**Creational Design Patterns:**

**Singleton:**

* It belongs to creational Design Pattern among Factory, Abstract Factory, Builder an Prototype Patterns.
* It is used when we need to ensure that only one object of a particular class instantiated. That singleton instance created is responsible to coordinate actions across the application.
* This single instance of the object is responsible to invoke underneath methods or events.
* Concurrent access to the resource is well managed by Singleton Design pattern.
* It ensures that there is only one instance of the class exists by declaring all constructors as private.
* To control the singleton access we need to provide static property that returns the singleton instance of the object.

// final restricts inheritance  
public final class Singleton {  
  
 // private constructor ensures that object is not instantiated other than with in the class itself  
 private Singleton() { }  
   
 private static Singleton *instance*;  
  
 // public method ensures to return only one instance of the class  
 public static Singleton getInstance() {  
 if(*instance* == null)  
 *instance* = new Singleton();  
 return *instance*;  
 }  
}

* To avoid the Singleton class to be inherited, we use **final**. The above code works well under single Threaded application.
* We are delaying the instance creation till the **getInstance()** is invoked, so this delayed instance creation called as lazy initialization. It works fine in single Threaded environment.
* If multiple threads are invoking **getInstance()** at the same point of time. At this point there is a chance that multiple instances can be created of this Singleton Object.
* To avoid this, the best way is to use Locks to overcome Thread race condition.

// final restricts inheritance  
public final class Singleton {  
 private static Object *lock* = new Object();  
  
 // private constructor ensures that object is not instantiated other than with in the class itself  
 private Singleton() { }  
  
 private static Singleton *instance*;  
  
 // public method ensures to return only one instance of the class  
 public static Singleton getInstance() {  
 synchronized ( *lock*) {  
 if (*instance* == null) {  
 *instance* = new Singleton();  
 }  
 }  
 return *instance*;  
 }  
}

* In Multi-threaded environment, synchronized ensures that only one Thread which arrives and succeeds to enter the code and gets the instance, the other will wait till the first one is completed.
* Locks are very expensive to use and there is no use the locks every time when we invoke the **getInstance()**. We make use Runtime to enter into to the lock scope only when the instance is null to avoid un-necessary lock checking. This double verification of null instance checking is called **Double- check-locking**.

public static Singleton getInstance() {  
  
 if (*instance* == null) {  
 synchronized (*lock*) {  
 if (*instance* == null) {  
 *instance* = new Singleton();  
 }  
 }  
 }  
 return *instance*;  
}

* [**https://en.wikipedia.org/wiki/Double-checked\_locking**](https://en.wikipedia.org/wiki/Double-checked_locking)
* **Lazy initialization** of the object improves the performance and avoids unnecessary computations thill the object is accessed. Further it also reduces the memory foot print during the startup of the program. Reducing the memory foot print will load the application faster. Example for lazy initialization is to assign some static properties in the Singleton class and those values need to be retrieved from database. These values do not need to retrieve these values till the singleton object is accessed.
* **Eager Loading**: To initialize the required object before its being accessed which means we instantiate the object and keep it ready and use it when we need it.

public final class Singleton {  
  
 // private constructor ensures that object is not instantiated other than with in the class itself  
 private Singleton() {  
   
 }  
  
 private static final Singleton *instance* = new Singleton();  
  
 // public method ensures to return only one instance of the class  
 public static Singleton getInstance() {  
 return *instance*;  
 }  
}

* Eager loading has created a Singleton instance. It is Thread safe. The common language runtime takes of the variable initialization and its thread safety.

**Singleton and Static Class:**

* Static is a keyword and Singleton is a Design Pattern.
* Singleton preserves conventional class approach and it gives us an object where as static class provides static methods.
* Singleton is an Object creational pattern with one instance of the class.
* Singleton can implement interfaces, inherit from other classes and it aligns with the OOPS concepts.
* Singleton object can be passed as a reference to other methods or objects.
* Singleton object is stored on heap and these can be cloned as well.
* We can have static inner class in java but not the static outer class.
* Singleton can be used in Logging, Connection pool Management, Printer spooling, File Management, Device Management, Application Configuration Management, Cache Management and session based shopping card.

**Factory Design Pattern:**

* It belongs to creational type patterns. It is one of the most used design patterns in real world applications.
* ***Define an interface for creating an Object, but let subclasses decide which class to instantiate them. The factory method lets a class defer instantiation it uses to subclasses.***
* Factory pattern creates object without exposing the creation login to the client and refer to newly created object using a common interface.
* If we have a super class and n-subclasses and based on data provided, we have to return the object of one of the sub-classes, we use a factory pattern.
* The basic principle behind this pattern is that, at runtime, we get an object of similar type based on the parameter we pass.
* If object creation code is spread in whole application, and if you need to change the process of creation then you need to go in each and every place to make necessary changes.
* Provide an interface for creating families of related or dependednt objects without specifying their concrete classes.

Product

Factory

Client

* *The* ***client*** *is an object that requires an instance of the* ***Product*** *object. Rather than creating the* ***Product*** *instance directly,* ***client*** *delegates this responsibility to the* ***Factory****. Once invoked, the* ***Factory*** *creates the new instance of the* ***Product*** *passing it back to the* ***client****.*
* *The* ***Client*** *uses the* ***Factory*** *to create an instance of the* ***Product****.*

***Choose Simple Factory Pattern when***

* The Object needs to be extended to sub-classes.
* The classes doesn’t know what exactly sub-classes it has to create.
* The Product implementation tend to change over time and the Client remains unchanged.

public class EmployeeManagerFactory {  
 public EmployeeManager getEmployeeManager(long employeeTypeId) {  
 EmployeeManager empManager = null;  
 if(employeeTypeId == 1) {  
 empManager = new PermanentEmpManager();  
 } else if(employeeTypeId == 2) {  
 empManager = new ContractEmpManager();  
 }  
 return empManager;  
 }  
}

**Abstract Factory Design Pattern:**

* Abstract Factory pattern is a super-factory which creates other factories. This factory is also called as Factory of Factories. It provides an interface for creating families of related or dependent objects without specifying their concrete classes. It provides one level of interface than the factory pattern. It is used to return one of several factories.
* An Abstract factory is one level of abstraction higher than a factory method pattern, which means it returns the factory classes. The name “Abstract” suggests that it provides some type of abstraction over the creation of objects.
* Abstract factory pattern returns factory class not the objects and that is the difference between factory and abstract factory pattern. So, when we have lot of application modules and need to use factory pattern, we use abstract factory. It returns factory classes and from the factory class we get the required object.

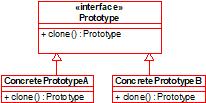
public abstract class AnimalFactory {  
 public abstract Animal getAnimal(String animalType);  
 public static AnimalFactory createAnimalFactory(String factoryType) {  
 if("sea".equalsIgnoreCase(factoryType)) {  
 return new SeaAnimalFactory();  
 } else if("land".equalsIgnoreCase(factoryType)) {  
 return new LandAnimalFactory();  
 } else if("air".equalsIgnoreCase(factoryType)) {  
 return new AirAnimalFactory();  
 } else {  
 return null;  
 }  
 }  
}

public class LandAnimalFactory extends AnimalFactory {  
 public Animal getAnimal(String type) {  
 Animal animal;  
 switch(type) {  
 case "dog" :  
 animal = new Dog();  
 break;  
 case "cat" :  
 animal = new Cat();  
 break;  
 default:  
 animal = null;  
 }  
 return animal;  
 }  
}

public class Demo {  
 public static void main(String[] args) {  
 String factoryType = "air";  
 String animalType = "sparrow";  
 Animal animal = null;  
 AnimalFactory animalFactory = null;  
 animalFactory = AnimalFactory.*createAnimalFactory*(factoryType);  
 System.*out*.println(animalFactory.getClass().getName());  
 animal = animalFactory.getAnimal(animalType);  
 System.*out*.println(animal.getClass().getName());  
 animal.speak();  
 }  
}

**Prototype Design Pattern:**

* The Prototype design pattern is used for creating new Objects by cloning other objects and this way we can improve the performance. This pattern uses java cloning to copy the object.
* This pattern is used when creation of object is costly or complex. For example, an object is to be created after a costly database operation. We can cache the object, return its clone on next request. Once we get the cloned object we can modify according to our needs.



* Only the first time Concrete Prototype Object will be created after that whenever client requests concrete prototype object, a cloned concrete prototype object will be returned to the client.

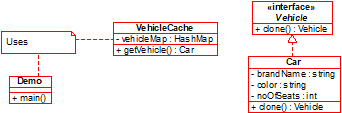
public interface Vehicle extends Cloneable{  
 public Vehicle clone();  
}

public class Car implements Vehicle {  
 private String brandName;  
 private String color;  
 private int noOfSeats;  
  
 Car(final String brandName, final String color, final int noOfSeats) {  
 super();  
 this.brandName = brandName;  
 this.color = color;  
 this.noOfSeats = noOfSeats;  
 }  
  
 @Override  
 public Vehicle clone() {  
 Car carObj = null;  
 try {  
 carObj = (Car)super.clone();  
 } catch (CloneNotSupportedException e) {  
 e.printStackTrace();  
 }  
 return carObj;  
 }

}

public class VehicleCache {  
 private static HashMap<String, Vehicle> *vehicleMap* = new HashMap<>();  
 public static Vehicle getVehicle(String vehicleType) {  
 Car car = (Car)*vehicleMap*.get(vehicleType);  
 if(car == null) {  
 car = new Car("Benz", "White", 4);  
 *vehicleMap*.put(vehicleType, car);  
 System.*out*.println("New car object is created and returned");  
 return car;  
 }  
 System.*out*.println("Cloned car object is returned");  
 return car.clone();  
 }  
}

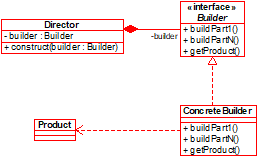
public class PrototypeDemo {  
 public static void main(String[] args) {  
 Car car = (Car) VehicleCache.*getVehicle*("car");  
 System.*out*.println(car.toString());  
 Car maruthiCar = (Car) VehicleCache.*getVehicle*("car");  
 maruthiCar.setBrandName("Maruthi Swift");  
 maruthiCar.setColor("Blue");  
 maruthiCar.setNoOfSeats(5);  
 System.*out*.println(maruthiCar.toString());  
 }  
}



* This pattern involves implementing a prototype interface which tells to create a clone of the current object. This pattern is used when creation of object directly is costly. For example, a object is to be created after a costly database operation. We can cache the object, returns its clone on next request and the cloned object client can modify as per their need.

**Builder Pattern:**

* Builder pattern builds a complex object using simple objects and using a step by step approach.
* The process of construction a complex object should be generic so that the same process can be used to create different representations of the same complex object.



* Separate the construction of a complex object from its actual representation so that the same construction process can create different representations. It is based on Directors and Builders. Any number of Builder classes can be called by a Director to produce according to the specification.
* The builder pattern is a design pattern that allows for the step-by-step creation of complex objects using the correct sequence of actions. The construction is controlled by a director object that only needs to know the type of the object it is to create.
* **Product –**The product class defines the type of the complex object that is to be generated by the builder pattern.
* **Builder –**This abstract base class defines all of the steps that must be taken in order to correctly create a product. Each step is generally abstract as the actual functionality of the builder is carried out in the concrete subclasses. The GetProduct method is used to return the final product. The builder class is often replaced with a simple interface.
* **ConcreteBuilder –**There may be any number of concrete builder classes inheriting from Builder. These classes contain the functionality to create a particular complex product.
* **Director –**The director class controls the algorithm that generates the final product object. A director object is instantiated and its Construct method is called. The method includes a parameter to capture the specific concrete builder object that is to be used to generate the product. The director then calls methods of the concrete builder in the correct order to generate the product object. On completion of the process, the GetProduct method of the builder object can be used to return the product.

// Acted as a Product  
public interface HousePlan {  
 public void setBasement(String basement);  
 public void setStructure(String structure);  
 public void setRoof(String roof);  
 public void setInterior(String interior);  
}

public class House implements HousePlan {  
 private String basement;  
 private String structure;  
 private String roof;  
 private String interior;  
  
 @Override  
 public void setBasement(String basement) { this.basement = basement; }  
  
 @Override  
 public void setStructure(String structure) { this.structure = structure; }  
  
 @Override  
 public void setRoof(String roof) { this.roof = roof; }  
  
 @Override  
 public void setInterior(String interior) {  
 this.interior = interior;  
}

// Acted as Builder  
public interface HouseBuilder {  
 public void buildBasement();  
 public void buildStructure();  
 public void buildRoof();  
 public void buildInterior();  
  
 public House getHouse();  
}

public class IglooHouseBuilder implements HouseBuilder {  
 private House house;  
  
 public IglooHouseBuilder() { this.house = new House(); }  
  
 @Override  
 public void buildBasement() { this.house.setBasement("Ice Base"); }  
  
 @Override  
 public void buildStructure() { this.house.setStructure("Ice Blocks"); }  
  
 @Override  
 public void buildRoof() { this.house.setRoof("Ice Carvings"); }  
  
 @Override  
 public void buildInterior() { this.house.setInterior("Ice Dome"); }  
  
 @Override  
 public House getHouse() { return this.house; }  
}

// Acted as Director  
public class CivilEngineer {  
  
 private HouseBuilder houseBuilder;  
  
 public CivilEngineer(HouseBuilder houseBuilder) {

this.houseBuilder = houseBuilder;

}  
  
 public House getHouse() { return this.houseBuilder.getHouse(); }  
  
 public void constructHouse() {  
 this.houseBuilder.buildBasement();  
 this.houseBuilder.buildStructure();  
 this.houseBuilder.buildRoof();  
 this.houseBuilder.buildInterior();  
 }  
}

public class BuilderDemo {  
 public static void main(String[] args) {  
 CivilEngineer engineer = new CivilEngineer(new WoodenHouseBuilder() );  
 engineer.constructHouse();  
 House house = engineer.getHouse();  
 System.*out*.println(house);  
 }  
}

**Behavioral Patterns:**

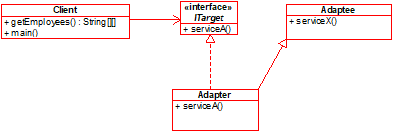
**Iterator Pattern:**

* Using Iterator Pattern we can access the elements of a collection object in sequential manner without any need to know its underlying data structure of the collection.
* Collection can be List, Set, Array.
* In OOP, the Iterator Pattern is a design pattern in which an iterator is used to traverse a container and access the container’s elements.
* The Collection interface from the java collections framework extends Iterable interface. Objects implementing the interface gets an Iterator from its only method, can be traversed using the enhanced for loop syntax.
* The idea of the iterator pattern is to take the responsibility of accessing and passing through the objects of the collection and put it in the iterator object. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated.
* It is a design pattern which is used to traverse aggregate object often called container and access container’s object without exposing its underlying representation.

**Structural Design Patterns: (7)**

Structural Design Patterns describe how (each component ) objects and classes can be combined to form flexible interconnecting modules which can work together in a larger system. It also describes how the data moves through the pattern.

**Adapter (Wrapper) Design Pattern:** (Match interfaces of different classes)



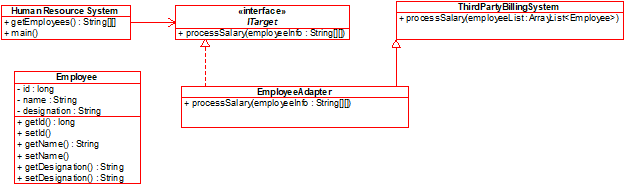
* An adapter design pattern allows two incompatible interfaces to work together. It allows incompatible classes to interact with each other by converting the interface of one class into an interface which is expected by the end clients.
* Leveraging on Adapter pattern improves reusability of older functionality.
* It works as a bridge between two incompatible interfaces. It combines the capability of two independent interfaces.
* This pattern involves a single class called adapter which is responsible for communication between two independent or incompatible interfaces.
* It allows you to make an existing class work with other existing class libraries without changing the code of the existing class.

public interface ITarget {  
 public void processSalary(String[][] employees);  
}

//Acts as Adapter  
public class EmployeeAdapter extends ThirdPartyBillingSystem implements ITarget {  
 @Override  
 public void processSalary(String[][] employeesInfo) {  
 String empId = null;  
 String name = null;  
 String designation = null;  
 ArrayList<Employee> employeeList = new ArrayList<>();  
// Conversion from Array to ArrayList  
 for(int record=0; record< employeesInfo.length; record++) {  
 for(int property =0 ; property < employeesInfo[record].length; property++) {  
 if(property == 0){  
 empId = employeesInfo[record][property];  
 } else if(property == 1){  
 name = employeesInfo[record][property];  
 } else if(property == 2){  
 designation = employeesInfo[record][property];  
 }  
 }  
 employeeList.add(new Employee(Long.*parseLong*(empId), name, designation));  
 }  
 processSalary(employeeList);  
 }  
}

//Acts as Adaptee  
public class ThirdPartyBillingSystem {  
 public void processSalary(ArrayList<Employee> employeeList ) {  
 double salary;  
 for(Employee emp : employeeList) {  
 switch(emp.getDesignation()) {  
 case "PL" : { salary = 100000; break; }  
 case "TL" : { salary = 80000; break; }  
 case "SSD" : { salary = 60000; break; }  
 case "TSD" : { salary = 40000; break; }  
 default : { salary = 20000; break; }  
 }  
 System.*out*.println("Rs. "+ salary+" credited into "+emp.getName() + " Account");  
 }  
 }  
}

// Acts as Client  
public class HumanResourceSystem {  
 public static String[][] getEmployees() {  
 String[][] employees = new String[4][];  
 employees[0] = new String[] {"101", "Sarma", "TL"};  
 employees[1] = new String[] {"102", "Swamy", "SD"};  
 employees[2] = new String[] {"103", "Sreedhar", "SSD"};  
 employees[3] = new String[] {"104", "Manoj", "TSD"};  
 return employees;  
 }  
  
 public static void main(String[] args) {  
 ITarget target = new EmployeeAdapter();  
 target.processSalary(*getEmployees*());  
 }  
}



**Choose Adapter Design pattern when:**

* A class need to be reused which does not have an interface.
* We need to work trough a separate Adapter that adapts the interface of an existing class without changing it.
* Clients don’t know whether they work with a target class directly or through another alternative that does not have the target interface.
* Example is Language Translator

**Filter Design Pattern:**

* Using filter or Criteria design pattern we can filter set of objects, using different criteria, chaining them in a decoupled way through logical operations.
* Filter a set of objects, using different criteria, chaining them in a decoupled way. Allow criterion reutilization and inheritance through logical operations (and, or, not).

public interface Criteria {  
 public List<Person> meetCriteria(List<Person> personList);  
}

public class MaleCriteria implements Criteria {  
 @Override  
 public List<Person> meetCriteria(List<Person> personList) {  
 List<Person> resultList = new ArrayList<>();  
 for(Person person : personList) {  
 if(person.getGender().equalsIgnoreCase(Constants.*MALE*) ) {  
 resultList.add(person);  
 }  
 }  
 return resultList;  
 }  
}

public class FemaleCriteria implements Criteria {  
 @Override  
 public List<Person> meetCriteria(List<Person> personList) {  
 List<Person> resultList = new ArrayList<>();  
 for(Person person : personList) {  
 if(person.getGender().equalsIgnoreCase(Constants.*FEMALE*)) {  
 resultList.add(person);  
 }  
 }  
 return resultList;  
 }  
}

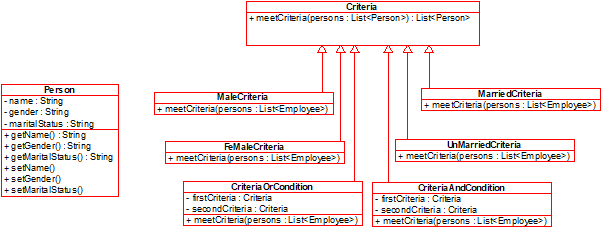
public class MarriedCriteria implements Criteria {  
 @Override  
 public List<Person> meetCriteria(List<Person> personList) {  
 List<Person> resultList = new ArrayList<>();  
 for(Person person : personList) {  
 if(person.getMaritalStatus().equalsIgnoreCase(Constants.*MARRIED*)) {  
 resultList.add(person);  
 }  
 }  
 return resultList;  
 }  
}

public class UnMarriedCriteria implements Criteria {  
 @Override  
 public List<Person> meetCriteria(List<Person> personList) {  
 List<Person> resultList = new ArrayList<>();  
 for(Person person : personList) {  
 if(person.getMaritalStatus().equalsIgnoreCase(Constants.*UNMARRIED*)) {  
 resultList.add(person);  
 }  
 }  
 return resultList;  
 }  
}

public class AnyOneCriteria implements Criteria {  
 private Criteria firstCriteria;  
 private Criteria secondCriteria;  
  
 public AnyOneCriteria(Criteria firstCriteria, Criteria secondCriteria) {  
 super();  
 this.firstCriteria = firstCriteria;  
 this.secondCriteria = secondCriteria;  
 }  
  
 @Override  
 public List<Person> meetCriteria(List<Person> personList) {  
 List<Person> firstCriteriaList = firstCriteria.meetCriteria(personList);  
 List<Person> secondCriteriaList = secondCriteria.meetCriteria(personList);  
// Check for proper condition,  
// If we loop secondlist and tries to add person to secondList we will get concurrentModificationException  
 for(Person person : secondCriteriaList) {  
 if(!firstCriteriaList.contains(person)) {  
 firstCriteriaList.add(person);  
 }  
 }  
 return firstCriteriaList;  
 }  
}

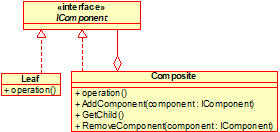
public class EveryCriteria implements Criteria {  
 private Criteria firstCriteria;  
 private Criteria secondCriteria;  
 public EveryCriteria(Criteria firstCriteria, Criteria secondCriteria) {  
 this.firstCriteria = firstCriteria;  
 this.secondCriteria = secondCriteria;  
 }  
 @Override  
 public List<Person> meetCriteria(List<Person> personList) {  
 List<Person> firstCriteriaList = firstCriteria.meetCriteria(personList);  
 return secondCriteria.meetCriteria(firstCriteriaList);  
 }  
}

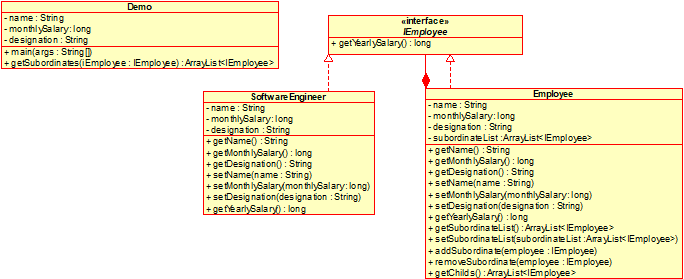
public class Person {  
 private String name;  
 private String gender;  
 private String maritalStatus;  
  
 Person(String name, String gender, String maritalStatus) {  
 this.name = name;  
 this.gender = gender;  
 this.maritalStatus = maritalStatus;  
 }  
  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public String getGender() {  
 return gender;  
 }  
 public void setGender(String gender) {  
 this.gender = gender;  
 }  
 public String getMaritalStatus() {  
 return maritalStatus;  
 }  
 public void setMaritalStatus(String maritalStatus) {  
 this.maritalStatus = maritalStatus;  
 }  
  
 @Override  
 public String toString() {  
 return "Person{" +  
 "name='" + name + '\'' +  
 ", gender='" + gender + '\'' +  
 ", maritalStatus='" + maritalStatus + '\'' +  
 '}';  
 }  
}



**Composite Design Pattern:**

* It allows you to have a tree structure and ask each node in the tree structure to perform a task.
* It builds complex objects out of simple objects and itself like a tree structure. Its goal is managing a hierarchy of objects where both leaf objects and composition of other objects conform to a common interace.
* This pattern creates a tree structure of group of objects. This class provides a way to modify its group of same objects.
* Composite pattern is used where we need to treat a group of objects in similar way as a single object. It composes objects in term of a tree structure to represent **part-whole hierarchy**.
* It treats each node in two ways : **Composite** or **Leaf**. Composite means it can have other elements below it. Leaf means it has no objects below it.



* When we want to represent **part-whole** hierarchy, use tree structure and compose objects. Compose objects into tree structures to represent part-whole hierarchies.
* Anything that can be treated as a tree structure can be considered as example of composite, eg Organization chart. 

public interface IEmployee {  
 long getYearlySalary();  
 String getName();  
 long getMonthlySalary();  
 String getDesignation();  
 IEmployee getReportingTo();  
}

// Leaf Node  
public class SoftwareEngineer implements IEmployee {  
 private String name;  
 private long monthlySalary;  
 private String designation;  
 private IEmployee reportingTo;  
  
 public SoftwareEngineer(String name, long monthlySalary, String designation, IEmployee reportingTo) {  
 this.name = name;  
 this.monthlySalary = monthlySalary;  
 this.designation = designation;  
 this.reportingTo = reportingTo;  
 }  
  
 public String getName() {  
 return name;  
 }  
 public long getMonthlySalary() {  
 return monthlySalary;  
 }  
 public String getDesignation() {  
 return designation;  
 }  
 public IEmployee getReportingTo() { return reportingTo; }  
  
 @Override  
 public long getYearlySalary() {  
 return monthlySalary \* 12;  
 }  
}

// Composite Node: Can have other nodes under it.  
public class Employee implements IEmployee {  
 private String name;  
 private long monthlySalary;  
 private String designation;  
 private ArrayList<IEmployee> subordinateList = new ArrayList<>();  
 private IEmployee reportingTo;  
  
 public Employee(String name, long monthlySalary, String designation) {  
 this.name = name;  
 this.monthlySalary = monthlySalary;  
 this.designation = designation;  
 }  
  
 public Employee(String name, long monthlySalary, String designation, IEmployee reportingTo) {  
 this.name = name;  
 this.monthlySalary = monthlySalary;  
 this.designation = designation;  
 this.reportingTo = reportingTo;  
 }  
  
 @Override  
 public long getYearlySalary() { return monthlySalary \* 12; }  
 public void addSubordinate(final IEmployee employee) { subordinateList.add(employee); }  
 public void removeSubordinate(final IEmployee employee) { subordinateList.remove(employee); }  
 public ArrayList<IEmployee> getChilds() { return getSubordinateList(); }  
 public ArrayList<IEmployee> getSubordinateList() { return subordinateList; }  
 public String getName() { return name; }  
 public long getMonthlySalary() { return monthlySalary; }  
 public String getDesignation() { return designation; }  
 public IEmployee getReportingTo() { return reportingTo; }  
}

public class Util {  
 public static void getSubordinate(IEmployee iemployee) {  
 if(iemployee instanceof Employee) {  
 Employee emp = (Employee) iemployee;  
 System.*out*.println("--------------------");  
 System.*out*.println("Subordinate List of "+emp.getName()+":");  
 ArrayList<IEmployee> empList = emp.getChilds();  
 for(IEmployee employee : empList) {  
 if(employee instanceof Employee) {  
 Employee e = (Employee) employee;  
 System.*out*.println(e.getName() + " -- "+ e.getDesignation() + " -- " + e.getMonthlySalary() + " " +  
 "-- " + e.getYearlySalary()+ ", Subordinates present" );  
 }  
 else {  
 *printLeafs*(employee);  
 }  
 }  
 } else {  
 *printLeafs*(iemployee);  
 }  
 }  
  
 public static void printLeafs(IEmployee employee) {  
 SoftwareEngineer emp = (SoftwareEngineer) employee;  
 System.*out*.println(emp.getName() + " -- "+ emp.getDesignation() + " -- " + emp.getMonthlySalary() + " " +  
 "-- " + emp.getYearlySalary()+ ", No Subordinates" );  
 }  
  
 public static void printChilds(IEmployee employee) {  
 Employee emp = (Employee) employee;  
 ArrayList<IEmployee> children = ((Employee) employee).getChilds();  
 System.*out*.println("Childs for "+ emp.getName());  
 String childsPresent;  
  
 for(IEmployee child : children) {  
 childsPresent = child instanceof Employee ? ", Subordinates present" : ", No Subordinates";  
 System.*out*.println(child.getName() + " -- "+ emp.getDesignation() + " -- " + emp.getMonthlySalary() + " " +  
 "-- " + emp.getYearlySalary() + childsPresent );  
 }  
 }  
 public static void getAllChilds(IEmployee iemployee) {  
 if( iemployee instanceof Employee) {  
 Employee emp = (Employee) iemployee;  
 System.*out*.println("--------------------");  
 *printChilds*(emp);  
 ArrayList<IEmployee> empList = emp.getChilds();  
 for(IEmployee employee : empList) {  
 if(employee instanceof Employee) {  
 *getAllChilds*(employee);  
 }  
 }  
 }  
 }  
 public static void getAllParents(IEmployee iemployee) {  
 System.*out*.println("--------------------");  
 Employee emp = (Employee) iemployee.getReportingTo();  
 if(emp != null) {  
 System.*out*.println(emp.getName() + " -- "+ emp.getDesignation() + " -- " + emp.getMonthlySalary() +  
 "-- " + emp.getYearlySalary() );  
 }  
 if(emp.getReportingTo() !=null) {  
 *getAllParents*(emp);  
 }  
 }  
}

public class CompositeDemo {  
 public static void main(String[] args) {  
 Employee bob = new Employee("BOB", 700000, "General Manager");  
 Employee kiran = new Employee("Kiran Nair", 500000, "Manager", bob);  
 Employee prasad = new Employee("Prasad", 200000, "Deputy Manager",kiran);  
 Employee aviral = new Employee("Aviral", 150000, "Assistant Manager",prasad);  
 Employee mouli = new Employee("Mouli", 50000, "Analyst", aviral);  
 SoftwareEngineer jasmine = new SoftwareEngineer("Jasmine", 50000, "Analyst",prasad);  
 SoftwareEngineer thomas = new SoftwareEngineer("Thomas", 20000, "Analyst",mouli);  
 Employee jos = new Employee("Jos Angel George", 150000, "PL",prasad);  
 SoftwareEngineer dp = new SoftwareEngineer("Durga Prasad", 70000, "SSD", jos);  
 SoftwareEngineer manoj = new SoftwareEngineer("Manoj Goud", 60000, "SD", jos);  
 SoftwareEngineer swamy = new SoftwareEngineer("Swamy", 50000, "SD", jos);  
 SoftwareEngineer sreedhar = new SoftwareEngineer("Sreedhar", 50000, "SD", jos);  
 SoftwareEngineer sarma = new SoftwareEngineer("Sarma", 80000, "TL", jos);  
  
 bob.addSubordinate(kiran);  
 kiran.addSubordinate(prasad);  
 kiran.addSubordinate(aviral);  
 prasad.addSubordinate(jos);  
 prasad.addSubordinate(jasmine);  
 aviral.addSubordinate(mouli);  
 mouli.addSubordinate(thomas);  
 jos.addSubordinate(dp);  
 jos.addSubordinate(manoj);  
 jos.addSubordinate(swamy);  
 jos.addSubordinate(sreedhar);  
 jos.addSubordinate(sarma);  
  
 Util.*getSubordinate*(bob);  
 Util.*getSubordinate*(jos);  
 Util.*getAllChilds*(prasad);  
 Util.*getAllParents*(sarma);  
 }  
}

**Bridge Pattern:**

* In Bridge pattern, there are two parts: Abstraction, Implementation. The bridge pattern allows the Abstraction and the Implementation to be developed independently, and the client code can access only the Abstraction part without being concerned about the implementation part.
* Bridge pattern separates the abstraction hierarchy and the implementation hierarchy in two different layers so that change in one hierarchy will not affect the development or functionality of the other hierarchy.

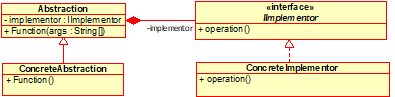
Persistence

Persistence Implementation

1. Save an Object
2. Delete an Object
3. FileSystemPersistenceImplementor
4. DatabasePersistenceImplementor

**Persistence will use of the implementor to save/delete the object**

* Client will use Abstraction only to use operations.

****