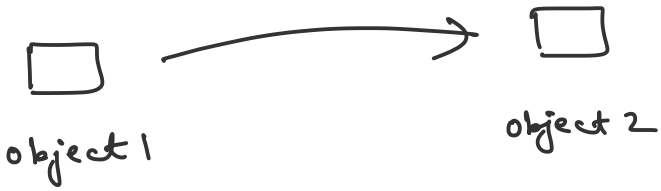
16__03-09-2022

Saturday, 3 September 2022 8:05 PM

Cloneable interface




```
public interface Cloneable
{
    public Object clone();
}
```

This provides support for cloning

Dept → Abstract class

CSE ✓

ECE ✓

Concrete classes.

Step 1 - Create an abstract class implementing Cloneable interface.

Dept ⟶ implements Cloneable

- getDeptType()
- getDeptId()
- setDeptId()

} non-abstract method

- DeptRank(); } Abstract method
- clone() } Cloneable interface

Step 2 - Create concrete class - CSE - extending abstract class Dept.

Step 3 - Create concrete class ECE - extending abstract class Dept.

Step 4 - Create Dept Cache class.

public class Dept Cache

- getDeptDetails() → Creating Backup
- loadCache() → Restoring the backup

Step 5 - Creating main method.

```
public abstract class Dept implements Cloneable {

   private String deptId;
   protected String deptType;

   //non abstract methods
   public String getDeptType()
   {
      return deptType;
   }

   public String getDeptId()
   {
      return deptId;
   }

   public void setDeptId(String deptId)
   {
      this.deptId = deptId;
   }

   //abstract methods
   abstract void rank();
```

```
    //as a part of cloneable interface
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

```
public class Cse extends Dept {
    public Cse()
    {
        deptType = "CSE";
    }

    @Override
    void rank() {
        System.out.println("Inside rank method - CSE class.");
    }
}
```

```
public class Ece extends Dept {
    public Ece()
    {
        deptType = "ECE";
    }

    @Override
    void rank() {
        System.out.println("Inside rank method - ECE class.");
    }
}
```

```
import java.util.Hashtable;

public class DeptCache {
```

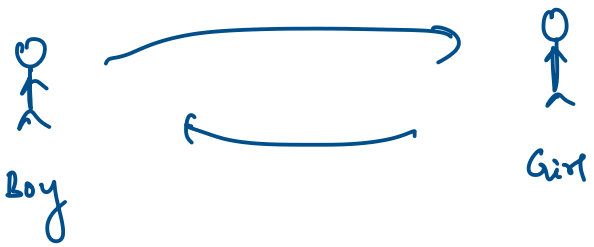
```
    private static Hashtable<String, Dept> deptHashtable = new Hashtable<>();

    //acting like creating backup
    /*
    here we are creating dept cache based upon dept id
    * */
    public static Dept getDeptDetails(String deptId) throws
CloneNotSupportedException {
        Dept cachedDept = deptHashtable.get(deptId);
        return (Dept) cachedDept.clone();
    }

    //acting like restoring backup
    /*
    here we are creating cache hashtable
    * */
    public static void loadDeptCache()
    {
        Cse cse = new Cse();
        cse.setDeptId("CSE");
        deptHashtable.put(cse.getDeptId(), cse);

        Ece ece = new Ece();
        ece.setDeptId("ECE");
        deptHashtable.put(ece.getDeptId(), ece);
    }
}




public class Program {
    public static void main(String[] args) throws CloneNotSupportedException {
        //load all the backups created
        DeptCache.loadDeptCache();

        Dept clonedEce = DeptCache.getDeptDetails("ECE");
        clonedEce.rank();
        System.out.println("Dept name : " + clonedEce.getDeptType());

        Dept clonedCse = DeptCache.getDeptDetails("CSE");
        clonedCse.rank();
        System.out.println("Dept name : " + clonedCse.getDeptType());
    }
}
```

Output:
Inside rank method - ECE class.
Dept name : ECE
Inside rank method - CSE class.
Dept name : CSE

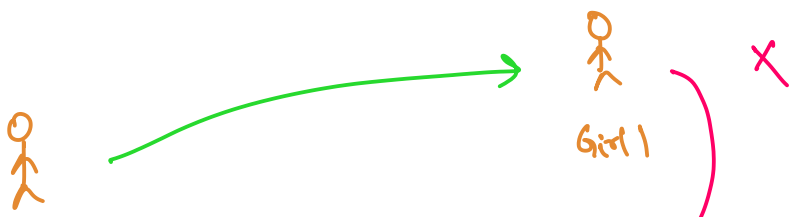
Behavioural design pattern

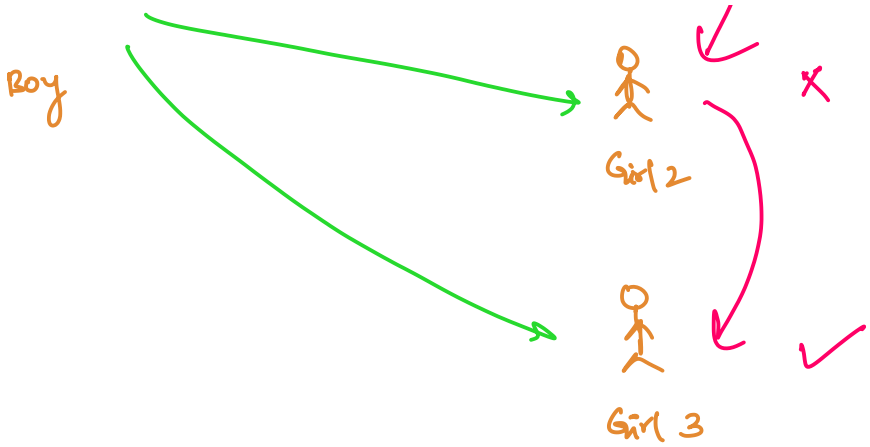
Chain of responsibility design pattern

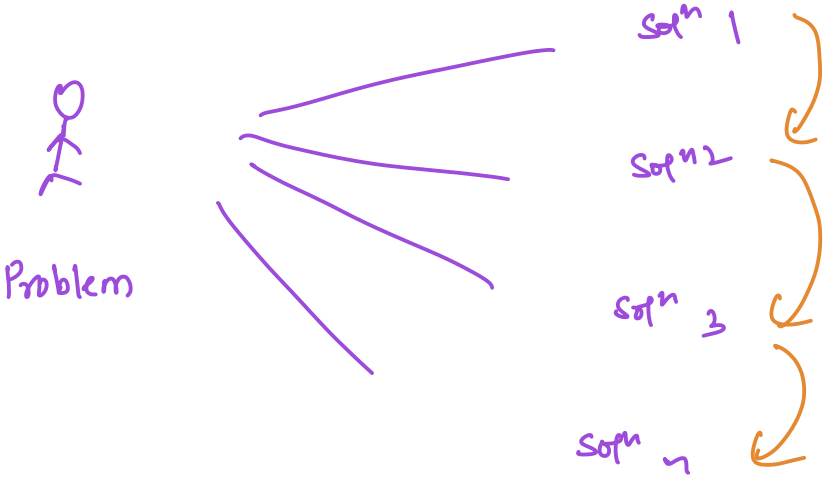
old lovers



Modern lovers | practical lovers | Tinder generation.








```
try
{
    =
    =
}

catch ( Arithmetic Exception  ex)
{
    =
    =
}

catch ( Array Out Of Bound Exception  ex)
```

```
{
    =
}
```

```
catch ( Exception ex)
{
    =
}
```

Chain of responsibility

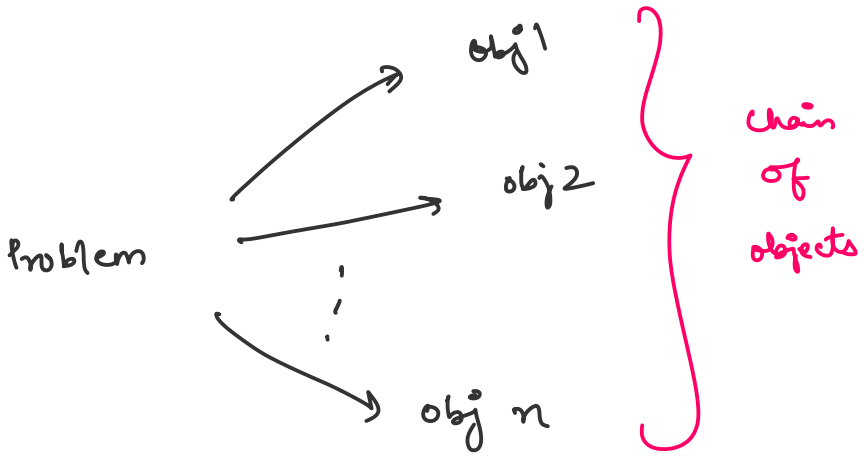
↳ Avoiding if-else hell

Gang of four

↳ Too much of if-else is a bad smell

In the chain of responsibility, the sender sends a request to a chain of

objects.



The request can be handled by any object in the chain.

This pattern creates a chain of receiver objects for a request, this pattern decouples sender and receiver of a request based on the type of request.

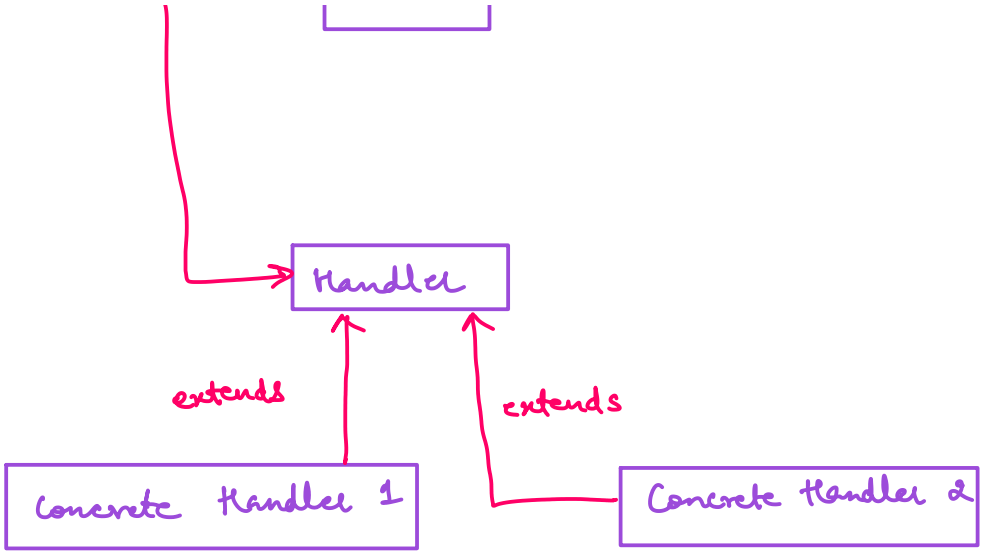
Avoid coupling the sender of a request

to its receiver by giving multiple objects a chance to handle the request.

Eg- ATM machine

1. Insert the card
2. Enter the pin
3. Select options- pin change, withdraw, etc
4. Select withdraw
5. Choose account type- current, savings
6. Enter amount.

Client



Client

↳ originator of the request and for this request we can have multiple responses.

Handler

↳ This can be an interface which will primarily receive the request and dispatches the request to the chain of handlers. It has reference to the only first handler in the chain

and does not know anything about the rest of the handlers.

Concrete handler

↳ These are actual handlers of the request chained in some sequential order.

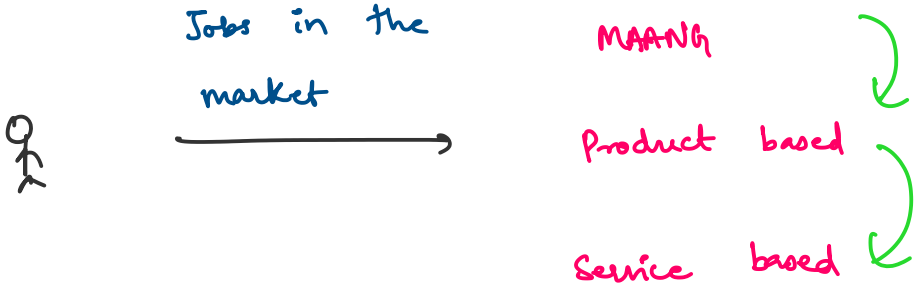
Advantages –

1. It reduces coupling
2. It adds flexibility while assigning the responsibilities to objects.
3. It allows a set of classes to act as one, events produced in one class can

be sent to the other handler classes

with the help of composition.

Eg 1



Step 1 – Create person class.

Person

↳ skill type

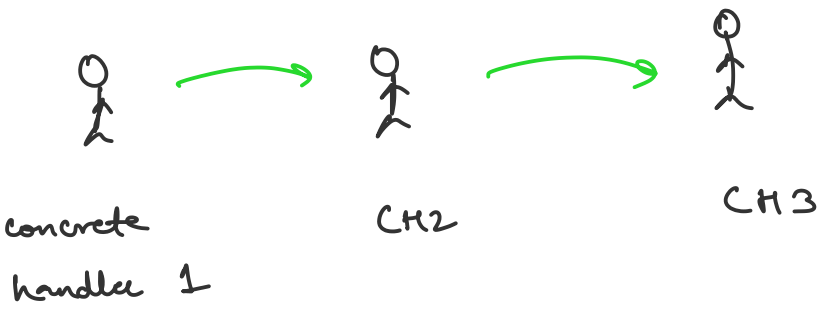
Step 2 – Create I Job Chain

I Job Chain

- Setting the next job
- Helping the person to get job

Step 3 – Create concrete handler – MAANG,

implementing IJobChain interface



They should know the chain they are part of.

step 4- Create concrete handlers- product and service based.

step 5 Create client class.

↳ Create chain

MAANG → product → service

↳ Based upon the Person's skill type get the job.

```
public interface IJobChain {
   //setting the next job
   public void setNextChain(IJobChain jobChain);

   /*
   Helps person to get job
   1. MAANG
   2. Product based
   3. Service based
   * */
   public void getJob(Person person);
}
```

```
public class Person {
   private String skillType;

   public Person(String skillType)
   {
      this.skillType = skillType;
   }

   public String getSkillType()
   {
      return skillType;
   }
}
```

```
public class MaangGroup implements IJobChain {
```

```
    //knows who is the next in the job chain
    private IJobChain jobChain;

    //this method will be used for setting the next job
    @Override
    public void setNextChain(IJobChain jobChain) {
       this.jobChain = jobChain;
    }

    //this method will be used for getting the job
    @Override
    public void getJob(Person person) {
       if(person.getSkillType().equals("MAANG"))
       {
          System.out.println("Got selected in MAANG group");
       }
       else {
          System.out.println("Got rejected in MAANG group");
          System.out.println("Applying for product based companies");
          this.jobChain.getJob(person);
       }
    }
}
```

```
public class ProductBased implements IJobChain {
    //knows who is the next in the job chain
    private IJobChain jobChain;

    //this method will be used for setting the next job
    @Override
    public void setNextChain(IJobChain jobChain) {
       this.jobChain = jobChain;
    }

    //this method will be used for getting the job
    @Override
    public void getJob(Person person) {
       if(person.getSkillType().equals("Product"))
       {
          System.out.println("Got selected in product based company");
       }
       else {
          System.out.println("Got rejected in product based company");
          System.out.println("Applying for service based companies");
          this.jobChain.getJob(person);
```

```
        }
    }
}
```

```
public class ServiceBased implements IJobChain {
    //knows who is the next in the job chain
    private IJobChain jobChain;

    //this method will be used for setting the next job
    @Override
    public void setNextChain(IJobChain jobChain) {
        this.jobChain = jobChain;
    }

    //this method will be used for getting the job
    @Override
    public void getJob(Person person) {
        if(person.getSkillType().equals("Service"))
        {
            System.out.println("Got selected in service based company");
        }
        else {
            System.out.println("Got rejected in service based company");
            System.out.println("Need to prepare more");
        }
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        MaangGroup maangJob = new MaangGroup();

        ProductBased productBasedJob = new ProductBased();

        ServiceBased serviceBasedJob = new ServiceBased();

        /*
        Maang --> product --> service
        * */
```

```
        maangJob.setNextChain(productBasedJob);
        productBasedJob.setNextChain(serviceBasedJob);

        //person with no skills
        System.out.println("For person 1 with no skills");
        Person person1 = new Person("");
        maangJob.getJob(person1);
        System.out.println();

        //person with service as skill type
        System.out.println("For person 2 with service as skill type");
        Person person2 = new Person("Service");
        maangJob.getJob(person2);
        System.out.println();

        //person with product as skill type
        System.out.println("For person 3 with product as skill type");
        Person person3 = new Person("Product");
        maangJob.getJob(person3);
        System.out.println();

        //person with maang as skill type
        System.out.println("For person 4 with maang as skill type");
        Person person4 = new Person("MAANG");
        maangJob.getJob(person4);
    }
}
```

Output:

```
For person 1 with no skills
Got rejected in MAANG group
Applying for product based companies
Got rejected in product based company
Applying for service based companies
Got rejected in service based company
Need to prepare more

For person 2 with service as skill type
Got rejected in MAANG group
Applying for product based companies
Got rejected in product based company
Applying for service based companies
Got selected in service based company

For person 3 with product as skill type
Got rejected in MAANG group
Applying for product based companies
```

Got selected in product based company

For person 4 with maang as skill type
Got selected in MAANG group