Haaris Infotech
Driven by Technology

Assignment – Pen and Paper Codathon

Re create the entire material using pen and paper and send the picture of the same to whatsapp – 9840135749 for evaluation. Early finishers will be rewarded with fantastic prizes.

Duration – 18 hrs

Apart from code, you also have descriptive questions, answer them by referring internet.

Part 1
Basic programming exercises to understand the concept.
Part 2 – Design Pattern
Part 3 – Q & A.

Part 2
Creational Patterns

1. Singleton
2. Abstract Factory
3. Builder
4. Prototype

Behavioral Patterns
1. Chain of Responsibility
2. Command
3. Interpreter
4. Mediator
5. Strategy
6. Template Method
7. Visitor

Structural Patterns
1. Adapter
2. Decorator
3. Bridge
4. Proxy

Part 3

4 Marks Question – Each Question carry 4 marks — Total Marks 5 *4 = 20

1. What is Open Close principle, demonstrate the same with an example.
2. What is Dependency Inversion Principle? demonstrate the same with an example.
3. What is Interface Segregation Principle? demonstrate the same with an example.
4. What is Inheritance, and what are the advantages of inheritance ? Also specify why the

super class has to be abstract ?

5. What is static key word, explain and demonstrate static variable, static method, static class, and static block.

3 Mark Questions – Each question carry 3 marks-– Total Marks – 3 *10 = 30

1. What is a constructor, overloaded constructor, default constructor, demonstrate the same with appropriate examples.
2. What is Method overloading, and constructor overloading , demonstrate with an example.
3. What is Method overriding and demonstrate the same with an example
4. What is super, final and this keyword, demonstrate with an example.
5. What are Access specifiers, demonstrate the same with sample code.
6. What is inner class and static inner class, demonstrate the same with example.
7. What is pass by value and what is pass by reference, demonstrate with an example
8. What is an interface and what is its significance. Demonstrate with an example.
9. What is the difference between abstract class and interface in java.
10. What is multiple inheritance? Is it supported by Java?

2 Mark Questions – Each question carry 2 marks – Total Marks 2 *10=20

1. What is the difference between equals() and == in Java?
2. When can you use the super keyword?
3. Can you call a constructor of a class inside another constructor?
4. What are the main concepts of OOPs in Java?
5. Differentiate between the constructors and methods in Java?
6. What is difference between instance variable and class variable
7. What is final keyword in java
8. What is the difference between continue and break statements.
9. What is polymorphism.
10. Can you override a private or static method in Java?

1 Mark questions – Answer all 10 – Total Marks - 5

1. What is an association?
2. What is aggregation ?
3. What is composition ?
4. What is generalization ?
5. What is realization ?
6. What gives Java its 'write once and run anywhere' nature?
7. What is the default value of the local variables?
8. What is the default value of local array variable ?
9. What is a class ?
10. What is a object ?

```
package ch6to10;

public class ClassDemo1 {

}

class RedPaint{//Classification class - Strictly should be
either individual or under a category

}

abstract class Paint{//Classifier Class - Strictly common
noun

}

interface Painter{//Action Class - Strictly Role - Classfier
Class

}

package ch6to10;

public class ClassDemo2 {
	public static void main(String[] args) {
		Test obj=new Test();//this is how we create a objet
of a class
		obj.met();//fire and forget
		obj.met2();
		int result=obj.met3();//fire and expect
		System.out.println(result);
	}
```

```
}

class Test{
     //Primitive types or simple types

     int i;//variables declared outside the methods are
called INSTANCE VARIABLES - denotes state

     void met() { //- denotes behavior
          //INSTANCE VARIABLES are initialized by default but
local variables are
          //not initialized by default
          int meti=10;//local variable
          System.out.println(i);
          System.out.println(meti);
     }
     void met2() {
          System.out.println(i);
     }

     int met3() {
          return 10;
     }
}

……

package ch6to10;

public class ClassDemo3 {
     public static void main(String[] args) {
          Bank bank=new Bank();
          System.out.println(bank.depositCash(900));
     }
}

class Bank{
	boolean depositCash(int amt) {
		if(amt >1000) {
			return true;
		}
		else {
			return false;
		}
	}

	long getMoney() {
		return 10L;
	}

	String getName() {
		return "ram";
	}
}
```

…….

```
package ch6to10;

public class ClassDemo4 {
	public static void main(String[] args) {
		Demo obj=new Demo();
		obj.met(20,30,"hello");

		int[] a=new int[] {10,20,30,40};
		obj.met2(a);//indirect

		obj.met2(new int[] {10,30,40,50});//direct


		obj.met3(120,34,56,67,78);//we are passing
arguments

		int b[]=obj.met3(20,30,40,7878,9898);
```

```
        for(int i:b) {
            System.out.println("........."+i);
        }
        obj.met4("aaa",23432,"affds",new Demo());
    }
}

class Demo{
    void met(int i,int j,String s) {
        System.out.println(i+":"+j+":"+s);
    }

    void met2(int a[]) {//method is accepting parameters
        for(int i=0;i<a.length;i++) {
            System.out.println(a[i]);
        }
        for(int i:a) {
            System.out.println(i);
        }
    }
    int[]  met3(int ...abc) {//variable arguments (VARARGS)
        for(int i:abc) {
            System.out.println(i);
        }
        return abc;
    }
    void met4(Object ...o) {

    }
}
```

……

```
package ch6to10;

public class ClassDemo44 {
    public static void main(String[] args) {
          Demo2 obj=new Demo2();
          obj.met(0, 0, null, 0);

          obj.met2(23,"aaaa","fdsaa",new
String("fdafdsa"),199,234,342);

          obj.met3(new Object[] {23,"aaaa","fdsaa",new
String("fdafdsa"),199,234,342});
     }
}

class Demo2{
     void met(int i,int j,String s,int k) {

     }

     void met2(Object ...obj) {
          System.out.println(obj.length);
          System.out.println(obj[2]);
          for(int i=0;i<obj.length;i++) {
               System.out.println(obj[i]);
          }
     }

     void met3(Object obj[]) {

     }
}
```

….

Constructors

```
package ch6to10;
//https://fluvid.com/videos/detail/p35djuXqDmIaY1VMZ#.YoytmO
Y-5Qs.link
```

```
public class ConsDemo {

	public ConsDemo() {
		System.out.println("cons with default called...");
	}
	public ConsDemo(int i) {
		System.out.println("cons with i value 
called...:"+i);
	}
	public ConsDemo(Employee e) {
		System.out.println("employee cons called...."+e);
	}


	public static void main(String[] args) {
		new ConsDemo();
		new ConsDemo(20);
		new ConsDemo(new Employee());
	}
}
class Employee{}
…
```

Inner Class

```
package ch6to10;

//https://fluvid.com/videos/detail/oKxdEhYROEi-
oqVn7#.Yo4ZFP19gD8.link
public class InnerClassDemo {
		public static void main(String[] args) {
			Outer outer=new Outer();

			Outer.Inner inner=new Outer().new Inner();
			inner.innerMethod();
```

```
                Outer2.StaticInner sinner=new
Outer2.StaticInner();
                sinner.staticClassInnerMethod();
          }
}


class Outer{
     private String money="hundred rupees";
     private void outerMethod() {
          new Inner().innerMethod();
          System.out.println("outer method..."+new
Inner().innerMoney);
     }
     class Inner{
          private String innerMoney="thousand rupees......";
          public void innerMethod() {
               outerMethod();
               System.out.println("inner method
called..."+money);
          }
     }
}

class Outer2{
     private static String money2="two hundred
rupees............";
     private static void outerMethod2() {
          System.out.println("outer static method
called...");
     }
     static class StaticInner{
          public void staticClassInnerMethod() {
               outerMethod2();
               System.out.println("static class inner method
called..."+money2);
          }
     }
}
…..

package ch6to10;

public class InnerClassDemo2 {
     public static void main(String[] args) {
          Pepsi pepsiCo=new Pepsi();

          Kalimark kali=new Kalimark();

          pepsiCo.sellCola();
          kali.sellCola();

     }
}

class Pepsi{
     public void sellCola() {
          Kalimark.CampaCola cc= new Kalimark().new
CampaCola();
          cc.makeCola();
          System.out.println("pepsi fills the cola in pepsi
bottle and sells....");
     }
}

class Kalimark{
     public void sellCola() {
          CampaCola cc=new CampaCola();
          cc.makeCola();
          System.out.println("kalimark fills the cola in
bovonto bottle and sells...");
     }
     class CampaCola{
          public void makeCola() {
```

```
                        System.out.println("campa cola makes 
cola...");
                }
                public class A{}
        }
}

….

Access Specifiers

package ch6to10;

//https://fluvid.com/videos/detail/aQadEULXXOt3k1YBg#.YpSJGC
ia9rU.link
public class SameClass{
        private int pri;
        int nomod;
        protected int pro;
        public int pub;
        public void met() {
                System.out.println(pri);
                System.out.println(nomod);
                System.out.println(pro);
                System.out.println(pub);
        }
}

class SamePackNonSub{
        public void met() {
                SameClass obj=new SameClass();
                System.out.println(obj.pri);
                System.out.println(obj.nomod);
                System.out.println(obj.pro);
                System.out.println(obj.pub);
        }
}
```

```
class SamePackSub extends SameClass{
	public void met() {
		System.out.println(pri);
		System.out.println(nomod);
		System.out.println(pro);
		System.out.println(pub);
	}
}

….

package ch6to10;
//https://fluvid.com/videos/detail/G6x-
Ycgd1ZTxvy_78#.YoteqxvpXrc.link
public class PassByValueDemo {
	public static void main(String[] args) {
		PassByValueDemo obj=new PassByValueDemo();
		obj.met(20);//passing a value of 20 the met method
as argument

		sMet();
	}

	public void met(int i) {
		i=i+10;
		System.out.println(i);
		met2("hello",i);

	}

	public void met2(String s,int i) {
		System.out.println(s+":"+i);
	}
	public static void sMet() {
		//static methods cannot access non static methods
and properties
```

```
		//met();
	}
}
.....
package ch6to10;

public class PassByReferenceDemo {
	public static void main(String[] args) {
		MyBank sbi=new MyBank();

		Money myMoney=new Money();

		sbi.acceptMoney(myMoney);

		sbi.TransferMoney(10000, new Account());
	}
}

class MyBank{
	public void acceptMoney(Money m) {

	}

	public void TransferMoney(int amt, Account acct) {
		acct.credit(100);
		acct.debit(200);
	}
}

class Money{

}

class Account{
	public void debit(int drid) {
		System.out.println("debited...:"+drid);
	}
```

```
     public void credit(int crdid) {
          System.out.println("credited...:"+crdid);
     }

}
```

….

Method Overloading

```
package ch6to10;

public class MethodOverLoadingDemo1 {
     public static void main(String[] args) {
          MethodOverLoadingDemo1 obj=new
MethodOverLoadingDemo1();

          obj.met(23.3f);
     }

     public void met(int i,int j) {
          System.out.println("int param met method
called..");
     }

      int met() {
          System.out.println("no param met method
called...");
          return 1;
     }
      void met(float f) {
           System.out.println("float param method
called...");
      }
}
```

…

```
package ch6to10;

public class MethodOverLoadingDemo2 {
	public static void main(String[] args) {
		Dog tiger=new Dog();
		Stick st=new Stick();
		Biscuit biscuit=new Biscuit();
		Stone sto=new Stone();

		tiger.play(sto);
	}
}

class Dog{
	public void play(Biscuit b) {
		System.out.println("nai valattum........"+b);
		b.eat();
	}
	public void play(Stick stick) {
		System.out.println("nai odip pogum...."+stick);
		stick.aiyoAppa();
	}

	public void play(Stone mystone) {
		System.out.println("nai kadikkum....."+mystone);
		mystone.kallu();
	}
}

class Biscuit{
	public void eat() {
		System.out.println("yummy yummy..........");
	}
}
class Stick{
	public void aiyoAppa() {
```

```
          System.out.println("pinni eduthachi....");
     }
}
class Stone{
     public void kallu() {
          System.out.println("kalla kanda naya kanum.....");
     }
}
```

….
This Demo

```
package ch6to10;
//https://fluvid.com/videos/detail/5Ad-
5CB2nZtAOgmpy#.Yo4D1mWskzM.link
public class ThisDemo {
     public static void main(String[] args) {
          Fan khaitan=new Fan();
          khaitan.setup();
     }
}


class Fan{
     public Fan() {
          System.out.println("fan object created...");
     }
     public void on() {
          System.out.println("fan switched on...");
     }
     public void setup() {
          Room myRoom=new Room();
          myRoom.fix(this);//this keyword represents the
current object
          //myRoom.fix(new Fan());
     }
}
```

```
class Room{
	public void fix(Fan f) {
		f.on();
	}
}
```

…

```
package ch6to10;

public class ThisDemo2 {
	String s;
	public ThisDemo2(String s) {
		this.s=s;
	}
	public static void main(String[] args) {
		//this keyword cannot be used inside a static
method
		ThisDemo2 obj=new ThisDemo2("some value...");
		obj.printS("aaaaaaaaaa");
	}

	public void printS(String s) {
		System.out.println(this.s);
	}
}
```

Inheritance

```
package ch6to10;
//https://fluvid.com/videos/detail/-
KRkYhqQ21hB9ygO2#.YpRVFbpklsg.link
public class InheritanceDemo1 {
	public static void main(String[] args) {
		//since subclass is a kind of super class, the
subclass object can be assigned to super class reference
type
```

```
          SuperClass sc=new SubClass();
          System.out.println(sc.name);
          sc.met();
          sc.absmet();
     }
}
//Good Rules pertaining to Inheritance - They are not
compulsory but good
/*
 * 1. The super class should be either abstract or interface
 * 2. The super class can have methods or need not have
methods
 * 3. Since the super class is abstract or interface, its
object cannot be created
 * 4. If you have to create a object of the super class then
you need inherit that class with child class
 * 5. The super class can have abstract methods.
 * 6. The super class can have non abstract methods, but
they should be declared final or private
 * 7. All the public, nomodifier and protected properties of
super class will be visible in sub class
 *
 */
abstract class SuperClass{
     public SuperClass() {
          System.out.println("super class cons called...");
     }
     final String name="anound";//final variables are
constants - value cannot be changed
     final public void met() {//make sure the method is
declared final, so that the method cannot be overriden
          System.out.println("met of super class
called.....");
     }
     abstract void absmet() ;
}
```

```
class SubClass extends SuperClass{
	//you can override the non private and non final methods
of the super class
	public void absmet() {
		System.out.println("met of sub class
called............");
	}
}
```

…

```
package ch6to10;
//super key word refers - super class constructor,super
class variable
//super class method
public class SuperKeyWordDemo {
	public static void main(String[] args) {
		Medium m=new Medium();
		m.topMet();
		//m.topMetSub();
	}
}
abstract class TopTopStar{
	public void topMet() {
		System.out.println("top top met method called
....");
	}
}

abstract class TopStar extends TopTopStar{
	public void topMet() {
		super.topMet();
		System.out.println("top met method called ....");
	}
}
```

```
abstract class Top extends TopStar{
	final public void topMet() {
		super.topMet();
		System.out.println("top met sub method 
called....");
	}
}

class Medium extends Top{

}


…
package ch6to10;

public class SuperWithCons {
	public static void main(String[] args) {
		ConsBot cb=new ConsBot("aaa");

	}
}

abstract class ConsTop{
	public ConsTop() {
		System.out.println("top cons object created..");
	}
}

abstract class ConsMed extends ConsTop{
	public ConsMed(int i) {
		System.out.println("med cons object created..."+i);
	}
}

class ConsBot extends ConsMed{
	public ConsBot(String s) {
		super(100);//super call to cons should be the first
line
		System.out.println("bottom cons object 
created...");
	}
}

…..
package ch6to10;

public class StaticBlockDemo {
	static {
		System.out.println("static block called....");
	}

	public static void main(String[] args) {
		//new SBDemo();
		SBDemo.met();
		//SBDemo.met2();
	}
}


class SBDemo{
	static {
		System.out.println("SBDemo static block 
called....");
	}
	SBDemo(){//constructor
		System.out.println("constructor called...");
	}
	static void met() {
		System.out.println("static method met called...");
	}

	static void met2() {
		System.out.println("static method met2 called...");
	}
```

```
}

….

package ch6to10;

public class StaticComplexType {
	public static void main(String[] args) {
		ClassRoom newton=new ClassRoom();

		ClassRoom edison=new ClassRoom();

		System.out.println(newton.sanyo+":"+newton.sulab);

		System.out.println(edison.sanyo+":"+edison.sulab);

		ClassRoom gandhi=new ClassRoom();
		System.out.println(gandhi.sanyo+":"+gandhi.sulab);
	}
}

class ClassRoom{
	Projector sanyo=new Projector();
	static Toilet sulab=new Toilet();
}

class Projector{

}

class Toilet{

}

class Human{}

…
```

```
package ch6to10;
//https://fluvid.com/videos/detail/9wa-
BiEwv6FQV578d#.Yosxf4jtfF4.link
public class StaticDemo1 {
	public static void main(String[] args) {
		
		House.saram="this is common saram...";
		System.out.println(House.saram);
		
		House veedu1=new House();
		veedu1.saram="2 ton saram bought...";
		veedu1.toilet="western toilet...";
		
		System.out.println("Veedu1 saram...:"+House.saram);
		System.out.println("Veedu1 
toilet..:"+veedu1.toilet);
		veedu1.toilet("plastic..taps...");
		
		House veedu2=new House();
		veedu2.toilet="indian toilet...";
		System.out.println("Veedu2 saram...:"+House.saram);
		System.out.println("Veedu2 
toilet..:"+veedu2.toilet);
		veedu2.toilet("steel tap....");
		
		System.out.println("v1..:"+veedu1.toilet);
		System.out.println("v2..:"+veedu2.toilet);
		
		House.saram="3 ton saram....";
		
		System.out.println("v2...:"+House.saram);
	}
}


class House{
	static String saram;//class variable
	String toilet;//instance variable
	public void toilet(String tap) {
		String taps=tap;//local variable
		System.out.println(taps);
	}
}


…
package ch6to10;
////https://fluvid.com/videos/detail/9wa-
BiEwv6FQV578d#.Yosxf4jtfF4.link

public class StaticDemo2 {
	public static void main(String[] args) {
		//FloorMill obj=new FloorMill();
		//obj.gridingWheet();
		FloorMill.gridingWheet();
	}
}

class FloorMill{
	public static void gridingWheet() {
		System.out.println("wheet is being
grinded.......");
	}
}
…
```

Interfaces

```
package ch6to10;

public class InterDemo2 {
	public static void main(String[] args) {
		CorporationIDCard ourstudents=new CorpStudent();
		ConventIDCard students2=new ConventStudent();
```

```
        InterDemo2 camp=new InterDemo2();
        camp.luxuryRoom(students2);
        camp.ordinaryRoom(ourstudents);
        camp.mess((ConventIDCard)ourstudents);
    }

    public void luxuryRoom(ConventIDCard cid) {

    }
    public void ordinaryRoom(CorporationIDCard cic) {

    }
    public void mess(ConventIDCard cid) {

    }
}

interface CorporationIDCard{

}

interface ConventIDCard{

}

class CorpStudent implements
CorporationIDCard,ConventIDCard{

}

class ConventStudent implements ConventIDCard{

}

…
```

String

```
package ch6to10;
//https://fluvid.com/videos/detail/Y5V24TV6VAf4VV6Lv#.YodAHG
do7ws.link
//https://fluvid.com/videos/detail/G6x-
YcgppRUxvvOXn#.YodzgNJKsBs.link
public class StringDemo {
    public static void main(String[] args) {
        String str=new String("hello");  //this will create
two objects
        String str2="hello";//this will create one object

        System.out.println(str==str2);//this will compare
the objects not the value

        System.out.println(str.equals(str2));//equals
method compares the values...

        if(str.equals(str2)) {
            System.out.println("both are equal...");
        }

        String v=String.format("My name is is
..:%s","Billa");

        System.out.println(v);

        v=String.format("My name is is ..:%d...%s and my
age is..:%d",100,"Billa",70);

        System.out.println(v);

        v=String.format("My name is is ..:%s and my age
is..:%d and my weight is..:%f","Billa",70,60.5f);

        System.out.println(v);
```

```
          System.out.printf("My name is is ..:%s and my age
is..:%d and my weight is..:%f","Billa",70,60.5f);

          System.out.printf("\n %-4s %10s %20s",
"Column1","Column2","Column3");

          System.out.printf("\n %-4s %10s %20s",
"SNo","Name","Company");
          System.out.printf("\n %-4s %-20s %20s",
"100","Ramu","Anound Technologies");

          System.out.printf("\n %-20s %-20s %-20s",
"SNo","Name","Company");
          System.out.printf("\n %-20d %-20s %-20s",
100,"Ramu","Anound Technologies");
          System.out.println();
          System.out.printf("%.2f",100.279f);
          String s="hello world";
          System.out.println(s.substring(2,5));
     }
}
```

.....

```
package ch6to10;
//https://fluvid.com/videos/detail/8EL-9T-
66Ys5BBp1y#.Yod7jzWa-So.link
public class StringDemo2 {
     public static void main(String[] args) {
          String s="Hello World";

          System.out.println(s.charAt(1));//prints the
character at that position

          System.out.println(s.indexOf('e'));//prints 1
          System.out.println(s.toUpperCase());

          String str=s.toLowerCase();

          System.out.println(str);

          char c[]=s.toCharArray();

          for(char cc:c) {
               System.out.print(cc+"\t");
          }

          str=s.substring(6);
          System.out.println();
          System.out.println(str);

          str=s.substring(2,8);
          System.out.println(str);//reads from 2nd character
to 7th character

          str=str.concat("myworld");

          System.out.println(str);

          System.out.println(str.length());
     }
}
```

Design Patterns

Singleton

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

Implementation

We're going to create a *Resource* class. *Resource* class have its constructor as private and have a static instance of itself.

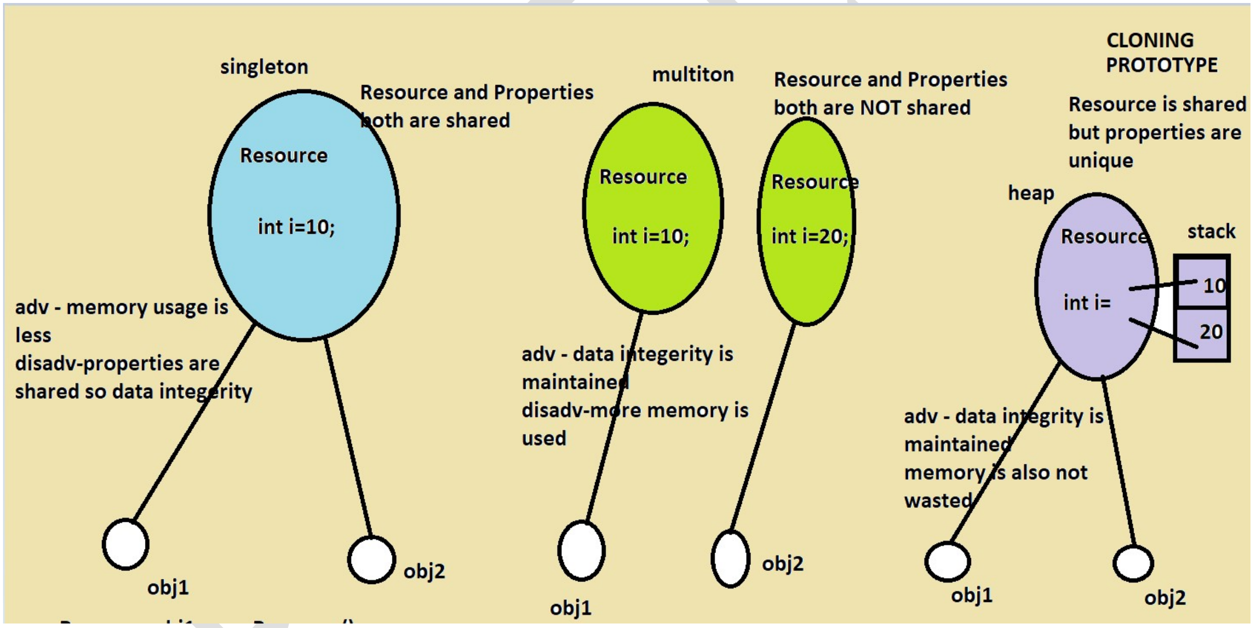
Resource class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *Resource* class to get a *SingleObject* object.

```
package dp;
/*
 * Objective - To ensure that the object creation of
particular class is
 * restricted to only one object.
 * Advantage -Multiple objects of certain classes could be a
problem, so we don't allow
 * them to create multiple objects.
 */
public class SingleTonPattern {
	public static void main(String[] args) {
		Resource obj=Resource.createSingleTonInstance();
		System.out.println(obj);

		Resource obj2=Resource.createSingleTonInstance();
		System.out.println(obj2);

	}
}

class Resource{
	private static Resource obj;
	static {
		System.out.println("static block can be called for
initialization");
	}
	private Resource() {
		System.out.println("resource object created...");
	}
synchronized public static Resource createSingleInstance() {
            if(obj==null) {
                  obj=new Resource();
            }
            return obj;
      }
}
```

Synchronized – This keyword is used here to ensure to avoid double dip problem. In case of two threads accessing this static method, we may encounter double dip problem.

Prototype

Prototype pattern refers to creating clone objects while keeping performance in mind. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.


singleton
Resource and Properties both are shared
Resource
int i=10;
adv - memory usage is less
disadv-properties are shared so data integrity
obj1
obj2
multiton
Resource and Properties both are NOT shared
Resource
int i=10;
Resource
int i=20;
adv - data integerity is maintained
disadv-more memory is used
obj1
obj2
CLONING PROTOTYPE
Resource is shared but properties are unique
heap
stack
Resource
int i=
10
20
adv - data integrity is maintained
memory is also not wasted
obj1
obj2

```
package dp;

public class PrototypeDemo {
      public static void main(String[] args)throws Exception {
            //singleton - Resources and Properties both are
shared
//		Sheep mothersheep=new Sheep();
//		Sheep dolly=mothersheep;
//
//		mothersheep.name="iam the mother...";
//		System.out.println(dolly.name);

		//multiton - Resources and properties both are 
unique
//		Sheep mothersheep=new Sheep();
//		Sheep dolly=new Sheep();
//		mothersheep.name="iam the mother..";
//		System.out.println(dolly.name);

		//Prototype/clone - Resources are shared but 
properties are unique
		Sheep mothersheep=new Sheep();
		Sheep dolly=mothersheep.createClone();

		mothersheep.name="i am mothersheep";
		dolly.name="iam the clone dolly...";
		System.out.println(dolly.name);
		System.out.println(mothersheep.name);
	}
}

class Sheep implements Cloneable{
	String name;
	public Sheep() {
		System.out.println("sheep object created...");
	}

	public Sheep createClone()throws Exception {
		return (Sheep)super.clone();
	}

}
```

Builder Pattern

Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

```
package dp;

public class BuilderPattern2 {
    public static void main(String[] args) {
        Food myfood=new
Food.Swiggy().setFoodItem("mutton...").setPacking("giftpack.
..").build();

        System.out.println(myfood);
    }
}

class Food{
    private String fooditem;
    private String packing;
    private int cost;
    private String address;
    @Override
    public String toString() {
        return fooditem+":"+packing+":"+cost+":"+address;
    }
    public Food(Swiggy swiggy) {
        this.fooditem=swiggy.fooditem;
        this.packing=swiggy.packing;
        this.cost=swiggy.cost;
        this.address=swiggy.address;
    }
    static class Swiggy{
        private String fooditem;
        private String packing;
        private int cost;
        private String address;
```

```
          public Swiggy setFoodItem(String fooditem) {
               this.fooditem=fooditem;
               return this;
          }
          public Swiggy setPacking(String packing) {
               this.packing=packing;
               return this;
          }
          //....

          public Food build() {
               return new Food(this);
          }
     }
}

Example 2

package dp;

public class BuilderPattern {
     public static void main(String[] args) {
          Computer mycomputer=new
Computer.BuildComputer().setCabinet("glass
cabinet").build();
          System.out.println(mycomputer);

     }
}
//builder pattern
class Computer{
     private String cabinet;
     private String ram;
     private String cpu;
     private String harddisk;
     public Computer(BuildComputer bc) {
          this.cabinet=bc.cabinet;
```

```
		this.ram=bc.ram;
		this.harddisk=bc.harddisk;
		this.cpu=bc.cpu;
	}
	@Override
	public String toString() {
		return 
this.cabinet+":"+this.ram+":"+this.cpu+":"+this.harddisk;
	}
	static class BuildComputer{
		private String cabinet;
		private String ram;
		private String cpu;
		private String harddisk;
		
		public BuildComputer setCabinet(String cabinet) {
			this.cabinet=cabinet;
			return this;
		}
		
		public Computer build() {
			return new Computer(this);
		}
	}
}
```

Factory, Factory Method, Abstract Factory Pattern

To create objects we use factories in real life and bringing them under a hierarchy we get all the advantages of inheritance, so rather than creating a single factory, we always try to create abstract factories which gives us the 6 advantages of inheritance.

1. Code Reusability
2. Object Reusability
3. Part whole hierarchial classification
4. Composition
5. Removal of if-else-if
6. Polymorphic queries

And factory method should be a method which should bring total transparency, which means it should be either implemented through builder pattern or command pattern.

```java
package dp;

public class FactoryPattern {
	public static void main(String[] args) {
		CakeShop myshop=new CakeShop();
		myshop.setCakeFactory(new RaniCakeFactory());
		System.out.println(myshop.sellCake());
	}
}
//Bad cake shop
class BadCakeShop{
	public Cake sellCake(int i) {
		if(i==1) {
			RajaCakeFactory rc=new RajaCakeFactory();
			return rc.makeCake();
		}
		else if(i==2) {
			RaniCakeFactory rani=new RaniCakeFactory();
			return rani.makeCake();
		}
		else {
			return null;
		}
	}
}

class CakeShop{
	CakeFactory cakeFactory;
	public void setCakeFactory(CakeFactory cakeFactory) {
		this.cakeFactory=cakeFactory;
	}
	public Cake sellCake() {
		return cakeFactory.makeCake();
	}
}
```

```
abstract class CakeFactory{
	public abstract Cake makeCake();
}
class RajaCakeFactory extends CakeFactory{
	public Cake makeCake() {
		return new
Cake.CakeMaker().setCakeType("forest.raja.").setShape("squar
e..")
				.setWeight(1).build();
	}
}
class RaniCakeFactory extends CakeFactory{
	public Cake makeCake() {
		return new
Cake.CakeMaker().setCakeType("softy.rani.").setShape("round.
.")
				.setWeight(2).build();
	}
}
class Cake{
	int weight;
	String shape;
	String caketype;
	@Override
	public String toString() {
		return "The cake
is..:"+caketype+"weight..:"+weight+":shape.."+shape;
	}
	public Cake(CakeMaker cm) {
		weight=cm.weight;
		shape=cm.shape;
		caketype=cm.caketype;
	}
	static class CakeMaker{
		int weight;
		String shape;
```

```
        String caketype;
        public CakeMaker setWeight(int weight) {
            this.weight=weight;
            return this;
        }
        public CakeMaker setShape(String shape) {
            this.shape=shape;
            return this;
        }
        public CakeMaker setCakeType(String caketype) {
            this.caketype=caketype;
            return this;
        }
        public Cake build() {
            return new Cake(this);
        }
    }
}
```

BEHAVIORAL PATTERNS

1. Chain of Responsibility
2. Command
3. Interpreter
4. Mediator
5. Strategy
6. Template Method
7. Visitor

Strategy Pattern

A very important pattern used to eliminate the if-else-if conditional statements.

Ex1

```
package dp;
//removing if-else-if conditional statements in strategy
pattern
public class StrategyPattern {
     public static void main(String[] args) {

     }
}
//super man code - bad code
class BadPaintBrush{
     public void doPaint(int paint) {
          if(paint==1) {
               RedPaint rp=new RedPaint();
               System.out.println(rp);
          }
          else if(paint==2) {
               BluePaint bp=new BluePaint();
               System.out.println(bp);
          }
     }
}
//good code
/*
 * 1. Convert the condition to classes.
 * 2. Group them under a hierarchy
 * 3. Create a association between the hierarchial class and
the using class.
 */
class PaintBrush{
     Paint paint;
     public void doPaint() {
          System.out.println(paint);
     }
}
abstract class Paint{

}

class RedPaint extends Paint{

}
class BluePaint extends Paint{

}

class GreenPaint extends Paint{

}

Ex2

package dp;

public class StrategyPattern2 {
     public static void main(String[] args) {
          Dog tiger=new Dog();
          Item item=new Stick();
          tiger.play(item);
     }
}

class BadDog{
     public void play(String item) {
          if(item.equals("stick")) {

          }
          else if(item.equals("stone")) {

          }
```

```
	}
}

//good dog
class Dog{
	public void play(Item item) {
		item.execute();
	}
}
abstract class Item{
	public abstract void execute();
}
class Stick extends Item{
	@Override
	public void execute() {
		System.out.println("you beat I bite....");
	}
}
class Stone extends Item{
	@Override
	public void execute() {
		System.out.println("you throw I catch....");
	}
}
```

State Pattern

Helps us to manage the changing state of an object without using if-else-if ladder.

```
package dp;
public class StatePattern {

}
```

```
//super man - kirukku code
class BadFan{
	int state=0;
	public void pull() {
		if(state==0) {
			System.out.println("switch on state....");
			state=1;
		}
		else if(state==1) {
			System.out.println("low speed state....");
			state=2;
		}
		else if(state==2) {
			System.out.println("Medium speed state....");
			state=3;
		}
		else if(state==3) {
			System.out.println("high speed state....");
			state=0;
		}
	}
}

//Good code
class GoodFan{
	/*
	 * To eliminate the conditions, you need to convert the
condition to classes
	 */
	State state=new SwitchOffState();
	public void pull() {
		state.changeFanState(this);
	}
}

abstract class State{
	public abstract void changeFanState(GoodFan fan);
}

class SwitchOffState extends State{
	@Override
	public void changeFanState(GoodFan fan) {
		System.out.println("switch on state....");
		fan.state=new SwitchOnState();
	}
}
class SwitchOnState extends State{
	@Override
	public void changeFanState(GoodFan fan) {
		System.out.println("Low Speed State....");
		fan.state=new LowSpeedState();
	}
}
class LowSpeedState extends State{
	@Override
	public void changeFanState(GoodFan fan) {
		System.out.println("Medium Speed State...");
		fan.state=new MediumSpeedState();
	}
}
class MediumSpeedState extends State{
	@Override
	public void changeFanState(GoodFan fan) {
		System.out.println("High speed state...");
		fan.state=new HighSpeedState();
	}
}
class HighSpeedState extends State{
	@Override
	public void changeFanState(GoodFan fan) {
		System.out.println("switch off state...");
		fan.state=new SwitchOffState();
	}
}
```

Template Method

This method is excellent example for abstract methods, it defines a template method which could be reused.

Ex1

```
package revision;

public class TemplateMethod {
        public static void main(String[] args) {
            HouseTemplate houseType = new WoodenHouse();

            //using template method
            houseType.buildHouse();
            System.out.println("************");

            houseType = new GlassHouse();

            houseType.buildHouse();
        }
}
abstract class HouseTemplate {

    //template method, final so subclasses can't override
    public final void buildHouse(){
        buildFoundation();
        buildPillars();
        buildWalls();
        buildWindows();
        System.out.println("House is built.");
    }

    //default implementation
    private void buildWindows() {
        System.out.println("Building Glass Windows");
```

```
     }

     //methods to be implemented by subclasses
     public abstract void buildWalls();
     public abstract void buildPillars();

     private void buildFoundation() {
          System.out.println("Building foundation with
cement,iron rods and sand");
     }
}
class GlassHouse extends HouseTemplate {

     @Override
     public void buildWalls() {
          System.out.println("Building Glass Walls");
     }

     @Override
     public void buildPillars() {
          System.out.println("Building Pillars with glass
coating");
     }

}

class WoodenHouse extends HouseTemplate {

     @Override
     public void buildWalls() {
          System.out.println("Building Wooden Walls");
     }

     @Override
     public void buildPillars() {
          System.out.println("Building Pillars with Wood
coating");
```

```
	}

}

Ex 2:

package dp;

public class TemplateMethod {
	public static void main(String[] args) {
		DominoPizza pizza=new RahimPizza();
		pizza.doPizzaBusiness();//we should call only the
template method
	}
}

abstract class DominoPizza{
	public final void shape() {
		System.out.println("round round round....");
	}
	public final void makeDominoPizza() {
		System.out.println("as per dominos logic....");
	}
	//the below methods, you should override..its compulsory
	public abstract void collectMoney();
	public abstract void deliveryPizza();
	public final void doPizzaBusiness() {//template method
		collectMoney();
		shape();
		makeDominoPizza();
		deliveryPizza();
	}
}

class RahimPizza extends DominoPizza{

	public void collectMoney() {
		System.out.println("If iam in india, i will collect 
in rupees");
	}
	
	public void deliveryPizza() {
		System.out.println("If iam in russia I will deliver 
pizza in car......");
	}
}
```

Visitor Pattern

This pattern is a classic example of polymorphism, based on the object which another object receives it starts behaving accordingly.

Ex1:

```
package revision;

public class Visitor {
	public static void main(String[] args) {
		ItemElement[] items = new ItemElement[]{new 
Book(20, "1234"),new Book(100, "5678"),
				new Fruit(10, 2, "Banana"), new Fruit(5, 
5, "Apple")};
		
		int total = calculatePrice(items);
		System.out.println("Total Cost = "+total);
	}

	private static int calculatePrice(ItemElement[] items) {
		ShoppingCartVisitor visitor = new 
ShoppingCartVisitorImpl();
		int sum=0;
		for(ItemElement item : items){
			sum = sum + item.accept(visitor);
```

```
          }
          return sum;
     }

}
interface ItemElement {

     public int accept(ShoppingCartVisitor visitor);
}
class ShoppingCartVisitorImpl implements ShoppingCartVisitor
{

     @Override
     public int visit(Book book) {
          int cost=0;
          //apply 5$ discount if book price is greater than
50
          if(book.getPrice() > 50){
               cost = book.getPrice()-5;
          }else cost = book.getPrice();
          System.out.println("Book
ISBN::"+book.getIsbnNumber() + " cost ="+cost);
          return cost;
     }

     @Override
     public int visit(Fruit fruit) {
          int cost = fruit.getPricePerKg()*fruit.getWeight();
          System.out.println(fruit.getName() + " cost =
"+cost);
          return cost;
     }

}

interface ShoppingCartVisitor {
	int visit(Book book);
	int visit(Fruit fruit);
}

class Fruit implements ItemElement {

	private int pricePerKg;
	private int weight;
	private String name;

	public Fruit(int priceKg, int wt, String nm){
		this.pricePerKg=priceKg;
		this.weight=wt;
		this.name = nm;
	}

	public int getPricePerKg() {
		return pricePerKg;
	}


	public int getWeight() {
		return weight;
	}

	public String getName(){
		return this.name;
	}

	@Override
	public int accept(ShoppingCartVisitor visitor) {
		return visitor.visit(this);
	}

}
class Book implements ItemElement {
	private int price;
	private String isbnNumber;
	
	public Book(int cost, String isbn){
		this.price=cost;
		this.isbnNumber=isbn;
	}
	
	public int getPrice() {
		return price;
	}
	
	public String getIsbnNumber() {
		return isbnNumber;
	}
	
	@Override
	public int accept(ShoppingCartVisitor visitor) {
		return visitor.visit(this);
	}
}
```

Ex2:

```
package revision;

public class Visitor2 {
	public static void main(String[] args) {
		Child ramu=new Child();
		Dog tiger=new Dog();
		
		Item item=new Stone();
		
		ramu.playWithDog(tiger, item);
	}
}
```

```
abstract class DogExceptions extends Exception{
	public abstract void visit();
}
class DogBiteException extends DogExceptions{
	String msg;
	public DogBiteException(String msg) {
		this.msg=msg;
	}
	@Override
	public String toString() {
		return this.msg;
	}
	@Override
	public void visit() {
		// TODO Auto-generated method stub
		new Handler().handle(this);
	}
}
class DogBarkException extends DogExceptions{
	String msg;
	public DogBarkException(String msg) {
		this.msg=msg;
	}
	@Override
	public String toString() {
		return this.msg;
	}
	@Override
	public void visit() {
		new Handler().handle(this);
	}
}
class Handler{
	public void handle(DogBiteException dbe) {
		System.out.println("handling bite 
exception...taking to hospital...:"+dbe);
	}
```

```
     public void handle(DogBarkException dre) {
          System.out.println("handling barking 
exception..being silent...:"+dre);
     }
}
abstract class Item{
     public abstract void execute()throws DogExceptions;
}
class Stick extends Item{
     @Override
     public void execute() throws DogExceptions {
          throw new DogBiteException("You beat I bite.....");
     }
}
class Stone extends Item{
     @Override
     public void execute() throws DogExceptions {
          throw new DogBarkException("you hit I 
bark.........");
     }
}
class Dog{
     public void play(Item item) throws DogExceptions{
          item.execute();
     }
}
class Child{
     public void playWithDog(Dog tiger,Item item) {
          try {
               tiger.play(item);
          }catch(DogExceptions de) {
               System.out.println(de);
               de.visit();
          }
     }
}
```

Command

Command Pattern brings in transparency, it gives the end user the ease of working with complex objects, especially in case of IOT, the situation becomes really complex in executing a repetitive command.
By way of command pattern you bring in a layer of abstraction and hide the complexity of the operation so that the end user gets the job done with no difficulties.

Example 1

```java
package dp;

public class CommandPattern {
	public static void main(String[] args) {
		//ChithiCitizen cc=new ChithiCitizen();
		//cc.getDeathCertificate();
		
		AmmaCitizen ac=new AmmaCitizen();
		ac.getDeathCertificate();
	}
}
//bad
class ChithiCitizen{
	public void getDeathCertificate() {
		Hospital hospital=new Hospital();
		hospital.doPostMortem();
		Police police=new Police();
		police.doInvestigation();
		Corporation corporation=new Corporation();
		corporation.issueDeathCertificate();
	}
}
//Good code using command pattern
class AmmaCitizen{
	static SevaCenter sc;
	static {
		sc=new SevaCenter();
		sc.setCommand(1, new Governor());
	}
	public void getDeathCertificate() {

		sc.executeCommand(1);
	}
}
class SevaCenter{
	Command c[]=new Command[5];
	public SevaCenter() {
		for(int i=0;i<5;i++) {
			c[i]=new DummyCommand();
		}
	}
	public void setCommand(int slot,Command command) {
		c[slot]=command;
	}
	public void executeCommand(int slot) {
		c[slot].execute();
	}
}
abstract class Command{
	public abstract void execute();
}
class DummyCommand extends Command{
	@Override
	public void execute() {
		System.out.println("i am dummy yet to be
operation...");
	}
}

class Governor extends Command{
	@Override
	public void execute() {
		Hospital hospital=new Hospital();
		hospital.doPostMortem();
		Police police=new Police();
```

```
        police.doInvestigation();
        Corporation corporation=new Corporation();
        corporation.issueDeathCertificate();
    }
}

class Hospital{
    public void doPostMortem() {
        System.out.println("post mortem done....");
    }
}
class Police{
    public void doInvestigation() {
        System.out.println("investigation done...");
    }
}
class Corporation{
    public void issueDeathCertificate() {
        System.out.println("death certificate issued....");
    }
    public void clearanceCertificate() {
        System.out.println("clearance certificate
issued...");
    }
}
```

Example 2

```
package dp;

public class CommandPattern2 {
    public static void main(String[] args) {
        //BadSon shoiab=new BadSon();
        //shoiab.serveFather();
        Servant servant=new Servant();
        servant.setCommand(new EatKadalaUrundaCommand(),
1);
```

```
		servant.setCommand(new PlayTTCommand(), 2);
		servant.setCommand(new PlayTTDoublesCommand(), 3);
		GoodSon shoiab=new GoodSon();
		shoiab.serveFather(servant);
	}
}

class BadSon{
	public void serveFather() {
		Father myfather=new Father();
		myfather.eatkadalaUrunda();
		myfather.playTTGame();
	}
}

class GoodSon{
	public void serveFather(Servant servant) {
		KingFather myfather=new KingFather();
		myfather.eatkadalaUrunda(servant);
		myfather.playTTGame(servant);
	}
}
class KingFather{
	public void eatkadalaUrunda(Servant servant) {
		servant.executeCommand(1);
	}
	public void playTTGame(Servant servant) {
		servant.executeCommand(2);
	}
	public void playTTDoublesGame(Servant servant) {
		servant.executeCommand(3);
	}

}
class Servant{
	Command2 c[]=new Command2[5];
	public Servant() {
		for(int i=0;i<5;i++) {
			c[i]=new DummyCommand2();
		}
	}
	public void executeCommand(int slot) {
		c[slot].execute();
	}
	public void setCommand(Command2 command,int slot) {
		c[slot]=command;
	}
}

abstract class Command2{
	public abstract void execute();
}
class DummyCommand2 extends Command2{
	@Override
	public void execute() {
		System.out.println("i dont know this command 
sir...");
	}
}

class EatKadalaUrundaCommand extends Command2{
	@Override
	public void execute() {
		Lift lift=new Lift();
		lift.operateLift();
		Car car=new Car();
		car.driveCar();
		PottiKadai kadai=new PottiKadai();
		kadai.buyKadalaUrunda();
	}
}
class PlayTTCommand extends Command2{
	@Override
	public void execute() {
```

```
          Tv tv=new Tv();
          tv.switchOn();
          tv.changeChannel();
          VGame vgame=new VGame();
          vgame.playTTGameSingles();
     }
}
class PlayTTDoublesCommand extends Command2{
     @Override
     public void execute() {
          Tv tv=new Tv();
          tv.switchOn();
          tv.changeChannel();
          VGame vgame=new VGame();
          vgame.playTTGameDoubles();
     }
}
class Father{
     public void eatkadalaUrunda() {
          Lift lift=new Lift();
          lift.operateLift();
          Car car=new Car();
          car.driveCar();
          PottiKadai kadai=new PottiKadai();
          kadai.buyKadalaUrunda();
     }
     public void playTTGame() {
          Tv tv=new Tv();
          tv.switchOn();
          tv.changeChannel();
          VGame vgame=new VGame();
          vgame.playTTGameDoubles();
     }
}

class Car{
     public void driveCar() {
            System.out.println("driving car....");
        }
}

class Tv{
        public void switchOn() {
            System.out.println("TV switch on...");
        }
        public void changeChannel() {
            System.out.println("channel changed....");
        }
}

class VGame{
        public void playTTGameSingles() {
            System.out.println("TT game singles...");
        }
        public void playTTGameDoubles() {
            System.out.println("TT game doubles...");
        }
}

class PottiKadai{
        public void buyKadalaUrunda() {
            System.out.println("buy and eat kadala urunda...");
        }
}
class WalkingTrack{
        public void walk() {
            System.out.println("walking on the walking
track..");
        }
}

class Lift{
        public void operateLift() {
            System.out.println("operating lift...");
```

```
	}
}
```

Decorator or Composite Pattern

This behavioral pattern is intended to bring dynamic binding, we decorate one object with another object to make a whole object. This pattern is extensively used in JDK IO package.

```
package dp;

public class DecoratorOrComposite {

	public static void main(String[] args) {
		IbacoIceCream iic=new Vanila(new Nuts(new Nuts(new
Gems())));
		System.out.println(iic);
		System.out.println("The cost of your
icecream..:"+iic.cost());
	}
}

abstract class IbacoIceCream{
	public abstract int cost();
}

abstract class Cream extends IbacoIceCream{

}
class Vanila extends Cream{
	IbacoIceCream iic;
	public Vanila() {
		// TODO Auto-generated constructor stub
	}
	public Vanila(IbacoIceCream iic) {
		this.iic=iic;
	}
	@Override
	public String toString() {
		return "vanila...:"+iic;
	}
	@Override
	public int cost() {
		return (iic==null)?10:10+iic.cost();
	}
}
class Starberry extends Cream{
	IbacoIceCream iic;
	public Starberry() {
		// TODO Auto-generated constructor stub
	}
	public Starberry(IbacoIceCream iic) {
		this.iic=iic;
	}
	@Override
	public String toString() {
		return "Starberry...:"+iic;
	}
	@Override
	public int cost() {
		return (iic==null)?20:20+iic.cost();
	}
}
abstract class AddOns extends IbacoIceCream{

}
class Nuts extends AddOns{
	IbacoIceCream iic;
	public Nuts() {
		// TODO Auto-generated constructor stub
	}
	public Nuts(IbacoIceCream iic) {
		this.iic=iic;
	}
	@Override
	public String toString() {
		return "Nuts...:"+iic;
	}
	@Override
	public int cost() {
		return (iic==null)?5:5+iic.cost();
	}
}
class ChocoChips extends AddOns{
	IbacoIceCream iic;
	public ChocoChips() {
		// TODO Auto-generated constructor stub
	}
	public ChocoChips(IbacoIceCream iic) {
		this.iic=iic;
	}
	@Override
	public String toString() {
		return "ChocoChips...:"+iic;
	}
	@Override
	public int cost() {
		return (iic==null)?5:5+iic.cost();
	}
}
class Gems extends AddOns{
	IbacoIceCream iic;
	public Gems() {
		// TODO Auto-generated constructor stub
	}
	public Gems(IbacoIceCream iic) {
		this.iic=iic;
	}
	@Override
	public String toString() {
		return "Gems...:"+iic;
	}
```

```
    @Override
    public int cost() {
        return (iic==null)?15:15+iic.cost();
    }
}
```

Adapter

```
package revision;

public class Adapter {
    public static void main(String[] args) {
        AmericanPlug lenovo=new LenovoPlug();

        IndianSocket socket=new ShakthiSocket();

        IndianAdapter ip=new IndianAdapter();
        ip.plug=lenovo;

        socket.roundPinHole(ip);

    }
}

abstract class AmericanPlug {
    public abstract void work();
}

class IndianAdapter extends IndianPlug{
    AmericanPlug plug;
    @Override
    public void work() {
        plug.work();
    }
}
```

```
abstract class IndianPlug {
	public abstract void work();
}

abstract class IndianSocket {
	public abstract void roundPinHole(IndianPlug ip);
}

class LenovoPlug extends AmericanPlug{
	@Override
	public void work() {
		System.out.println("american plug
working.........");
	}
}

class ShakthiSocket extends IndianSocket{
	@Override
	public void roundPinHole(IndianPlug ip) {
		ip.work();
	}
}
```

Bridge

```
package revision;

public class Bridge {
	public static void main(String[] args) {
		Shape tri = new Triangle(new RedColor());
		tri.applyColor();

		Shape pent = new Pentagon(new GreenColor());
		pent.applyColor();
	}
}
```

Haaris Infotech
Driven by Technology

```
class GreenColor implements Color{

	public void applyColor(){
		System.out.println("green.");
	}
}
class RedColor implements Color{

	public void applyColor(){
		System.out.println("red.");
	}
}
interface Color {

	public void applyColor();
}
abstract class Shape {
	//Composition - implementor
	protected Color color;

	//constructor with implementor as input argument
	public Shape(Color c){
		this.color=c;
	}

	abstract public void applyColor();
}
class Pentagon extends Shape{

	public Pentagon(Color c) {
		super(c);
	}

	@Override
	public void applyColor() {
		System.out.print("Pentagon filled with color ");
		color.applyColor();
```

```
    }

}

class Triangle extends Shape{

    public Triangle(Color c) {
        super(c);
    }

    @Override
    public void applyColor() {
        System.out.print("Triangle filled with color ");
        color.applyColor();
    }

}
```