**Design Patterns**

1. **What is SOLID design principle?**

**Single Responsibility**

Let's kick things off with the single responsibility principle. As we might expect, this principle states that **a class should only have one responsibility. Furthermore, it should only have one reason to change.**

**How does this principle help us to build better software?** Let's see a few of its benefits:

1. **Testing** – A class with one responsibility will have far fewer test cases
2. **Lower coupling** – Less functionality in a single class will have fewer dependencies
3. **Organization** – Smaller, well-organized classes are easier to search than monolithic ones

Take, for example, a class to represent a simple book:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class Book {        private String name;      private String author;      private String text;        //constructor, getters and setters  } |

In this code, we store the name, author, and text associated with an instance of a *Book*.

Let's now add a couple of methods to query the text:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Book {        private String name;      private String author;      private String text;        //constructor, getters and setters        // methods that directly relate to the book properties      public String replaceWordInText(String word){          return text.replaceAll(word, text);      }        public booleanisWordInText(String word){          return text.contains(word);      }  } |

Now, our *Book* class works well, and we can store as many books as we like in our application. But, what good is storing the information if we can't output the text to our console and read it?

Let's throw caution to the wind and add a print method:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Book {      //...        void printTextToConsole(){          // our code for formatting and printing the text      }  } |

This code does, however, violate the single responsibility principle we outlined earlier. To fix our mess, we should implement a separate class that is concerned only with printing our texts:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class BookPrinter {        // methods for outputting text      void printTextToConsole(String text){          //our code for formatting and printing the text      }        void printTextToAnotherMedium(String text){          // code for writing to any other location..      }  } |

Awesome. Not only have we developed a class that relieves the *Book*of its printing duties, but we can also leverage our *BookPrinter*class to send our text to other media.

Whether it's email, logging, or anything else, we have a separate class dedicated to this one concern.

**4. Open for Extension, Closed for Modification**

Now, time for the ‘O' – more formally known as the **open-closed principle**. Simply put, **classes should be open for extension, but closed for modification.** **In doing so, we** **stop ourselves from modifying existing code and causing potential new bugs** in an otherwise happy application.

Of course, the **one exception to the rule is when fixing bugs in existing code.**

Let's explore the concept further with a quick code example. As part of a new project, imagine we've implemented a *Guitar*class.

It's fully fledged and even has a volume knob:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class Guitar {        private String make;      private String model;      private int volume;        //Constructors, getters & setters  } |

We launch the application, and everyone loves it. However, after a few months, we decide the *Guitar*is a little bit boring and could do with an awesome flame pattern to make it look a bit more ‘rock and roll'.

At this point, it might be tempting to just open up the *Guitar*class and add a flame pattern – but who knows what errors that might throw up in our application.

Instead, let's **stick to the open-closed principle and simply extend our *Guitar*class**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class SuperCoolGuitarWithFlames extends Guitar {        private String flameColor;        //constructor, getters + setters  } |

By extending the *Guitar*class we can be sure that our existing application won't be affected.

**5. Liskov Substitution**

Next up on our list is Liskov substitution, which is arguably the most complex of the 5 principles. Simply put, **if class *A* is a subtype of class *B*, then we should be able to replace *B*with *A*without disrupting the behavior of our program.**

Let's just jump straight to the code to help wrap our heads around this concept:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface Car {        void turnOnEngine();      void accelerate();  } |

Above, we define a simple *Car*interface with a couple of methods that all cars should be able to fulfill – turning on the engine, and accelerating forward.

Let's implement our interface and provide some code for the methods:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class MotorCar implements Car {        private Engine engine;        //Constructors, getters + setters        public void turnOnEngine() {          //turn on the engine!          engine.on();      }        public void accelerate() {          //move forward!          engine.powerOn(1000);      }  } |

As our code describes, we have an engine that we can turn on, and we can increase the power. But wait, its 2019, and Elon Musk has been a busy man.

We are now living in the era of electric cars:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class ElectricCar implements Car {        public void turnOnEngine() {          throw new AssertionError("I don't have an engine!");      }        public void accelerate() {          //this acceleration is crazy!      }  } |

By throwing a car without an engine into the mix, we are inherently changing the behavior of our program. This is**a blatant violation of Liskov substitution and is a bit harder to fix than our previous 2 principles**.

One possible solution would be to rework our model into interfaces that take into account the engine-less state of our *Car*.

**6. Interface Segregation**

The ‘I ‘ in SOLID stands for interface segregation, and it simply means that **larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.**

For this example, we're going to try our hands as zookeepers. And more specifically, we'll be working in the bear enclosure.

Let's start with an interface that outlines our roles as a bear keeper:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface BearKeeper {      void washTheBear();      void feedTheBear();      void petTheBear();  } |

As avid zookeepers, we're more than happy to wash and feed our beloved bears. However, we're all too aware of the dangers of petting them. Unfortunately, our interface is rather large, and we have no choice than to implement the code to pet the bear.

Let's **fix this by splitting our large interface into 3 separate ones:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public interface BearCleaner {      void washTheBear();  }    public interface BearFeeder {      void feedTheBear();  }    public interface BearPetter {      void petTheBear();  } |

Now, thanks to interface segregation, we're free to implement only the methods that matter to us:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class BearCarer implements BearCleaner, BearFeeder {        public void washTheBear() {          //I think we missed a spot...      }        public void feedTheBear() {          //Tuna Tuesdays...      }  } |

And finally, we can leave the dangerous stuff to the crazy people:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class CrazyPerson implements BearPetter {        public void petTheBear() {          //Good luck with that!      }  } |

Going further, we could even split our [*BookPrinter*](https://www.baeldung.com/solid-principles#s)class from our example earlier to use interface segregation in the same way. By implementing a *Printer* interface with a single *print*method, we could instantiate separate *ConsoleBookPrinter*and *OtherMediaBookPrinter*classes.

**7. Dependency Inversion**

**The principle of Dependency Inversion refers to the decoupling of software modules. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.**

To demonstrate this, let's go old-school and bring to life a Windows 98 computer with code:

|  |  |
| --- | --- |
| 1 | public class Windows98Machine {} |

But what good is a computer without a monitor and keyboard? Let's add one of each to our constructor so that every *Windows98Computer*we instantiate comes pre-packed with a *Monitor*and a *StandardKeyboard*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class Windows98Machine {        private final StandardKeyboard keyboard;      private final Monitor monitor;        public Windows98Machine() {          monitor = new Monitor();          keyboard = new StandardKeyboard();      }    } |

This code will work, and we'll be able to use the *StandardKeyboard* and *Monitor* freely within our *Windows98Computer*class. Problem solved? Not quite. **By declaring the *StandardKeyboard*and *Monitor*with the *new*keyword, we've tightly coupled these 3 classes together.**

Not only does this make our *Windows98Computer*hard to test, but we've also lost the ability to switch out our *StandardKeyboard*class with a different one should the need arise. And we're stuck with our *Monitor* class, too.

Let's decouple our machine from the *StandardKeyboard* by adding a more general *Keyboard* interface and using this in our class:

|  |  |
| --- | --- |
| 1 | public interface Keyboard { } |
| 1  2  3  4  5  6  7  8  9  10 | public class Windows98Machine{        private final Keyboard keyboard;      private final Monitor monitor;        public Windows98Machine(Keyboard keyboard, Monitor monitor) {          this.keyboard = keyboard;          this.monitor = monitor;      }  } |

Here, we're using the dependency injection pattern here to facilitate adding the *Keyboard* dependency into the *Windows98Machine* class.

Let's also modify our *StandardKeyboard* class to implement the *Keyboard* interface so that it's suitable for injecting into the *Windows98Machine* class:

|  |  |
| --- | --- |
|  | public class StandardKeyboard implements Keyboard { } |

Now our classes are decoupled and communicate through the *Keyboard* abstraction. If we want, we can easily switch out the type of keyboard in our machine with a different implementation of the interface. We can follow the same principle for the *Monitor* class.

Excellent! We've decoupled the dependencies and are free to test our *Windows98Machine*with whichever testing framework we choose

1. **What are the different categories of Design Patterns used in Object Oriented Design?**

In Object Oriented design mainly three categories of design patterns are used. These categories are:

* + - Creational Design Patterns:



* + - Builder



* + - Factory Method



* + - Abstract Factory



* + - Object Pool



* + - Singleton



* + - Prototype



* + - Structural Design Patterns:



* + - Adapter



* + - Bridge



* + - Façade



* + - Decorator



* + - Composite



* + - Flyweight



* + - Proxy



* + - Behavioural Design Patterns:



* + - Command



* + - Iterator



* + - Chain of Responsibility



* + - Observer



* + - State



* + - Strategy



* + - Mediator



* + - Interpreter 

1. **When will you use Strategy Design Pattern in Java?**

Strategy pattern is very useful for implementing a family of algorithms. It is a behavioural design pattern.

With Strategy pattern we can select the algorithm at runtime. We can use it to select the sorting strategy for data. We can use it to save files in different formats like- .txt, .csv, .jpg etc.

In Strategy pattern we create an abstraction, which is an interface through which clients interact with our system. Behind the abstraction we create multiple implementation of same interface with different algorithms.

For a client, at runtime we can vary the algorithm based on the type of request we have received.

So we use Strategy pattern to hide the algorithm implementation details from client.

In Java Collections.sort() method uses strategy design pattern.

1. **What is Observer design pattern?**

GoF Definition: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**Concept**

In this pattern, there are many observers (objects) which are observing a particular subject (object). Observers are basically interested and want to be notified when there is a change made inside that subject.

So, they register themselves to that subject. When they lose interest in the subject they simply unregister from the subject. Sometimes this model is also referred to as the Publisher-Subscriber model.

**Real-Life Example**

We can think about a celebrity who has many fans. Each of these fans wants to get all the latest updates of

his/her favourite celebrity. So, he/she can follow the celebrity as long as his/her interest persists. When he loses interest, he simply stops following that celebrity. Here we can think of the fan as an observer and the celebrity as a subject.

**Computer World Example**

In the world of computer science, consider a simple UI-based example, where this UI is connected with some database (or business logic). A user can execute some query through that UI and after searching the database, the result is reflected back in the UI. In most of the cases we segregate the UI with the database. If a change occurs in the database, the UI should be notified so that it can update its display according to the change.



**package** com.design.patterns.behavioural .observer;

**public** **interface** ISubject {

**void** register(Observer o);

**void** unregister(Observer o);

**void** notifyObservers();

}

**package** com.design.patterns.behavioural .observer;

**public** **class** Observer {

**public** **void** update() {

System.***out***.println("flag value changed in Subject");

}

}

**package** com.design.patterns.behavioural .observer;

**import** java.util.ArrayList;

**import** java.util.List;

**class** Subject **implements** ISubject {

List<Observer> observerList = **new** ArrayList<Observer>();

**private** **int** \_flag;

**public** **int** getFlag() {

**return** \_flag;

}

**public** **void** setFlag(**int** \_flag) {

**this**.\_flag = \_flag;

// flag value changed .So notify observer(s)

notifyObservers();

}

**public** **void** register(Observer o) {

observerList.add(o);

}

**public** **void** unregister(Observer o) {

observerList.remove(o);

}

**public** **void** notifyObservers() {

**for** (**int** i = 0; i < observerList.size(); i++) {

observerList.get(i).update();

}

}

}

1. **What are the examples of Observer design pattern in JDK?**

In JDK there are many places where Observer design pattern is used. Some of these are as follows:

1. java.util.Observer, java.util.Observable
2. javax.servlet.http.HttpSessionAttributeListener
3. javax.servlet.http.HttpSessionBindingListener
4. All implementations of java.util.EventListener, and also in Swing packages
5. javax.faces.event.PhaseListener

1. **How Strategy design pattern is different from State design pattern in Java?**

State design pattern is a behavioural design pattern that is use for defining the state machine for an object. Each state of an object is defined in a child class of State class. When different actions are taken on an Object, it can change its state.

Strategy pattern is also a behavioural pattern, but it is mainly used for defining multiple algorithms. With same action of a client, the algorithm to be used can change.

Some people consider State pattern similar to Strategy pattern, since an Object changes its Strategy with different method invocations. But the main difference is that in State pattern internal state of an Object is one of the determining factors for selecting the Strategy for change of state.

Where as in Strategy pattern, client can pass some external parameter in input during method invocation that determines the strategy to be used at run time.

Therefore State pattern is based on the Object’s internal state, where as Strategy pattern is based on Client’s invocation.

State pattern is very useful in increasing the maintainability of the code in a large code-base.

1. **Can you explain Decorator design pattern with an example in Java?**

**GoF Definition:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**Concept**

This main principle of this pattern says that we cannot modify existing functionalities but we can extend them. In other words, this pattern is open for extension but closed for modification. The core concept applies when we want to add some specific functionalities to some specific object instead of to the whole class.

**Real-Life Example**

Suppose you already own a house. Now you have decided to add an additional floor. Obviously, you do not want to change the architecture of ground floor (or existing floors). You may want to change the design of the architecture for the newly added floor without affecting the existing architecture for existing floor(s).

**Computer World Example**

Suppose in a GUI-based toolkit, we want to add some border properties. We can do this by inheritance. But it cannot be treated as the best solution because our user or client cannot have absolute control from the creation. The core of that choice is static there. Decorator can offer us a more flexible approach: here we may surround the component in another object. The enclosing object is termed "decorator." It conforms to the interface of the component it decorates. It forwards requests to the component. It can perform additional operations before or after those forwarding requests. An unlimited number of responsibilities can be added with this concept.



**package** com.design.patterns.structural.decorator;

**public** **abstract** **class** AbstractDecorator **extends** Component {

**protected** Component com;

**public** **void** setTheComponent(Component c) {

com = c;

}

**public** **void** doJob() {

**if** (com != **null**) {

com.doJob();

}

}

}

**package** com.design.patterns.structural.decorator;

**public** **abstract** **class** Component {

**public** **abstract** **void** doJob();

}

**package** com.design.patterns.structural.decorator;

**public** **class** ConcreteComponent **extends** Component {

@Override

**public** **void** doJob() {

System.***out***.println(" I am from Concrete Component-I am closed for modification.");

}

}

**package** com.design.patterns.structural.decorator;

**public** **class** ConcreteDecoratorEx\_1 **extends** AbstractDecorator {

**public** **void** doJob() {

**super**.doJob();

// Add additional thing if necessary

System.***out***.println("I am explicitly from Ex\_1");

}

}

**package** com.design.patterns.structural.decorator;

**class** ConcreteDecoratorEx\_2 **extends** AbstractDecorator {

**public** **void** doJob() {

System.***out***.println("");

System.***out***.println("\*\*\*START Ex-2\*\*\*");

**super**.doJob();

// Add additional thing if necessary

System.***out***.println("Explicitly From DecoratorEx\_2");

System.***out***.println("\*\*\*END. EX-2\*\*\*");

}

}

**package** com.design.patterns.structural.decorator;

**public** **class** DecoratorDemo {

**public** **static** **void** main(String[] args) {

System.***out***.println("\*\*\*Decorator pattern Demo\*\*\*");

ConcreteComponent cc = **new** ConcreteComponent();

ConcreteDecoratorEx\_1 cd\_1 = **new** ConcreteDecoratorEx\_1();

// Decorating ConcreteComponent Object //cc with ConcreteDecoratorEx\_1

cd\_1.setTheComponent(cc);

cd\_1.doJob();

ConcreteDecoratorEx\_2 cd\_2 = **new** ConcreteDecoratorEx\_2();

// Decorating ConcreteComponent Object //cc with ConcreteDecoratorEx\_1 &

// ConcreteDecoratorEX\_2

cd\_2.setTheComponent(cd\_1);// Adding //results from cd\_1 now

cd\_2.doJob();

}

}

Some people call Decorator pattern as Wrapper pattern as well. It is used to add the behaviour to an object, without changing the behaviour of other objects of same class.

One of the very good uses of Decorator pattern is in java.io package. We can have a FileInputStream to handle a File. To add Buffering behaviour we can decorate FileInputStream with BufferedInputStream. To add the gzip behaviour BufferedInputStream we can decorate it with GzipInputStream. To add serialization behavior to GzipInputStream, we can decorate it with ObjectInputStream.

1. **What are the main advantages of using decorator patterns?**

* Without disturbing existing objects in the system, we can add new functionality to a particular object.
* We can code incrementally. For example, we’ll make a simple class first and then one by one we can add decorator objects to them as needed. As a result, we do not need to take care of each and every possible scenario in the beginning. On the other hand, we must acknowledge that making a complex class first and then extending its functionality is a much more complex procedure.

1. **What is the major disadvantage of using this pattern?**

First of all, if we are careful enough, there is no significant disadvantage. But if we create too many decorators in the system, the system will be hard to maintain and debug. At the same time, the decorators can create unnecessary confusion.

1. **How is Decorator pattern different from inheritance?**

We can add or remove responsibilities by simply attaching or detaching decorators. But with the simple inheritance technique, we need to create a new class for new responsibilities. So, there will be many classes inside the system, which in turn can make the system complex.

1. **Explain the Composite Design Pattern**

GoF Definition: Compose objects into tree structures to represent part-whole hierarchies. The composite pattern lets clients treat individual objects and compositions of objects uniformly.

**Concept**

This pattern can show part-whole hierarchy among objects. A client can treat a composite object just like a single object. In object-oriented programming, we make a composite object when we have many objects with common functionalities. This relationship is also termed a “has-a” relationship among objects.

**Real-Life Example**

We can think of any organization that has many departments, and in turn each department has many employees to serve. Please note that actually all are employees of the organization. Groupings of employees create a department, and those departments ultimately can be grouped together to build the whole organization.

**Computer World Example**

Any tree structure in computer science can follow a similar concept.

**package** com.design.patterns.structural.composite;

**public** **class** Accountant **implements** Employee {

/\*

\* In this class,there are many methods which are not applicable to cashier

\* because it is a leaf node.

\*/

**private** **int** id;

**private** String name;

**private** **double** salary;

**public** Accountant(**int** id, String name, **double** salary) {

**this**.id = id;

**this**.name = name;

**this**.salary = salary;

}

@Override

**public** **void** add(Employee employee) {

// this is leaf node so this method is not applicable to this class.

}

@Override

**public** Employee getChild(**int** i) {

// this is leaf node so this method is not applicable to this class.

**return** **null**;

}

@Override

**public** **int** getId() {

// **TODO** Auto-generated method stub

**return** id;

}

@Override

**public** String getName() {

**return** name;

}

@Override

**public** **double** getSalary() {

**return** salary;

}

@Override

**public** **void** print() {

System.***out***.println("=========================");

System.***out***.println("Id =" + getId());

System.***out***.println("Name =" + getName());

System.***out***.println("Salary =" + getSalary());

System.***out***.println("=========================");

}

@Override

**public** **void** remove(Employee employee) {

// this is leaf node so this method is not applicable to this class.

}

}

**package** com.design.patterns.structural.composite;

// this is the BankManager class i.e. Composite.

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** BankManager **implements** Employee {

**private** **int** id;

**private** String name;

**private** **double** salary;

**public** BankManager(**int** id, String name, **double** salary) {

**this**.id = id;

**this**.name = name;

**this**.salary = salary;

}

List<Employee> employees = **new** ArrayList<Employee>();

@Override

**public** **void** add(Employee employee) {

employees.add(employee);

}

@Override

**public** Employee getChild(**int** i) {

**return** employees.get(i);

}

@Override

**public** **void** remove(Employee employee) {

employees.remove(employee);

}

@Override

**public** **int** getId() {

**return** id;

}

@Override

**public** String getName() {

**return** name;

}

@Override

**public** **double** getSalary() {

**return** salary;

}

@Override

**public** **void** print() {

System.***out***.println("==========================");

System.***out***.println("Id =" + getId());

System.***out***.println("Name =" + getName());

System.***out***.println("Salary =" + getSalary());

System.***out***.println("==========================");

Iterator<Employee> it = employees.iterator();

**while** (it.hasNext()) {

Employee employee = it.next();

employee.print();

}

}

}// End of the BankManager class.

**package** com.design.patterns.structural.composite;

**public** **class** Cashier **implements** Employee {

/\*

\* In this class,there are many methods which are not applicable to cashier

\* because it is a leaf node.

\*/

**private** **int** id;

**private** String name;

**private** **double** salary;

**public** Cashier(**int** id, String name, **double** salary) {

**this**.id = id;

**this**.name = name;

**this**.salary = salary;

}

@Override

**public** **void** add(Employee employee) {

// this is leaf node so this method is not applicable to this class.

}

@Override

**public** Employee getChild(**int** i) {

// this is leaf node so this method is not applicable to this class.

**return** **null**;

}

@Override

**public** **int** getId() {

// **TODO** Auto-generated method stub

**return** id;

}

@Override

**public** String getName() {

**return** name;

}

@Override

**public** **double** getSalary() {

**return** salary;

}

@Override

**public** **void** print() {

System.***out***.println("==========================");

System.***out***.println("Id =" + getId());

System.***out***.println("Name =" + getName());

System.***out***.println("Salary =" + getSalary());

System.***out***.println("==========================");

}

@Override

**public** **void** remove(Employee employee) {

// this is leaf node so this method is not applicable to this class.

}

}

**package** com.design.patterns.structural.composite;

**public** **class** CompositePatternDemo {

**public** **static** **void** main(String[] args) {

Employee emp1 = **new** Cashier(101, "Sohan Kumar", 20000.0);

Employee emp2 = **new** Cashier(102, "Mohan Kumar", 25000.0);

Employee emp3 = **new** Accountant(103, "Seema Mahiwal", 30000.0);

Employee manager1 = **new** BankManager(100, "Aayush", 100000.0);

manager1.add(emp1);

manager1.add(emp2);

manager1.add(emp3);

manager1.print();

}

}

* The pattern is ideal to represent the part-whole hierarchy among objects.
* Here the client can treat composition of objects like a single object.
* Clients can add new types of component easily.
* If we are forced to maintain child ordering (e.g., parse trees as components), we need to take special care to maintain that order.

***What is the best data structure to store components of Composite?***

There is no universal rule. It depends on the requirement (e.g., efficiency). We can use linked list, trees, arrays, etc., based on our demand. GoF also suggests that it is not mandatory to use any general-purpose data structure.

1. **What is a good scenario for using Composite design Pattern in Java?**

Some of the good scenarios where Composite design pattern can be used are as follows:

Tree Structure: The most common use of Composite design pattern is Tree structure. If you want to represent data in a Tree data structure, Composite pattern can be used.

E.g. In an Organization, to a Manager has Employees. But Manager is also an Employee. If we start from CEO level, there is one big tree for the whole organization structure. Under that big tree there are many sub-trees. This can be easily represented with Composite design pattern.

Recursion: Another use of Composite design pattern is Recursion. If we have a Recursion based algorithm, we need data to be passed to algorithm in a data structure that treats individual objects and compositions at each level of recursion uniformly.

E.g. To implement a recursive Polynomial Solving algorithm, we can use Composite design pattern to store the intermediate results.

Graphics: Another good use of Composite design pattern is in Graphics. We can group shapes inside a composite and make higher-level groups of smaller groups of shapes to complete the

1. **What is a Singleton class?**

GoF Definition: Ensure a class only has one instance, and provide a global point of access to it.

**Concept**

A particular class should have only one instance. We will use only that instance whenever we are in need.

**Real-Life Example**

Suppose you are a member of a cricket team. And in a tournament your team is going to play against another team. As per the rules of the game, the captain of each side must go for a toss to decide which side will bat (or bowl) first. So, if your team does not have a captain, you need to elect someone as a captain first. And at the same time, your team cannot have more than one captain.

**Computer World Example**

In a software system sometimes we may decide to use only one file system. Usually we may use it for the centralized management of resources.

**package** com.design.patterns.creational.singleton;

**public** **class** SingletonBillPlough {

**private** SingletonBillPlough() {

}

**private** **static** **class** StaticHolder {

**private** **static** SingletonBillPlough *INSTANCE* = **new** SingletonBillPlough();

}

**public** **static** SingletonBillPlough getInstance() {

**return** StaticHolder.*INSTANCE*;

}

}

**package** com.design.patterns.creational.singleton;

**public** **class** SingletonBillPloughDemo {

**public** **static** **void** main(String[] args) {

SingletonBillPlough inst1 = SingletonBillPlough.*getInstance*();

SingletonBillPlough inst2 = SingletonBillPlough.*getInstance*();

SingletonBillPlough inst3 = SingletonBillPlough.*getInstance*();

System.***out***.println(inst1);

System.***out***.println(inst2);

System.***out***.println(inst3);

}

}

**Singleton using Enum:**

**package** com.design.patterns.creational.singleton;

**public** **enum** SingletonEnum {

***INSTANCE***;

**int** value;

**public** **void** process(String desc) {

System.***out***.println("Value:" + desc + " " + value);

}

}

**package** com.design.patterns.creational.singleton;

**public** **class** SingletonEnumDemo {

**public** **static** **void** main(String[] args) {

SingletonEnum singleton = SingletonEnum.***INSTANCE***;

singleton.process("hello");

}

}

**Problems With Reflection**

An advanced user can change the private access modifier of the constructor to anything they want at runtime using reflection. If this happens, our only mechanism for non-instantiability breaks. Let's see how this can be done.

public class ReflectionDemo {

public static void main(String[] args) throws Exception {

Singleton singleton = Singleton.INSTANCE;

Constructor constructor = singleton.getClass().getDeclaredConstructor(new Class[0]);

constructor.setAccessible(true);

Singleton singleton2 = (Singleton) constructor.newInstance();

if (singleton == singleton2) {

System.out.println("Two objects are same");

} else {

System.out.println("Two objects are not same");

}

singleton.setValue(1);

singleton2.setValue(2);

System.out.println(singleton.getValue());

System.out.println(singleton2.getValue());

}

}

Output:

Two objects are not same

1

2

In this way, the non-accessible private constructor becomes accessible and the whole idea of making the class a singleton breaks.

## Problems With Serialization and Deserialization

In order to serialize the above singleton classes, we must implement those classes with a Serializable interface. But doing that is not enough. When deserializing a class, new instances are created. Now it doesn't matter the constructor is private or not. Now there can be more than one instance of the same singleton class inside the JVM, violating the singleton property.

public class SerializeDemo implements Serializable {

public static void main(String[] args) {

Singleton singleton = Singleton.INSTANCE;

singleton.setValue(1);

// Serialize

try {

FileOutputStream fileOut = new FileOutputStream("out.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut);

out.writeObject(singleton);

out.close();

fileOut.close();

} catch (IOException e) {

e.printStackTrace();

}

singleton.setValue(2);

// Deserialize

Singleton singleton2 = null;

try {

FileInputStream fileIn = new FileInputStream("out.ser");

ObjectInputStream in = new ObjectInputStream(fileIn);

singleton2 = (Singleton) in.readObject();

in.close();

fileIn.close();

} catch (IOException i) {

i.printStackTrace();

} catch (ClassNotFoundException c) {

System.out.println("singletons.SingletonEnum class not found");

c.printStackTrace();

}

if (singleton == singleton2) {

System.out.println("Two objects are same");

} else {

System.out.println("Two objects are not same");

}

System.out.println(singleton.getValue());

System.out.println(singleton2.getValue());

}

}

The output for the above code is:

Two objects are not same

2

1

Here, singleton and singleton2 are two different instances having two different values as their field variables. This violates the singleton property. The solution is that we have to implement the readResolve method, which is called when preparing the deserialized object before returning it to the caller. The solution is as follows.

public class Singleton implements Serializable{

public static final Singleton INSTANCE = new Singleton();

private Singleton() {

}

protected Object readResolve() {

return INSTANCE;

}

}

Now the output for above code will be:

Two objects are same

2

2

Now the singleton property is preserved, and only one instance of the singleton class exists within the JVM

1. **What is the difference between Singleton class and Static class?**

A static class in Java has only static methods. It is a container of functions. It is created based on procedural programming design.

Singleton class is a pattern in Object Oriented Design. A Singleton class has only one instance of an object in JVM. This pattern is implemented in such a way that there is always only one instance of that class present in JVM.

1. **Have you used Singleton design pattern in your Java project?**

Yes. Singleton is one of the most popular design patterns in enterprise level Java applications. Almost in every project we see some implementation of Singleton.

With Singleton pattern we can be sure that there is only one instance of a class at any time in the application.

This helps in storing properties that have to be used in the application in a unique location.

1. **What are the main uses of Singleton design pattern in Java project?**

Some of the main uses of Singleton design pattern in Java are as follows:

1. Runtime: In JDK, java.lang.Runtime is a singleton-based class. There is only one instance of Runtime in an application. This is the only class that interfaces with the environment/machine in which Java process is running.
2. Enum: In Java, enum construct is also based on Singleton pattern. Enum values can be accessed globally in same way by all classes.
3. Properties: In an application it makes sense to keep only one copy of the properties that all classes can access. This can be achieved by making properties class Singleton so that every class gets same copy of properties.
4. Spring: In Spring framework, all the beans are by default Singleton per container. So there is only one instance of bean in a Spring IoC container. But Spring also provides options to make the scope of a bean prototype in a container.
5. **Why java.lang.Runtime is a Singleton in Java?**

In Java, java.lang.Runtime is implemented on Singleton design pattern.

Runtime is the class that acts as an interface with the environment in which Java process is running. Runtime contains methods that can interact with the environment.

Like- totalmemory() method gives the total memory in JVM. maxMemory() method gives the maximum memory that JVM can use.

There is an exit() method to exit the Java process. We do not want multiple objects in JVM to have exit() method.

Similarly there is gc() method that can run the Garbage Collector. With only one copy of gc() method, we can ensure that no other object can run the Garbage Collector when one instance of GC is already running.

Due to all these reasons there is only one copy of Runtime in Java. To ensure single copy of Runtime, it is implemented as a Singleton in Java.

1. **What is the way to implement a thread-safe Singleton design pattern in Java?**

In Java there are many options to implement a thread-safe Singleton pattern. Some of these are as follows:

1. Double Checked Locking: This is the most popular method to implement Singleton in Java. It is based on Lazy Initialization. In this we first check the criteria for locking before acquiring a lock to create an object. In Java we use it with volatile keyword.

Sample code:

class DoubleCheckSingleton {

private volatile HelloSingletonhelloSingleton; // Use Volatile

public HelloSingletongetHelloSingleton() {

HelloSingleton result = helloSingleton;

if (result == null) {

synchronized(this) { // Synchronize for thread safety result = helloSingleton;

if (result == null) {

result = new HelloSingleton();

helloSingleton = result;

}

}

}

return result;

}

}

1. Bill Pugh Singleton: We can also use the method by Bill Pugh for implementing Singleton in Java. In this we use an Inner Static class to create the Singleton instance.

Sample code:

public class SingletonBillPugh {

* Inner class that holds instance private static class InnerSingleton{

private static final SingletonBillPugh INSTANCE = new

SingletonBillPugh();

}

// Private constructor

private SingletonBillPugh(){}

public static SingletonBillPughgetInstance(){ return InnerSingleton.INSTANCE;

}

}

When first time SingletonBillPugh is loaded in memory, InnerSingleton is not loaded. Only when getInstance() method is called, InnerSingleton class is loaded and an Instance is created.

1. Enum: We can also use Java enum to create thread-safe implementation. Java enum values are accessible globally so these can be used as a Singleton.

Sample Code:

public enumSingletonEnum {

INSTANCE;

public static void doImplementation(){

…..

}

}

1. **What is Template Method design pattern in Java?**

**GoF Definition:** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. The template method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

**Concept**

In a template method, we define the minimum or essential structure of an algorithm. Then we defer some functionalities (responsibilities) to the subclasses. As a result, we can redefine certain steps of an algorithm by keeping the key structure fixed for that algorithm.

**Real-Life Example**

Suppose we want to make pizza. The basic mechanism is the same, but extra materials are added based upon the customer’s choice—whether he/she wants a vegetarian pizza or a non-vegetarian pizza.

**Computer World Example**

For an engineering student, in general, most of the subjects in the first semester are common for all concentrations. Later, additional papers are added in his/her course based on his/her specialization (Computer Science, Electronics, etc.).   


**package** com.design.patterns.behavioural.template;

**public** **abstract** **class** BasicEngineering {

// papers() in the template method

**public** **void** papers() {

// Common Papers:

math();

softSkills();

// Specialized Paper:

specialPaper();

}

//Non-Abstract method math(), softSkills() are //already implemented by Template class

**private** **void** math() {

System.***out***.println("Mathematics");

}

**private** **void** softSkills() {

System.***out***.println("SoftSkills");

}

//Abstract method SpecialPaper() must be implemented in derived classes

**public** **abstract** **void** specialPaper();

}

**package** com.design.patterns.behavioural.template;

**public** **class** ComputerScience **extends** BasicEngineering {

@Override

**public** **void** specialPaper() {

System.***out***.println("Object Oriented Programming");

}

}

**package** com.design.patterns.behavioural.template;

**public** **class** Electronics **extends** BasicEngineering {

@Override

**public** **void** specialPaper() {

System.***out***.println("Digital Logic and Circuit Theory");

}

}

**package** com.design.patterns.behavioural.template;

**class** TemplateMethodPatternEx {

**public** **static** **void** main(String[] args) {

System.***out***.println("\*\*\*Template Method Pattern Demo\*\*\*\n");

BasicEngineering bs = **new** ComputerScience();

System.***out***.println("Computer Sc Papers:");

bs.papers();

System.***out***.println();

bs = **new** Electronics();

System.***out***.println("Electronics Papers:");

bs.papers();

}

}

1. **What are the examples of Template method design pattern in JDK?**

In JDK there are many places where Template method design pattern is used. Some of these are as follows:

1. In Java Abstract Collection classes like

java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap implement a template for their corresponding Collection.

1. javax.servlet.http.HttpServlet: In the HttpServlet class all the doGet(), doPost() etc. methods send a HTTP 405 "Method Not Allowed" error to the response. This error response is like a Template that can be further customized for each of these methods.
2. In java.io package there are Stream and Writer classes like java.io.InputStream, java.io.OutputStream, java.io.Reader and java.io.Writer that provide non-abstract methods. These methods are implementation of Template method design pattern.

1. **Can you tell some examples of Factory Method design pattern implementation in Java?**

Factory Method pattern is a creational design pattern. A Factory is an object that is used to create more objects.

In general, a Factory object has methods that can be used to create a type of objects. Some people call it Factory Method design pattern as well.

Some of the examples of Factory Method pattern in JDK are:

Java.lang.Class.forName()



java.net.URLStreamHandlerFactory.createURLStreamHan



java.util.Calendar.getInstance()



java.util.ResourceBundle.getBundle()



java.text.NumberFormat.getInstance()



java.nio.charset.Charset.forName()



java.util.EnumSet.of()



javax.xml.bind.JAXBContext.createMarshaller()

1. **What is the benefit we get byusing static factory method to create object?**

By using Static Factory Method we encapsulate the creation process of an object. We can use new() to create an Object from its constructor. Instead we use static method of a Factory to create the object. One main advantage of using Factory is that Factory can choose the correct implementation at runtime and create the right object. The caller of method can specify the desired behavior.

E.g. If we have a ShapeFactory with createShape(String type) method. Client can call ShapeFactory.createShape(“Circle”) to get a circular shape. ShapeFactory.createShape(“Square”) will return square shape. In this way, ShapeFactory knows how to create different shapes based on the input by caller.

Another use of Factory is in providing access to limited resources to a large set of users.

E.g. In ConnectionPool, we can limit the total number of connections that can be created as well as we can hide the implementation details of creating connection. Here ConnectionPool

is the factory. Clients call static method ConnectionPool.getConnection().

1. **What are the examples of Builder design pattern in JDK?**

In JDK there are many places where Builder design pattern is used.

Some of these are as follows:

1. java.lang.StringBuilder.append(): StringBuilder is based on Builder pattern.
2. java.nio.IntBuffer.put(): Invocation of put() method return IntBuffer. Also there are many variants of this method to build the IntBuffer.
3. javax.swing.GroupLayout.Group.addComponent(): We can use addComponent() method to build a UI that can contain multiple levels of components.
4. java.lang.Appendable
5. java.lang.StringBuffer.append(): StringBuffer is similar to StringBuilder and it is also based on Builder design pattern.
6. **What are the examples of Abstract Factory design pattern in JDK?**

In JDK there are many places where Abstract Factory design pattern is used. Some of these are as follows:

javax.xml.xpath.XPathFactory.newInstance()



javax.xml.parsers.DocumentBuilderFactory.newInstance()



javax.xml.transform.TransformerFactory.newInstance()

1. **What are the examples of Decorator design pattern in JDK?**

In JDK there are many places where Decorator design pattern is used. Some of these are as follows:

1. In java.io package many classes use Decorator pattern. Subclasses of java.io.InputStream, OutputStream, Reader and Writer have a constructor that can take the instance of same type and decorate it with additional behavior.
2. In java.util.Collections, there are methods like

checkedCollection(),checkedList(),checkedMap(),

synchronizedList(), synchronizedMap(), synchronizedSet(), unmodifiableSet(), unmodifiableMap() and unmodifiableList() methods that can decorate an object and return the same type.

1. In javax.servlet package, there are classes like

javax.servlet.http.HttpServletRequestWrapper and HttpServletResponseWrapper that are based on Decorator design pattern.

1. **What are the examples of Proxy design pattern in JDK?**

Proxy design pattern provides an extra level of indirection for providing access to another object. It can also protect a real object from any extra level of complexity.

In JDK there are many places where Proxy design pattern is used.

Some of these are as follows:

java.lang.reflect.Proxy



java.rmi.\*



javax.inject.Inject



javax.ejb.EJB



javax.persistence.PersistenceContext

1. **What are the examples of Chain of Responsibility design pattern in JDK?**

In JDK there are many places where Chain of Responsibility design pattern is used. Some of these are as follows:

1. java.util.logging.Logger.log(): In this case Logger class provides multiple variations of log() method that can take the responsibility of logging from client in different scenarios. The client has to just call the appropriate log() method and Logger will take care of these commands.
2. javax.servlet.Filter.doFilter(): In the Filter class, the Container calls the doFilter method when a request/response pair is passed through the chain. With filter the request reaches to the appropriate resource at the end of the chain. We can pass FilterChain in doFilter() method to allow the Filter to pass on the request and response to the next level in the chain.
3. **What are the main uses of Command design pattern?**

Command design pattern is a behavioural design pattern. We use it to encapsulate all the information required to trigger an event. Some of the main uses of Command pattern are:

1. Graphic User Interface (GUI): In GUI and menu items, we use command pattern. By clicking a button we can read the current information of GUI and take an action.
2. Macro Recording: If each of user action is implemented as a separate Command, we can record all the user actions in a Macro as a series of Commands. We can use this series to implement the “Playback” feature. In this way, Macro can keep on doing same set of actions with each replay.
3. Multi-step Undo: When each step is recorded as a Command, we can use it to implement Undo feature in which each step can by undo. It is used in text editors like MS-Word.
4. Networking: We can also send a complete Command over the network to a remote machine where all the actions encapsulated within a Command are executed.
5. Progress Bar: We can implement an installation routine as a series of Commands. Each Command provides the estimate time. When we execute the installation routine, with each command we can display the progress bar.
6. Wizard: In a wizard flow we can implement steps as Commands. Each step may have complex task that is just implemented within one command.
7. Transactions: In a transactional behavior code there are multiple tasks/updates. When all the tasks are done then only transaction is committed. Else we have to rollback the transaction. In such a scenario each step is implemented as separate Command.
8. **What are the examples of Command design pattern in JDK?**

In JDK there are many places where Command design pattern is used. Some of these are as follows:

All implementations of java.lang.Runnable

All implementations of javax.swing.Action

1. **What are the examples of Interpreter design pattern in JDK?**

Interpreter design pattern is used to evaluate sentences in a language. E.g. In SQL we can use it to evaluate a query by evaluating each keyword like SELECT, FROM, WHERE clause.

In an Interpreter implementation there is a class for each keyword/symbol. A sentence is just a composite of these keywords. But the sentence is represented by Syntax tree that can be interpreted.

In JDK there are many places where Interpreter design pattern is used. Some of these are as follows:

java.util.Pattern



java.text.Normalizer



Subclasses of java.text.Format: DateFormat, MessageFormat, NumberFormat



Subclasses of javax.el.ELResolver: ArrayELResolver, MapELResolver, CompositeELResolver etc.

1. **What are the examples of Mediator design pattern in JDK?**

By using Mediator pattern we can decouple the multiple objects that interact with each other. With a Mediator object we can create many-to-many relationships in multiple objects.

In JDK there are many places where Mediator design pattern is used. Some of these are as follows:

java.util.Timer: schedule() methods in Timer class act as Mediator between the clients and the TimerTask to be scheduled.



java.util.concurrent.Executor.execute(): The execute() method in an Executor class acts as a Mediator to execute the different tasks.



java.util.concurrent.ExecutorService



java.lang.reflect.Method.invoke(): In Method class of reflection package, invoke() method acts as a Mediator.



java.util.concurrent.ScheduledExecutorService: Here also schedule() method and its variants are Mediator pattern implementations.

1. **What are the examples of Strategy design pattern in JDK?**

In JDK there are many places where Strategy design pattern is used.

Some of these are as follows:

1. java.util.Comparator: In a Comparator we can use compare() method to change the strategy used by Collections.sort() method.
2. javax.servlet.http.HttpServlet: In a HttpServlet class service() and doGet(), doPost() etc. methods take HttpServletRequest and HttpServletResponse and the implementor of Servlet processes it based on the strategy it selects.
3. **What are the examples of Visitor design pattern in JDK?**

By using Visitor design pattern we can add new virtual methods to existing classes without modifying their core structure.

In JDK there are many places where Visitor design pattern is used.

Some of these are as follows:

javax.lang.model.element.AnnotationValue and AnnotationValueVisitor



java.nio.file.FileVisitor and SimpleFileVisitor



javax.lang.model.type.TypeMirror and TypeVisitor



javax.lang.model.element.Element and ElementVisitor



javax.faces.component.visit.VisitContext and VisitCallback

1. **How Decorator design pattern is different from Proxy pattern?**

Main differences between Decorator and Proxy design pattern are:

Decorator provides an enhanced interface after decorating it with additional features. Proxy provides same interface since it is just acting as a proxy to another object.



Decorator is a type of Composite pattern with only one component. But each decorator can add additional features. Since it is one component in Decorator, there is no object aggregation.



Proxy can also provide performance improvement by lazy loading. There is nothing like this available in Decorator.



Decorator follows recursive composition. Proxy is just one object to another object access.



Decorator is mostly used for building a variety of objects.



Proxy is mainly used for access to another object.

1. **What are the different scenarios to use Setter and Constructor based injection in Dependency Injection (DI) design pattern?**

We use Setter injection to provide optional dependencies of an object. Constructor injection is used to provide mandatory dependency of an object.

In Spring IoC, Dependency Injection is heavily used. There we have to differentiate between the scenario suitable for Setter based and Constructor based dependency injection.

1. **What are the different scenarios for using Proxy design pattern?**

Proxy design pattern can be used in a wide variety of scenario in

Java. Some of these are as follows:

1. Virtual Proxy: This is a virtual object that acts as a proxy for objects that are very expensive to create. It is used in Lazy Loading. When client makes the first request, the real object is created.
2. Remote Proxy: This is a local object that provides access to a remote object. It is generally used in Remote Method Invocation (RMI) and Remote Procedure Call (RPC). It is also known as a Stub.
3. Protective Proxy: This is an object that control the access to a Master object. It can authenticate and authorize the client for accessing the Master object. If client has right permissions, it allows client to access the main object.
4. Smart Proxy: It is an object that can add additional information to the main object. It can track the number of other objects accessing the main object. It can track the different clients from where request is coming. It can even deny access to an object if the number of requests is greater than a threshold.

1. **What is the main difference between Adapter and Proxy design pattern?**

Adapter pattern provides a different interface to an object. But the Proxy always provides same interface to the object.

Adapter is like providing an interface suitable to client’s use. But Proxy is same interface that has additional feature or check.

E.g. In electrical appliances we use Adapter to convert from one type of socket to another type of socket. In case of proxy, we have a plug with built-in surge protector. The interface for plug and the original device remains same.

1. **When will you use Adapter design pattern in Java?**

If we have two classes with incompatible interfaces, we use Adapter pattern to make it work. We create an Adapter object that can adapt the interface of one class to another class.

It is generally used for working with third party libraries. We create an Adapter class between third party code and our class. In case of any change in third party code we have to just change the Adapter code. Rest of our code can remain same and just take to Adapter.

1. **What are the examples of Adapter design pattern in JDK?**

In JDK there are many places where Adapter design pattern is used.

Some of these are as follows:

java.util.Arrays.asList(): This method can adapt an Array to work as a List.



java.util.Collections.list(): This method can adapt any collection to provide List behavior.



java.util.Collections.enumeration(): This method returns an enumeration over the collection.



java.io.InputStreamReader(InputStream): This method adapts a Stream to Reader class.



java.io.OutputStreamWriter(OutputStream): This method adapts an OutputStream to Writer class.



javax.xml.bind.annotation.adapters.XmlAdapter.marshal()

1. **What is the difference between Factory and Abstract Factory design pattern?**

With Factory design pattern we can create concrete products of a type that Factory can manufacture. E.g. If it is CarFactory, we can produce, Ford, Toyota, Honda, Maserati etc.

With Abstract Factory design pattern we create a concrete implementation of a Factory. E.g. DeviceFactory can be Abstract and it can give us GoogleDeviceFactory, AppleDeviceFactory etc. With AppleDeviceFactory we will get products like- iPhone, iPad, Mac etc. With GoogleDeviceFactory we will get products like-Nexus phone, Google Nexus tablet, Google ChromeBook etc.

So it is a subtle difference between Factory and Abstract Factory design pattern. One way to remember is that within Abstract Factory pattern, Factory pattern is already implemented.

1. **What is Builder design pattern?**

Builder design pattern is a creational design pattern. We can use Builder pattern to create complex objects with multiple options.

E.g. when we have to create a Meal in a restaurant we can use Builder pattern. We can keep adding options like- Starter, Drink, Main Course, and Dessert etc. to create complete meal. When a user selects other options of Starter, Drink Main Course, Dessert another type of meal is created.

Main feature of Builder pattern is step-by-step building of a complex object with multiple options.

1. **What is the design pattern suitable to access elements of a Collection?**

We can use Iterator design pattern to access the individual elements of a Collection. In case of an ordered collection we can get Iterator that returns the elements in an order.

In Java there are many implementation of Iterator in Collections package. We have iterators like- Spliterator, ListIterator etc. that implement Iterator pattern.

1. **How can we implement Producer Consumer design pattern in Java?**

We can use BlockingQueue in Java to implement Producer Consumer design pattern.

It is a concurrent design pattern.

1. **What design pattern is suitable to add new features to an existing object?**

We can use Decorator design pattern to add new features to an existing object. With a Decorator we work on same object and return the same object with more features. But the structure of the object remains same since all the decorated versions of object implement same interface.

1. **Which design pattern can be used when to decouple abstraction from the implementation?**

We can use Bridge design pattern to detach the implementation from the abstraction.

Bridge is mainly used for separation of concern in design. We can create an implementation and store it in the interface, which is an abstraction. Where as specific implementation of other features can be done in concrete classes that implement the interface.

Often Bridge design pattern is implemented by using Adapter pattern.

E.g. we have Shape interface. We want to make Square and Circle shapes. But further we want to make RedSquare, BlackSquare shapes and GreenCircle, WhiteCircle shapes. In this case rather than creating one hierarchy of all the shapes, we separate the Color concern from Shape hierarchy.

So we create two hierarchies. One is Shape to Square and Shape to Circle hierarchy. Another one is Color to Red, Black, Green, White hierarchy. In this way we can create multiple types of shapes with multiple colors with Bridge design pattern.

1. **Which is the design pattern used in Android applications?**

Android applications predominantly use Model View Presenter design pattern.

1. Model: This is the domain model of the Android application. It contains the business logic and business rules.
2. View: These are the UI components in your application. These are part of the view. Also any events on UI components are part of view module.
3. Presenter: This is the bridge between Model and View to control the communication. Presenter can query the model and return data to view to update it.
4. E.g. If we have a Model with large news article data, and view needs only headline, then presenter can query the data from model and only give headline to view. In this way view remains very light in this design pattern.

1. **How can we prevent users from creating more than one instance of singleton object by using clone() method?**

First we should not implement the Cloneable interface by the object that is a Singleton.

Second, if we have to implement Cloneable interface then we can throw exception in clone() method.

This will ensure that no one can use clone() method or Cloneable interface to create more than one instance of Singleton object.

1. **What is the use of Interceptor design pattern?**

Interceptor design pattern is used for intercepting a request. Primary use of this pattern is in Security policy implementation.

We can use this pattern to intercept the requests by a client to a resource. At the interception we can check for authentication and authorization of client for the resource being accessed.

In Java it is used in javax.servlet.Filter interface.

This pattern is also used in Spring framework in HandlerInterceptor and MVC interceptor.

1. **What are the Architectural patterns that you have used?**

Architecturalpatterns are used to define the architecture of aSoftware system. Some of the patterns are as follows:

1. MVC: Model View Controller. This pattern is extensively used in the architecture of Spring framework.
2. Publish-subscribe: This pattern is the basis of messaging architecture. In this case messages are published to a Topic. And subscribers subscribe to the topic of their interests. Once the message is published to a topic in which a Subscriber has an interest, the message is consumed by the relevant subscriber.
3. Service Locator: This design pattern is used in a service like JNDI to locate the available services. It uses as central registry to maintain the list of services.
4. n-Tier: This is a generic design pattern to divide the architecture in multiple tiers. E.g. there is 3-tier architecture with Presentation layer, Application layer and Data access layer. It is also called multi-layer design pattern.
5. Data Access Object (DAO): This pattern is used in providing access to database objects. The underlying principle is that we can change the underlying database system, without changing the business logic. Since business logic talks to DAO object, there is no impact of changing Database system on business logic.
6. Inversion of Control (IoC): This is the core of Dependency Injection in Spring framework. We use this design pattern

to increase the modularity of an application. We keep the objects loosely coupled with Dependency Injection.

1. **What are the popular uses of Façade design pattern?**

Some of the popular uses of Façade design pattern are as follows:

1. A Façade provides convenient methods for common tasks that are used more often.
2. A Façade can make the software library more readable.
3. A Façade can reduce the external dependencies on the working of inner code.
4. A Façade can act as a single well-designed API by wrapping a collection of poorly designed APIs.
5. A Façade pattern can be used when a System is very complex and difficult to use. It can simplify the usage of complex system.
6. **What is the difference between Builder design pattern and Factory design pattern?**

Both Factory and Builder patterns are creational design patterns. They are similar in nature but Factory pattern is a simplified generic version of Builder pattern.

We use Factory pattern to create different concrete subtypes of an Object. The client of a Factory may not know the exact subtype. E.g. If we call createDrink() of a Factory, we may get Tea or Coffee drinks.

We can also use Builder pattern to create different concrete subtypes of an object. But in the Builder pattern the composition of the object can be more complex. E.g. If we call createDrink() for Builder, we can getCappuccino Coffee with Vanilla Cream and Sugar, or we can get Latte Coffee with Splenda and milk cream.

So a Builder can support creation of a large number of variants of an object. But a Factory can create a broader range of known subtypes of an object.

1. **What is Memento design pattern?**

Memento design pattern is used to implement rollback feature in an object. In a Memento pattern there are three objects:



* Originator: This is the object that has an internal state.



* Caretaker: This is the object that can change the state of Originator. But it wants to have control over rolling back the change.



Memento: This is the object that Caretaker gets from Originator, before making and change. If Caretaker wants to Rollback the change it gives Memento back to Originator. Originator can use Memento to restore its own state to the original state.



E.g. One good use of memento is in online Forms. If we want to show to user a form pre-populated with some data, we keep this copy in memento. Now user can update the form. But at any time when user wants to reset the form, we use memento to make the form in its original pre-populated state. If user wants to just save the form we save the form and update the memento. Now onwards any new changes to the form can be rolled back to the last saved Memento object.

1. **What is an AntiPattern?**

An AntiPattern is opposite of a Design Pattern. It is a common practice in an organization that is used to deal with a recurring problem but it has more bad consequences than good ones.

AntiPattern can be found in an Organization, Architecture or Software Engineering.

Some of the AntiPatterns in Software Engineering are:

1. Gold Plating: Keep on adding extra things on a working solution even though these extra things do not add any additional value.
2. Spaghetti Code: Program that are written in a very complex way and are hard to understand due to misuse of data structures.
3. Coding By Exception: Adding new code just to handle exception cases and corner case scenarios.
4. Copy Paste Programming: Just copying the same code multiple times rather than writing generic code that can be parameterized.
5. **What is a Data Access Object (DAO) design pattern?**

DAO design pattern is used in the data persistent layer of a Java application. It mainly uses OOPS principle of Encapsulation.

By using DAO pattern it makes the application loosely coupled and less dependent on actual database.

We can even implement some in-memory database like H2 with DAO to handle the unit-testing.

In short, DAO hides the underlying database implementation from the class that accesses the data via DAO object.

Recently we can combine DAO with Spring framework to inject any DB implementation.