**4 Reasons to Use Composition over Inheritance in Java and OOP  
  
Favour composition over inheritance is a one of the popular object oriented design principle, which helps to create flexible and maintainable code in Java and other object oriented languages.   
  
4 Reasons to Prefer Composition over Inheritance in Java**

Just to revise, composition and Inheritance are ways to reuse code to get additional functionality. In Inheritance, a new class, which wants to reuse code, inherit an existing class, known as super class. This new class is then known as sub class. On composition, a class, which desire to use functionality of an existing class, doesn't inherit, instead it holds a reference of that class in a member variable, that’s why the name composition. Inheritance and composition relationships are also referred as IS-A and HAS-A relationships. Because of IS-A relationship, an instance of sub class can be passed to a method, which accepts reference of super class. This is a kind of Polymorphism, which is achieved using Inheritance. A super class reference variable can refer to an instance of sub class. By using composition, you don’t get this behaviour, but still it offers a lot more to tilde the balance in its side.  
  
  
1) One reason of favouring Composition over Inheritance in Java is fact that Java doesn't support multiple inheritance. Since you can only extend one class in Java, but if you need multiple functionality like e.g. for reading and writing character data into file, you need Reader and Writer functionality and having them as private members makes your job easy. That’s called composition. If you are following program to interface than implementation principle, and using type of base class as member variable, you can use different Reader and Writer implementation at different situation. You won’t get this flexibility by using Inheritance, in case of extending a class, you only get facilities which are available at compile time.

Inheritance example:

Class FileReader

{

Read()

{

// code to read from file

}

}

Class FileWriter

{

Write()

{

// code to write to file

}

}

Class MyApplication:FileWriter, FileReader

{

Perform()

{

Read() // of FileReader // tight coupling

Write() // of FileWriter // tight coupling

}

}

drawbacks:

* Tight coupling- if base class ( FileReader or FileWriter) is changed, sub class (MyApplication) will break.
* inheritance breaks encapsulation. **white-box reuse**. That is ,with inheritance, the parent class implementation is often visible to the subclasses.  
    
  2) Composition offers better test-ability of a class than Inheritance. If one class is composed of another class, you can easily create Mock Object representing composed class for sake of testing. Inheritance doesn't provide this luxury. In order to test derived class, you must need its super class. Since unit testing is one of the most important thing to consider during software development, especially in test driven development, composition wins over inheritance.

e.g.

interface Reader

{

Void read();

}

Interface Writer

{

Void write();

}

Class FileReader implements Reader

{

Void read()

{

Code to read from file

}

}

Class FileWriter implements Writer

{

Void write()

{

Code to write to file

}

}

Class MyApplication

{

// program to interface, enables loose coupling

Reader ref1;

Writer ref2;

Public MyApplication(Reader ref1,Writer ref2)

{

This.ref1=ref1;

This.ref2=ref2;

}

Void perform()

{

Ref1.read(); // late binding

Ref2.write(); // late binding

}

}

In the above example “MyApplication” is composed of “Reader” and “Writer”.

Advantages:

* **black-box reuse** as it does not break encapsulation. MyApplication knows only selected functionalities from “Reader” and “Writer”.
* Loose coupling, program to interface. During runtime any implementations (such as “FileReader” or “SocketReader” and “FileWriter” or “SocketWriter”) can be passed to “Reader” or “Writer” respectively and “read()” method can be invoked on it polymorphically.  
    
  3) Many object oriented design patterns mentioned by Gang of Four in there timeless classic Design Patterns: Elements of Reusable Object-Oriented Software, favours Composition over Inheritance. Classical examples of this is Strategy design pattern, where composition and delegation is used to change Context’s behaviour, without touching context code. Since Context uses composition to hold strategy, instead of getting it via inheritance, it’s easy to provide a new Strategy implementation at run-time.

e.g.

import java.util.\*;

class Employee

{

private String name;

private int age;

public Employee(String name, int age)

{

this.name = name;

this.age = age;

}

public String toString()

{

return "["+name+"\t"+age+"]";

}

public String getName()

{

return name;

}

public int getAge()

{

return age;

}

}

class SortByName implements Comparator<Employee>

{

public int compare(Employee e1, Employee e2)

{

return e1.getName().compareTo(e2.getName());

}

}

class SortByAge implements Comparator<Employee>

{

public int compare(Employee e1, Employee e2)

{

if(e1.getAge()>e2.getAge())

{

return 1;

}

else if(e1.getAge()<e2.getAge())

{

return -1;

}

else

{

return 0;

}

}

}

public class ComparatorDemo1

{

public static void main(String[] args)

{

List<Employee> employeeList = new ArrayList<Employee>();

employeeList.add( new Employee("Tim", 23) );

employeeList.add( new Employee("Rolvin", 11) );

employeeList.add( new Employee("Gerald", 32) );

System.out.println("Sort by name");

Collections.sort(employeeList, new SortByName() );//sort by name;

System.out.println(employeeList);

System.out.println("sort by age");

Collections.sort(employeeList, new SortByAge() );//sort by age;

System.out.println(employeeList);

}

}

Here “Collections.sort()” method takes two arguments i.e. collection implementation and “Comparator” interface. i.e. once again “Program to interface”. And hence we can pass any implementation of “Comparator” i.e. “SortByName” or “SortByAge” in the above example.

Another good example of using composition over inheritance is Decorator design pattern. In Decorator pattern, we don't extend any class to add additional functionality, instead we keep an instance of the class we are decorating and delegates original task to that class after doing decoration. This is one of the biggest proof of choosing composition over inheritance, since these design patterns are well tried and tested in different scenarios and withstand test of time, keeping their head high.

e.g.

public interface Room

{

public String showRoom();

}

public class LivingRoom implements Room

{

@Override

public String showRoom() {

return "Living Room";

}

public void applyColors()

{

// code to applycolors

}

public void applyCurtains()

{

// code to applycurtains

}

}

public class BedRoom implements Room

{

@Override

public String showRoom()

{

return "Bed Room";

}

public void applyColors(Color c)

{

// code to applycolors

}

public void applyCurtains(Color c)

{

// code to applycurtains

}

}

The following class is the decorator class. It is the core of the decorator design pattern. It contains an attribute for the type of Room interface. Instance is assigned dynamically at the creation of decorator using its constructor. Once assigned, that instance method will be invoked.

RoomDecorator.java(Decorator):

abstract public class RoomDecorator

{

protected Room ref;

public RoomDecorator (Room ref)

{

this.ref=ref;

}

public void display()

{

ref.showRoom();

}

public abstract void decorate();

}

ColorDecorator.java(ConcreteDecorator):

public class ColorDecorator extends RoomDecorator

{

Color c;

public ColorDecorator (Room ref,Color c) {

super(ref);

this.c=c;

}

public void decorate()

{

addColors();

}

private void addColors() {

ref.applyColors(c);

}

}

CurtainDecorator.java(ConcreteDecorator):

public class CurtainDecorator extends RoomDecorator

{

Color c;

public CurtainDecorator (Room ref,Color c) {

super(ref);

this.c=c;

}

public void decorate()

{

addCurtains();

}

private void addCurtains() {

ref.applyCurtains(c);

}

}

DecoratorDesignPatternMain.java:

public class DecoratorDesignPatternMain

{

public static void main(String args[])

{

Room room1 = new LivingRoom();

ColorDecorator cd=new ColorDecorator(room1,pink);

cd.decorate();

cd.display();

CurtainDecorator cd1=new CurtainDecorator(room1,red);

cd1.decorate();

cd1.display();

Room room2 = new BedRoom();

ColorDecorator cd2=new ColorDecorator(room2,blue);

cd2.decorate();

cd2.display();

}

}

In short, don't use Inheritance just for the sake of code reuse, Composition allows more flexible and extensible mechanism to reuse code.

5. Another reason of favouring Composition over inheritance is flexibility. If you use Composition you are flexible enough to replace implementation of composed class with better and improved version. One example is using Comparator class which provides compare functionality. If your Container object contains a Comparator instead of extending a particular Comparator for comparing, it’s easier to change the way comparison is performed by setting different type of Comparator to composed instance, while in case of inheritance you can only have one comparison behaviour on runtime, you cannot change it runtime.  
  
  
In nutshell favouring Composition results in more flexible and robust code than using Inheritance. Though there are certainly some cases where using Inheritance makes much sense like when a genuine parent child relation exists, but most of time it makes sense to favour composition over inheritance for code reuse.