridge design pattern is used to decouple the interfaces from implementation and hiding the implementation details from the client programs.

Decouple an abstraction from its implementation so that the two can vary independently

The implementation of bridge design pattern follows the notion to **prefer Composition over inheritance.**



Now we will use bridge design pattern to decouple the interfaces from implementation. UML diagram for the classes and interfaces after applying bridge pattern will look like below image.



Notice the bridge between Shape and Color interfaces and use of composition in implementing the bridge pattern.

Here is the java code for Shape and Color interfaces

1. **public** **interface** Color {
3. **public** **void** applyColor();
4. }
5. **public** **abstract** **class** Shape {
6. //Composition - implementor
7. **protected** Color color;
9. //constructor with implementor as input argument
10. **public** Shape(Color c){
11. **this**.color=c;
12. }
14. **abstract** **public** **void** applyColor();
15. }

We have Triangle and Pentagon implementation classes as below.

1. **public** **class** Triangle **extends** Shape{
3. **public** Triangle(Color c) {
4. **super**(c);
5. }
7. @Override
8. **public** **void** applyColor() {
9. System.out.print("Triangle filled with color ");
10. color.applyColor();
11. }
13. }
14. **public** **class** Pentagon **extends** Shape{
16. **public** Pentagon(Color c) {
17. **super**(c);
18. }
20. @Override
21. **public** **void** applyColor() {
22. System.out.print("Pentagon filled with color ");
23. color.applyColor();
24. }
26. }

Here are the implementation classes for RedColor and GreenColor.

1. **public** **class** RedColor **implements** Color{
3. **public** **void** applyColor(){
4. System.out.println("red.");
5. }
6. }
7. **public** **class** GreenColor **implements** Color{
9. **public** **void** applyColor(){
10. System.out.println("green.");
11. }
12. }

Lets test our bridge pattern implementation with a test program.

1. **public** **class** BridgePatternTest {
3. **public** **static** **void** main(String[] args) {
4. Shape tri = **new** Triangle(**new** RedColor());
5. tri.applyColor();
7. Shape pent = **new** Pentagon(**new** GreenColor());
8. pent.applyColor();
9. }
11. }

Bridge design pattern can be used when both abstraction and implementation can have different hierarchies independently and we want to hide the implementation from the client application.

**Behavioral Pattern**

* What is **state design pattern**?

State design pattern is one of the behavioral design pattern. State design pattern is used when an Object change its behavior based on its internal state.

If we have to change the behavior of an object based on its state, we can have a state variable in the Object. Then use if-else condition block to perform different actions based on the state. State design pattern is used to provide a systematic and loosely coupled way to achieve this through **Context** and **State** implementations.

State Pattern Context is the class that has a State reference to one of the concrete implementations of the State. Context forwards the request to the state object for processing. Let’s understand this with a simple example.

Suppose we want to implement a TV Remote with a simple button to perform action.

If the State is ON, it will turn on the TV and if state is OFF, it will turn off the TV.

We can implement it using if-else condition like below;

1. **public** **class** TVRemoteBasic {
3. **private** String state="";
5. **public** **void** setState(String state){
6. **this**.state=state;
7. }
9. **public** **void** doAction(){
10. **if**(state.equalsIgnoreCase("ON")){
11. System.out.println("TV is turned ON");
12. }**else** **if**(state.equalsIgnoreCase("OFF")){
13. System.out.println("TV is turned OFF");
14. }
15. }
17. **public** **static** **void** main(String args[]){
18. TVRemoteBasic remote = **new** TVRemoteBasic();
20. remote.setState("ON");
21. remote.doAction();
23. remote.setState("OFF");
24. remote.doAction();
25. }
27. }

Notice that client code should know the specific values to use for setting the state of remote. Further more if number of states increase then the tight coupling between implementation and the client code will be very hard to maintain and extend.

Now we will use State pattern to implement above TV Remote example.

State Design Pattern Interface: First of all we will create State interface that will define the method that should be implemented by different concrete states and context class.

1. **public** **interface** State {
3. **public** **void** doAction();
4. }

In our example, we can have two states – one for turning TV on and another to turn it off. So we will create two concrete state implementations for these behaviors.

1. **public** **class** TVStartState **implements** State {
3. @Override
4. **public** **void** doAction() {
5. System.out.println("TV is turned ON");
6. }
8. }
9. **public** **class** TVStopState **implements** State {
11. @Override
12. **public** **void** doAction() {
13. System.out.println("TV is turned OFF");
14. }
16. }

Now we are ready to implement our Context object that will change its behavior based on its internal state.

1. **public** **class** TVContext **implements** State {
3. **private** State tvState;
5. **public** **void** setState(State state) {
6. **this**.tvState=state;
7. }
9. **public** State getState() {
10. **return** **this**.tvState;
11. }
13. @Override
14. **public** **void** doAction() {
15. **this**.tvState.doAction();
16. }
18. }

Notice that Context also implements State and keep a reference of its current state and forwards the request to the state implementation.

Testcode

1. **public** **class** TVRemote {
3. **public** **static** **void** main(String[] args) {
4. TVContext context = **new** TVContext();
5. State tvStartState = **new** TVStartState();
6. State tvStopState = **new** TVStopState();
8. context.setState(tvStartState);
9. context.doAction();

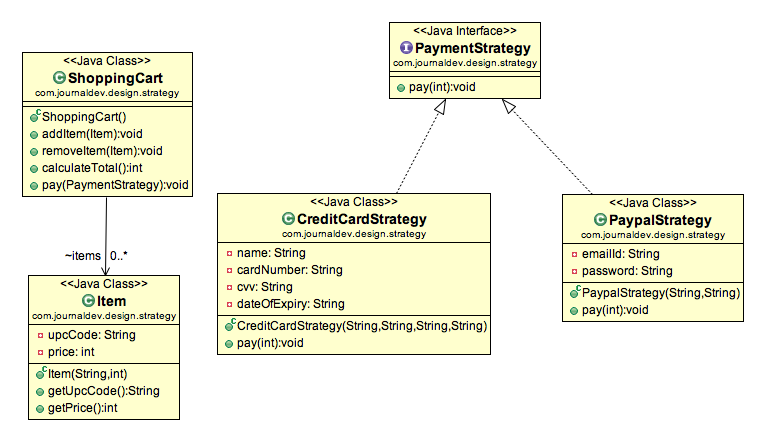
12. context.setState(tvStopState);
13. context.doAction();
15. }
17. }

* What is **Strategy design pattern** in java?

Strategy pattern is also known as Policy Pattern. We define multiple algorithms and let client application pass the algorithm to be used as a parameter.

One of the best example of strategy pattern is **Collections.sort()** method that takes Comparator parameter. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways.

we will try to implement a simple Shopping Cart where we have two payment strategies – using Credit Card or using PayPal.



1. **public** **interface** PaymentStrategy {
3. **public** **void** pay(**int** amount);
4. }

Now we will have to create concrete implementation of algorithms for payment using credit/debit card or through paypal.

1. **public** **class** CreditCardStrategy **implements** PaymentStrategy {
3. **private** String name;
4. **private** String cardNumber;
5. **private** String cvv;
6. **private** String dateOfExpiry;
8. **public** CreditCardStrategy(String nm, String ccNum, String cvv, String expiryDate){
9. **this**.name=nm;
10. **this**.cardNumber=ccNum;
11. **this**.cvv=cvv;
12. **this**.dateOfExpiry=expiryDate;
13. }
14. @Override
15. **public** **void** pay(**int** amount) {
16. System.out.println(amount +" paid with credit/debit card");
17. }
19. }
20. **public** **class** PaypalStrategy **implements** PaymentStrategy {
22. **private** String emailId;
23. **private** String password;
25. **public** PaypalStrategy(String email, String pwd){
26. **this**.emailId=email;
27. **this**.password=pwd;
28. }
30. @Override
31. **public** **void** pay(**int** amount) {
32. System.out.println(amount + " paid using Paypal.");
33. }
35. }

Now our strategy pattern example algorithms are ready. We can implement Shopping Cart and payment method will require input as Payment strategy.

1. **public** **class** Item {
3. **private** String upcCode;
4. **private** **int** price;
6. **public** Item(String upc, **int** cost){
7. **this**.upcCode=upc;
8. **this**.price=cost;
9. }
11. **public** String getUpcCode() {
12. **return** upcCode;
13. }
15. **public** **int** getPrice() {
16. **return** price;
17. }
19. }
20. **public** **class** ShoppingCart {
22. //List of items
23. List<Item> items;
25. **public** ShoppingCart(){
26. **this**.items=**new** ArrayList<Item>();
27. }
29. **public** **void** addItem(Item item){
30. **this**.items.add(item);
31. }
33. **public** **void** removeItem(Item item){
34. **this**.items.remove(item);
35. }
37. **public** **int** calculateTotal(){
38. **int** sum = 0;
39. **for**(Item item : items){
40. sum += item.getPrice();
41. }
42. **return** sum;
43. }
45. **public** **void** pay(PaymentStrategy paymentMethod){
46. **int** amount = calculateTotal();
47. paymentMethod.pay(amount);
48. }
49. }

Notice that payment method of shopping cart requires payment algorithm as argument and doesn’t store it anywhere as instance variable.

Test:

1. **public** **class** ShoppingCartTest {
3. **public** **static** **void** main(String[] args) {
4. ShoppingCart cart = **new** ShoppingCart();
6. Item item1 = **new** Item("1234",10);
7. Item item2 = **new** Item("5678",40);
9. cart.addItem(item1);
10. cart.addItem(item2);
12. //pay by paypal
13. cart.pay(**new** PaypalStrategy("myemail@example.com", "mypwd"));
15. //pay by credit card
16. cart.pay(**new** CreditCardStrategy("Pankaj Kumar", "1234567890123456", "786", "12/15"));
17. }
19. }

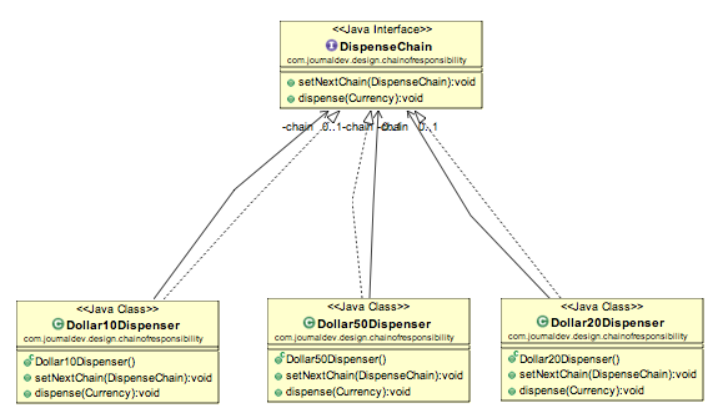
**Strategy Design Pattern Important Points**

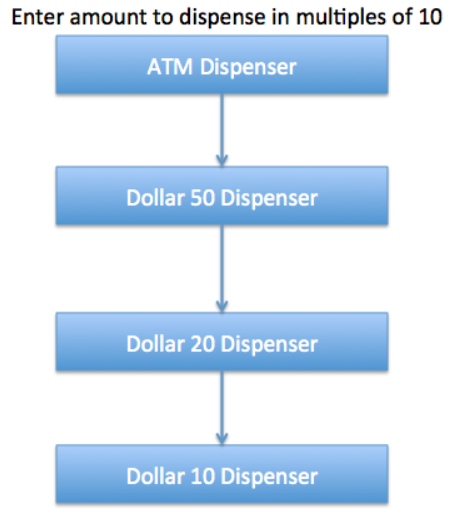
* We could have used composition to create instance variable for strategies but we should avoid that as we want the specific strategy to be applied for a particular task. Same is followed in Collections.sort() and Arrays.sort() method that take comparator as argument.
* Strategy Pattern is very similar to State Pattern. One of the difference is that Context contains state as instance variable and there can be multiple tasks whose implementation can be dependent on the state whereas in strategy pattern strategy is passed as argument to the method and context object doesn’t have any variable to store it.
* Strategy pattern is useful when we have multiple algorithms for specific task and we want our application to be flexible to chose any of the algorithm at runtime for specific task.
* What is Chain of responsibility pattern in java?

Chain of responsibility pattern is used to achieve loose coupling in software design where a request from client is passed to a chain of objects to process them. Then the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

In JDK, we have multiple catch blocks in a try-catch block code. Here every catch block is kind of a processor to process that particular exception.

So when any exception occurs in the try block, its send to the first catch block to process. If the catch block is not able to process it, it forwards the request to next object in chain i.e next catch block. If even the last catch block is not able to process it, the exception is thrown outside of the chain to the calling program.





1. **public** **class** Currency {
3. **private** **int** amount;
5. **public** Currency(**int** amt){
6. **this**.amount=amt;
7. }
9. **public** **int** getAmount(){
10. **return** **this**.amount;
11. }
12. }

The base interface should have a method to define the next processor in the chain and the method that will process the request. Our ATM Dispense interface will look like below.

*Chain of Responsibilities Pattern – Chain Implementations*

We need to create different processor classes that will implement the DispenseChain interface and provide implementation of dispense methods. Since we are developing our system to work with three types of currency bills – 50$, 20$ and 10$, we will create three concrete implementations.

1. **public** **class** Dollar50Dispenser **implements** DispenseChain {
3. **private** DispenseChain chain;
5. @Override
6. **public** **void** setNextChain(DispenseChain nextChain) {
7. **this**.chain=nextChain;
8. }
10. @Override
11. **public** **void** dispense(Currency cur) {
12. **if**(cur.getAmount() >= 50){
13. **int** num = cur.getAmount()/50;
14. **int** remainder = cur.getAmount() % 50;
15. System.out.println("Dispensing "+num+" 50$ note");
16. **if**(remainder !=0) **this**.chain.dispense(**new** Currency(remainder));
17. }**else**{
18. **this**.chain.dispense(cur);
19. }
20. }
22. }
23. **public** **class** Dollar20Dispenser **implements** DispenseChain{
25. **private** DispenseChain chain;
27. @Override
28. **public** **void** setNextChain(DispenseChain nextChain) {
29. **this**.chain=nextChain;
30. }
32. @Override
33. **public** **void** dispense(Currency cur) {
34. **if**(cur.getAmount() >= 20){
35. **int** num = cur.getAmount()/20;
36. **int** remainder = cur.getAmount() % 20;
37. System.out.println("Dispensing "+num+" 20$ note");
38. **if**(remainder !=0) **this**.chain.dispense(**new** Currency(remainder));
39. }**else**{
40. **this**.chain.dispense(cur);
41. }
42. }
44. }
45. **public** **class** Dollar10Dispenser **implements** DispenseChain {
47. **private** DispenseChain chain;
49. @Override
50. **public** **void** setNextChain(DispenseChain nextChain) {
51. **this**.chain=nextChain;
52. }
54. @Override
55. **public** **void** dispense(Currency cur) {
56. **if**(cur.getAmount() >= 10){
57. **int** num = cur.getAmount()/10;
58. **int** remainder = cur.getAmount() % 10;
59. System.out.println("Dispensing "+num+" 10$ note");
60. **if**(remainder !=0) **this**.chain.dispense(**new** Currency(remainder));
61. }**else**{
62. **this**.chain.dispense(cur);
63. }
64. }
66. }

The important point to note here is the implementation of dispense method. You will notice that every implementation is trying to process the request and based on the amount, it might process some or full part of it.

If one of the chain not able to process it fully, it sends the request to the next processor in chain to process the remaining request. If the processor is not able to process anything, it just forwards the same request to the next chain.

Chain of Responsibilities Design Pattern – Creating the Chain

1. **public** **class** ATMDispenseChain {
3. **private** DispenseChain c1;
5. **public** ATMDispenseChain() {
6. // initialize the chain
7. **this**.c1 = **new** Dollar50Dispenser();
8. DispenseChain c2 = **new** Dollar20Dispenser();
9. DispenseChain c3 = **new** Dollar10Dispenser();
11. // set the chain of responsibility
12. c1.setNextChain(c2);
13. c2.setNextChain(c3);
14. }
16. **public** **static** **void** main(String[] args) {
17. ATMDispenseChain atmDispenser = **new** ATMDispenseChain();
18. **while** (**true**) {
19. **int** amount = 0;
20. System.out.println("Enter amount to dispense");
21. Scanner input = **new** Scanner(System.in);
22. amount = input.nextInt();
23. **if** (amount % 10 != 0) {
24. System.out.println("Amount should be in multiple of 10s.");
25. **return**;
26. }
27. // process the request
28. atmDispenser.c1.dispense(**new** Currency(amount));
29. }
31. }
32. }

When we run above application, we get output like below.

1. Enter amount to dispense
2. 530
3. Dispensing 10 50$ note
4. Dispensing 1 20$ note
5. Dispensing 1 10$ note
6. Enter amount to dispense
7. 100
8. Dispensing 2 50$ note
9. Enter amount to dispense
10. 120
11. Dispensing 2 50$ note
12. Dispensing 1 20$ note
13. Enter amount to dispense
14. 15
15. Amount should be in multiple of 10s.

Chain of Responsibility Design Pattern Important Points

* Client doesn’t know which part of the chain will be processing the request and it will send the request to the first object in the chain.
* Each object in the chain will have it’s own implementation to process the request, either full or partial or to send it to the next object in the chain.
* Every object in the chain should have reference to the next object in chain to forward the request to, its achieved by java composition.
* Creating the chain carefully is very important otherwise there might be a case that the request will never be forwarded to a particular processor or there are no objects in the chain who are able to handle the request.
* Chain of Responsibility design pattern is good to achieve lose coupling but it comes with the trade-off of having a lot of implementation classes and maintenance problems if most of the code is common in all the implementations.

Chain of Responsibility Pattern Examples in JDK

java.util.logging.Logger#log()

javax.servlet.Filter#doFilter()

**Design Pattern Interview Questions**

* How to design a parking lot using object-oriented principles?

For our purposes right now, we’ll make the following assumptions.

1) The parking lot has multiple levels. Each level has multiple rows of spots.

2) The parking lot can park motorcycles, cars, and buses.

3) The parking lot has motorcycle spots, compact spots, and large spots.

4) A motorcycle can park in any spot.

5) A car can park in either a single compact spot or a single large spot.

6) A bus can park in five large spots that are consecutive and within the same row. It cannot park in small spots.

In the below implementation, we have created an abstract class Vehicle, from which Car, Bus, and Motorcycle inherit. To handle the different parking spot sizes, we have just one class ParkingSpot which has a member variable indicating the size.

1. **public** **enum** VehicleSize {Motorcycle, Compact, Large }
3. **public** **abstract** **class** Vehicle
4. {
5. **protected** ArrayList<ParkingSpot> parkingSpots =  **new** ArrayList<ParkingSpot>();
6. **protected** String licensePlate;
7. **protected** **int** spotsNeeded;
8. **protected** VehicleSize size;
10. **public** **int** getSpotsNeeded()
11. {
12. **return** spotsNeeded;
13. }
14. **public** VehicleSize getSize()
15. {
16. **return** size;
17. }
19. /\* Park vehicle in this spot (among others,
20. potentially) \*/
21. **public** **void** parkinSpot(ParkingSpot s)
22. {
23. parkingSpots.add(s);
24. }

27. /\* Remove vehicle from spot, and notify spot
28. that it's gone \*/
29. **public** **void** clearSpots() { ... }
31. /\* Checks if the spot is big enough for the
32. vehicle (and is available).
33. This \* compares the SIZE only.It does not
34. check if it has enough spots. \*/
35. **public** **abstract** **boolean** canFitinSpot(ParkingSpot spot);
36. }
38. **public** **class** Bus **extends** Vehicle
39. {
40. **public** Bus()
41. {
42. spotsNeeded = 5;
43. size = VehicleSize.Large;
44. }
46. /\* Checks if the spot is a Large. Doesn't check
47. num of spots \*/
48. **public** **boolean** canFitinSpot(ParkingSpot spot)
49. {... }
50. }
52. **public** **class** Car **extends** Vehicle
53. {
54. **public** Car()
55. {
56. spotsNeeded = 1;
57. size = VehicleSize.Compact;
58. }
60. /\* Checks if the spot is a Compact or a Large. \*/
61. **public** **boolean** canFitinSpot(ParkingSpot spot)
62. { ... }
63. }
65. **public** **class** Motorcycle **extends** Vehicle
66. {
67. **public** Motorcycle()
68. {
69. spotsNeeded = 1;
70. size = VehicleSize.Motorcycle;
71. }
72. **public** **boolean** canFitinSpot(ParkingSpot spot)
73. { ... }
74. }

The ParkingSpot is implemented by having just a variable which represents the size of the spot. We could have implemented this by having classes for LargeSpot, CompactSpot, and MotorcycleSpot which inherit from ParkingSpot, but this is probably overkilled. The spots probably do not have different behaviors, other than their sizes.

1. **public** **class** ParkingSpot
2. {
3. **private** Vehicle vehicle;
4. **private** VehicleSize spotSize;
5. **private** **int** row;
6. **private** **int** spotNumber;
7. **private** Level level;
9. **public** ParkingSpot(Level lvl, **int** r, **int** n,
10. VehicleSize s)
11. { ... }
13. **public** **boolean** isAvailable()
14. {
15. **return** vehicle == **null**;
16. }
18. /\* Check if the spot is big enough and is available \*/
19. **public** **boolean** canFitVehicle(Vehicle vehicle) { ... }
21. /\* Park vehicle in this spot. \*/
22. **public** **boolean** park(Vehicle v) {..}
24. **public** **int** getRow()
25. {
26. **return** row;
27. }
28. **public** **int** getSpotNumber()
29. {
30. **return** spotNumber;
31. }
33. /\* Remove vehicle from spot, and notify
34. level that a new spot is available \*/
35. **public** **void** removeVehicle() { ... }
36. }

* Design an online book reader system.

Let’s assume we want to design a basic online reading system which provides the following functionality:

Searching the database of books and reading a book.

• User membership creation and extension.

• Only one active user at a time and only one active book by this user

The class OnlineReaderSystem represents the body of our program. We could implement

the class such that it stores information about all the books, deals with user management, and refreshes the display, but that would make this class rather hefty.Instead, we’ve chosen to tear off these components into Library, UserManager, and Display classes.

The classes:

1. User

2. Book

3. Library

4. UserManager

5. Display

6. OnlineReaderSystem

Full code is given below :

1. **class** OnlineReaderSystem {
2. **private** Library library;
3. **private** UserManager userManager;
4. **private** Display display;
5. **private** Book activeBook;
6. **private** User activeUser;
8. **public** OnlineReaderSystem()
9. {
10. userManager = **new** UserManager();
11. library = **new** Library();
12. display = **new** Display();
13. }
15. **public** Library getLibrary()
16. {
17. **return** library;
18. }
20. **public** UserManager getUserManager()
21. {
22. **return** userManager;
23. }
25. **public** Display getDisplay()
26. {
27. **return** display;
28. }
30. **public** Book getActiveBook()
31. {
32. **return** activeBook;
33. }
35. **public** **void** setActiveBook(Book book)
36. {
37. activeBook = book;
38. display.displayBook(book);
39. }
41. **public** User getActiveUser()
42. {
43. **return** activeUser;
44. }
46. **public** **void** setActiveUser(User user)
47. {
48. activeUser = user;
49. display.displayUser(user);
50. }
51. }
53. /\*
54. \* We then implement separate classes to handle the user
55. \* manager, the library, and the display components
56. \*/
58. /\*
59. \* This class represents the Library which is responsible
60. \* for storing and searching the books.
61. \*/
62. **class** Library {
63. **private** HashMap<Integer, Book> books;
65. **public** Library()
66. {
67. books = **new** HashMap<Integer, Book>();
68. }
70. **public** Boolean addBook(**int** id, String details, String title)
71. {
72. **if** (books.containsKey(id)) {
73. **return** **false**;
74. }
75. Book book = **new** Book(id, details, title);
76. books.put(id, book);
77. **return** **true**;
78. }
80. **public** Boolean addBook(Book book)
81. {
82. **if** (books.containsKey(book.getId())) {
83. **return** **false**;
84. }
86. books.put(book.getId(), book);
87. **return** **true**;
88. }
90. **public** **boolean** remove(Book b)
91. {
92. **return** remove(b.getId());
93. }
95. **public** **boolean** remove(**int** id)
96. {
97. **if** (!books.containsKey(id)) {
98. **return** **false**;
99. }
100. books.remove(id);
101. **return** **true**;
102. }
104. **public** Book find(**int** id)
105. {
106. **return** books.get(id);
107. }
108. }
110. /\*
111. \* This class represents the UserManager which is responsible
112. \* for managing the users, their membership etc.
113. \*/
115. **class** UserManager {
116. **private** HashMap<Integer, User> users;
118. **public** UserManager()
119. {
120. users = **new** HashMap<Integer, User>();
121. }
122. **public** Boolean addUser(**int** id, String details, String name)
123. {
124. **if** (users.containsKey(id)) {
125. **return** **false**;
126. }
127. User user = **new** User(id, details, name);
128. users.put(id, user);
129. **return** **true**;
130. }
132. **public** Boolean addUser(User user)
133. {
134. **if** (users.containsKey(user.getId())) {
135. **return** **false**;
136. }
138. users.put(user.getId(), user);
139. **return** **true**;
140. }
142. **public** **boolean** remove(User u)
143. {
144. **return** remove(u.getId());
145. }
147. **public** **boolean** remove(**int** id)
148. {
149. **if** (users.containsKey(id)) {
150. **return** **false**;
151. }
152. users.remove(id);
153. **return** **true**;
154. }
156. **public** User find(**int** id)
157. {
158. **return** users.get(id);
159. }
160. }
162. /\*
163. \* This class represents the Display, which is responsible
164. \* for displaying the book, it's pages and contents. It also
165. \* shows the current user. \* It provides the method
166. \* turnPageForward, turnPageBackward, refreshPage etc.
167. \*/
169. **class** Display {
170. **private** Book activeBook;
171. **private** User activeUser;
172. **private** **int** pageNumber = 0;
174. **public** **void** displayUser(User user)
175. {
176. activeUser = user;
177. refreshUsername();
178. }
180. **public** **void** displayBook(Book book)
181. {
182. pageNumber = 0;
183. activeBook = book;
185. refreshTitle();
186. refreshDetails();
187. refreshPage();
188. }
190. **public** **void** turnPageForward()
191. {
192. pageNumber++;
193. System.out.println("Turning forward to page no " +
194. pageNumber + " of book having title " +
195. activeBook.getTitle());
196. refreshPage();
197. }
199. **public** **void** turnPageBackward()
200. {
201. pageNumber--;
202. System.out.println("Turning backward to page no " +
203. pageNumber + " of book having title " +
204. activeBook.getTitle());
205. refreshPage();
206. }
208. **public** **void** refreshUsername()
209. {
210. /\* updates username display \*/
211. System.out.println("User name " + activeUser.getName() +
212. " is refreshed");
213. }
215. **public** **void** refreshTitle()
216. {
217. /\* updates title display \*/
218. System.out.println("Title of the book " +
219. activeBook.getTitle() + " refreshed");
220. }
222. **public** **void** refreshDetails()
223. {
224. /\* updates details display \*/
225. System.out.println("Details of the book " +
226. activeBook.getTitle() + " refreshed");
227. }
229. **public** **void** refreshPage()
230. {
231. /\* updated page display \*/
232. System.out.println("Page no " + pageNumber + " refreshed");
233. }
234. }
236. /\*
237. \* The classes for User and Book simply hold data and
238. \* provide little functionality.
239. \* This class represents the Book which is a simple POJO
240. \*/
242. **class** Book {
243. **private** **int** bookId;
244. **private** String details;
245. **private** String title;
247. **public** Book(**int** id, String details, String title)
248. {
249. bookId = id;
250. **this**.details = details;
251. **this**.title = title;
252. }
254. **public** **int** getId()
255. {
256. **return** bookId;
257. }
259. **public** **void** setId(**int** id)
260. {
261. bookId = id;
262. }
264. **public** String getDetails()
265. {
266. **return** details;
267. }
269. **public** **void** setDetails(String details)
270. {
271. **this**.details = details;
272. }
274. **public** String getTitle()
275. {
276. **return** title;
277. }
279. **public** **void** setTitle(String title)
280. {
281. **this**.title = title;
282. }
283. }
285. /\*
286. \* This class represents the User which is a simple POJO
287. \*/
289. **class** User {
290. **private** **int** userId;
291. **private** String name;
292. **private** String details;
294. **public** **void** renewMembership()
295. {
296. }
298. **public** User(**int** id, String details, String name)
299. {
300. **this**.userId = id;
301. **this**.details = details;
302. **this**.name = name;
303. }
305. **public** **int** getId()
306. {
307. **return** userId;
308. }
310. **public** **void** setId(**int** id)
311. {
312. userId = id;
313. }
315. **public** String getDetails()
316. {
317. **return** details;
318. }
320. **public** **void** setDetails(String details)
321. {
322. **this**.details = details;
323. }
325. **public** String getName()
326. {
327. **return** name;
328. }
330. **public** **void** setName(String name)
331. {
332. **this**.name = name;
333. }
334. }
336. // This class is used to test the Application
338. **public** **class** AppTest {
340. **public** **static** **void** main(String[] args)
341. {
343. OnlineReaderSystem onlineReaderSystem = **new** OnlineReaderSystem();
345. Book dsBook = **new** Book(1, "It contains Data Structures", "Ds");
346. Book algoBook = **new** Book(2, "It contains Algorithms", "Algo");
348. onlineReaderSystem.getLibrary().addBook(dsBook);
349. onlineReaderSystem.getLibrary().addBook(algoBook);
351. User user1 = **new** User(1, " ", "Ram");
352. User user2 = **new** User(2, " ", "Gopal");
354. onlineReaderSystem.getUserManager().addUser(user1);
355. onlineReaderSystem.getUserManager().addUser(user2);
357. onlineReaderSystem.setActiveBook(algoBook);
358. onlineReaderSystem.setActiveUser(user1);
360. onlineReaderSystem.getDisplay().turnPageForward();
361. onlineReaderSystem.getDisplay().turnPageForward();
362. onlineReaderSystem.getDisplay().turnPageBackward();
363. }
364. }