ABOUT ROHIT JOSHI

Rohit Joshi is a Senior Software Engineer from India. He is a Sun Certified Java Programmer and has worked on projects related to different domains. His expertise is in Core Java and J2EE technologies, but he also has good experience with front-end technologies like Javascript, JQuery, HTML5, and JQWidgets.

Chain of Responsibility Design Pattern

Posted by: Rohit Joshi in Core Java September 30th, 2015

This article is part of our Academy Course titled Java Design Patterns.

In this course you will delve into a vast number of Design Patterns and see how those are implemented and utilized in Java. You will understand the reasons why patterns are so important and learn when and how to apply each one of them. Check it out here!

Table Of Contents

1. Chain of Responsibility Pattern

The Chain of Responsibility pattern is a behavior pattern in which a group of objects is chained together in a sequence and a responsibility (a request) is provided in order to be handled by the group. If an object in the group can process the particular request, it does so and returns the corresponding response. Otherwise, it forwards the request to the subsequent object in the group.

For a real life scenario, in order to understand this pattern, suppose you got a problem to solve. If you are able to handle it on your own, you will do so, otherwise you will tell your friend to solve it. If he'll able to solve he will do that, or he will also forward it to some other friend. The problem would be forwarded until it gets solved by one of your friends or all your friends have seen the problem, but no one is able to solve it, in which case the problem stays unresolved.

Let's address a real life scenario. Your company has got a contract to provide an analytical application to a health company. The application would tell the user about the particular health problem, its history, its treatment, medicines, interview of the person suffering from it etc,

Now, your job is to save this data in the company's database. Users will provide the data in any format and you should provide them a single interface to upload the data into the database. The user is not interested, not even aware, to know that how you are saving the different unstructured data?

The problem here is that you need to develop different handlers to save the various formats of data. For example, a text file save handler does not know how to save an mp3 file.

To solve this problem you can use the Chain of Responsibility design pattern. You can create different objects which process different formats of data and chain them together. When a request comes to a single object, it will check whether it can process and handle the specfic file format. If it can, it will process it; otherwise, it will forward it to the next object chained to it. This design pattern also decouples the user from the object that is serving the request; the user is not aware which object is actually serving its request.

Before solving the problem, let's first know more about the Chain of Responsibility design pattern.

2. What is the Chain of Responsibility Pattern

The intent of this pattern is to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. We chain the receiving objects and pass the request along the chain until an object handles it.

This pattern is all about connecting objects in a chain of notification; as a notification travels down the chain, it's handled by the first object that is set up to deal with the particular notification.

When there is more than one objects that can handle or fulfill a client request, the pattern recommends giving each of these objects a chance to process the request in some sequential order. Applying the pattern in such a case, each of these potential handlers can be arranged in the form of a chain, with each object having a reference to the next object in the chain. The first object in the chain receives the request and decides either to handle the request or to pass it on to the next object in the chain. The request flows through all objects in the chain one after the other until the request is handled by one of the handlers in the chain or the request reaches the end of the chain without getting processed.

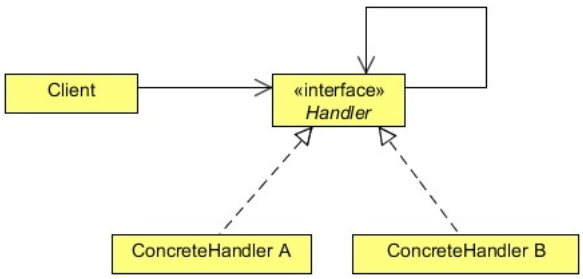

Figure 1

Handler

1. Defines an interface for handling requests.
2. (Optionally) Implements the successor link.

ConcreteHandler

1. Handles requests it is responsible for.
2. Can access its successor.
3. If the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.

Client

1. Initiates the request to a ConcreteHandler object on the chain.

When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

3. Implementing Chain of Responsibility

To implement the Chain of Responsibility in order to solve the above problem, we will create an interface, Handler.

```
1  package com.javacodegeeks.patterns.chainofresponsibility;
2
3  public interface Handler {
4
5      public void setHandler(Handler handler);
6      public void process(File file);
7      public String getHandlerName();
8  }
```

The above interface contains two main methods, the

```
setHandler
```

is used to set the next handler in the chain, whereas; the process method is used to process the request, only if the handler can able process the request. Optionally, we have the

```
getHandlerName
```

method which is used to return the handler's name.

The handlers are designed to process files which contain data. The concrete handler checks if it's able to handle the file by checking the file type, otherwise forwards to the next handler in the chain.

The

```
File
```

class looks like this.

```
01 package com.javacodegeeks.patterns.chainofresponsibility;
02
03 public class File {
04
05     private final String fileName;
06     private final String fileType;
07     private final String filePath;
08
09     public File(String fileName, String fileType, String filePath){
10         this.fileName = fileName;
11         this.fileType = fileType;
12         this.filePath = filePath;
13     }
14
15     public String getFileName() {
16         return fileName;
17     }
18
19     public String getFileType() {
20         return fileType;
21     }
22
23     public String getFilePath() {
24         return filePath;
25     }
26
27 }
```

The

```
File
```

class creates simple file objects which contain the file name, file type, and the file path. The file type would be used by the handler to check if the file can be handled by them or not. If a handler can, it will process and save it, or it will forward it to the next handler in the chain.

Let's see some concrete handlers now.

```
01 package com.javacodegeeks.patterns.chainofresponsibility;
02
03 public class TextFileHandler implements Handler {
04
05     private Handler handler;
06     private String handlerName;
07
08     public TextFileHandler(String handlerName){
09         this.handlerName = handlerName;
10     }
11
12     @Override
13     public void setHandler(Handler handler) {
14         this.handler = handler;
15     }
16
17     @Override
18     public void process(File file) {
19
20         if(file.getFileType().equals("text")){
21             System.out.println("Process and saving text file... by "+handlerName);
22         }else if(handler!=null){
23             System.out.println(handlerName+" fowards request to "+handler.getHandlerName());
24             handler.process(file);
25         }else{
26             System.out.println("File not supported");
27         }
28
29     }
30
31     @Override
32     public String getHandlerName() {
33         return handlerName;
34     }
35 }
```

The

```
TextFileHandler
```

interface and overrides its three methods. It holds a reference to the next handler in the chain. In the

```
process
```

method, it checks the file type if the file type is text, it processes it or it forwards it to the next handler.

The other handlers are similar to the above handler.

```
001 package com.javacodegeeks.patterns.chainofresponsibility;
002
003 public class DocFileHandler implements Handler{
004
005     private Handler handler;
006     private String handlerName;
007
008     public DocFileHandler(String handlerName){
009         this.handlerName = handlerName;
010     }
011
012     @Override
013     public void setHandler(Handler handler) {
014         this.handler = handler;
015     }
016
017     @Override
018     public void process(File file) {
019
020         if(file.getFileType().equals("doc")){
021             System.out.println("Process and saving doc file... by "+handlerName);
022         }else if(handler!=null){
023             System.out.println(handlerName+" fowards request to "+handler.getHandlerName());
024             handler.process(file);
025         }else{
026             System.out.println("File not supported");
027         }
028
029     }
030
031     @Override
032     public String getHandlerName() {
033         return handlerName;
034     }
035
036 }
037
038 package com.javacodegeeks.patterns.chainofresponsibility;
039
040 public class AudioFileHandler implements Handler {
041
042     private Handler handler;
043     private String handlerName;
044
045     public AudioFileHandler(String handlerName){
046         this.handlerName = handlerName;
047     }
048
049     @Override
050     public void setHandler(Handler handler) {
051         this.handler = handler;
052     }
053
054     @Override
055     public void process(File file) {
056
057         if(file.getFileType().equals("audio")){
058             System.out.println("Process and saving audio file... by "+handlerName);
059         }else if(handler!=null){
060             System.out.println(handlerName+" fowards request to "+handler.getHandlerName());
061             handler.process(file);
062         }else{
063             System.out.println("File not supported");
064         }
065
066     }
067
068     @Override
069     public String getHandlerName() {
070         return handlerName;
071     }
072
073 }
074
075 package com.javacodegeeks.patterns.chainofresponsibility;
076
077 public class ExcelFileHandler implements Handler{
078
079     private Handler handler;
080     private String handlerName;
081
082     public ExcelFileHandler(String handlerName){
083         this.handlerName = handlerName;
084     }
085
086     @Override
087     public void setHandler(Handler handler) {
088         this.handler = handler;
089     }
090
```

```
            System.out.println("Process and saving excel file... by "+handlerName);
        }else if(handler!=null){
            System.out.println(handlerName+" fowards request to "+handler.getHandlerName());
            handler.process(file);
        }else{
            System.out.println("File not supported");
        }

    }

    @Override
    public String getHandlerName() {
        return handlerName;
    }
}

package com.javacodegeeks.patterns.chainofresponsibility;

public class ImageFileHandler implements Handler {

    private Handler handler;
    private String handlerName;

    public ImageFileHandler(String handlerName){
        this.handlerName = handlerName;
    }

    @Override
    public void setHandler(Handler handler) {
        this.handler = handler;
    }

    @Override
    public void process(File file) {

        if(file.getFileType().equals("image")){
            System.out.println("Process and saving image file... by "+handlerName);
        }else if(handler!=null){
            System.out.println(handlerName+" fowards request to "+handler.getHandlerName());
            handler.process(file);
        }else{
            System.out.println("File not supported");
        }

    }

    @Override
    public String getHandlerName() {
        return handlerName;
    }

}

package com.javacodegeeks.patterns.chainofresponsibility;

public class VideoFileHandler implements Handler {

    private Handler handler;
    private String handlerName;

    public VideoFileHandler(String handlerName){
        this.handlerName = handlerName;
    }

    @Override
    public void setHandler(Handler handler) {
        this.handler = handler;
    }

    @Override
    public void process(File file) {

        if(file.getFileType().equals("video")){
            System.out.println("Process and saving video file... by "+handlerName);
        }else if(handler!=null){
            System.out.println(handlerName+" fowards request to "+handler.getHandlerName());
            handler.process(file);
        }else{
            System.out.println("File not supported");
        }

    }

    @Override
    public String getHandlerName() {
        return handlerName;
    }
}
```

Now, let's test the code above.

```
package com.javacodegeeks.patterns.chainofresponsibility;

public class TestChainofResponsibility {

    public static void main(String[] args) {
        File file = null;
        Handler textHandler = new TextFileHandler("Text Handler");
        Handler docHandler = new DocFileHandler("Doc Handler");
```

```
13
14          textHandler.setHandler(docHandler);
15          docHandler.setHandler(excelHandler);
16          excelHandler.setHandler(audioHandler);
17          audioHandler.setHandler(videoHandler);
18          videoHandler.setHandler(imageHandler);
19
20          file = new File("Abc.mp3", "audio", "C:");
21          textHandler.process(file);
22
23          System.out.println("---------------------------------");
24
25          file = new File("Abc.jpg", "video", "C:");
26          textHandler.process(file);
27
28          System.out.println("---------------------------------");
29
30          file = new File("Abc.doc", "doc", "C:");
31          textHandler.process(file);
32
33          System.out.println("---------------------------------");
34
35          file = new File("Abc.bat", "bat", "C:");
36          textHandler.process(file);
37      }
38
39  }
```

The above program will have the following output.

```
01  Text Handler fowards request to Doc Handler
02  Doc Handler fowards request to Excel Handler
03  Excel Handler fowards request to Audio Handler
04  Process and saving audio file... by Audio Handler
05  ---------------------------------
06  Text Handler fowards request to Doc Handler
07  Doc Handler fowards request to Excel Handler
08  Excel Handler fowards request to Audio Handler
09  Audio Handler fowards request to Video Handler
10  Process and saving video file... by Video Handler
11  ---------------------------------
12  Text Handler fowards request to Doc Handler
13  Process and saving doc file... by Doc Handler
14  ---------------------------------
15  Text Handler fowards request to Doc Handler
16  Doc Handler fowards request to Excel Handler
17  Excel Handler fowards request to Audio Handler
18  Audio Handler fowards request to Video Handler
19  Video Handler fowards request to Image Handler
20  File not supported
```

In the example above, first we created different handlers and chained them. The chain starts from the text handler, which is used to process text files, to the doc handler and so on, till the last handler, the image handler.

Then we created different file objects and passed it to the text handler. If the file can be processed by the text handler it does that, otherwise it forwards the file to the next chained handler. You can see in the output how the requested file was forwarded by the chained objects until it reached the appropriate handler.

Also, please note down, we have not created a handler to process a bat file. So, it passes through all the handlers and results in the output – "File not supported".

The client code is decoupled from the served object. It only sends the request, and the request gets served by any one of the handlers in the chain or does not get processed in case there is support for it.

4. When to use the Chain of Responsibility Pattern

Use Chain of Responsibility when

1. More than one objects may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
2. You want to issue a request to one of several objects without specifying the receiver explicitly.
3. The set of objects that can handle a request should be specified dynamically.

5. Chain of Responsibility in JDK

The following are the usages of the Chain of Responsibility Pattern in Java.

```
java.util.logging.Logger#log()
```

```
javax.servlet.Filter#doFilter()
```

6. Download the Source Code

This was a lesson on the Chain of Responsibility Pattern. You may download the source code here: **ChainofResponsibility-Project**