

Design Patterns

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

 armahmood786@yahoo.com


 alphapeeler.sf.net/pubkeys/pkey.htm

 pk.linkedin.com/in/armahmood

 www.twitter.com/alphapeeler

 www.facebook.com/alphapeeler

 abdulmahmood-sss  alphasecure

 armahmood786@hotmail.com

 <http://alphapeeler.sf.net/me>

 alphasecure@gmail.com

 <http://alphapeeler.sourceforge.net>

 <http://alphapeeler.tumblr.com>

 armahmood786@jabber.org

 alphapeeler@aim.com

 mahmood_cubix  48660186

 alphapeeler@icloud.com

 <http://alphapeeler.sf.net/acms/>

Decorator Pattern

What is Decorator pattern?

Decorator is one of the 23 Design Patterns which were selected by the GoF (Gang of Four).

		Purpose		
		Creation	Structure	Behavior
Scope	Class	Factory Method		Interpreter Template
	Objects	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



Problem!

- We have a simple class for logging message and one implementation.

```
public class DefaultLogger {  
    public virtual void Log(string file, string msg) {  
        //Open File  
        //Write msg to file  
    }  
}
```

- A new requirement states that we need to extend the DefaultLogger to encrypt messages before we log. This can be done simply with inheritance:

```
public class EncryptedLogger extends DefaultLogger {  
    public override void Log(string file, string msg) {  
        //Encrypt Message  
        //Call base functionality  
        super.Log(file, msg);  
    }  
}
```

Problem!

- We get another requirement that when logging, we need to compress the message, then encrypt it, then write to log. This can also be done easily with inheritance:

```
public class CompressLogger extends EncryptedLogger {  
    public override void Log(string file, string msg) {  
        //Compress Message  
        //Call base functionality, which Encrypts and  
        //then writes basic functionality  
        super.Log(file, msg);  
    }  
}
```

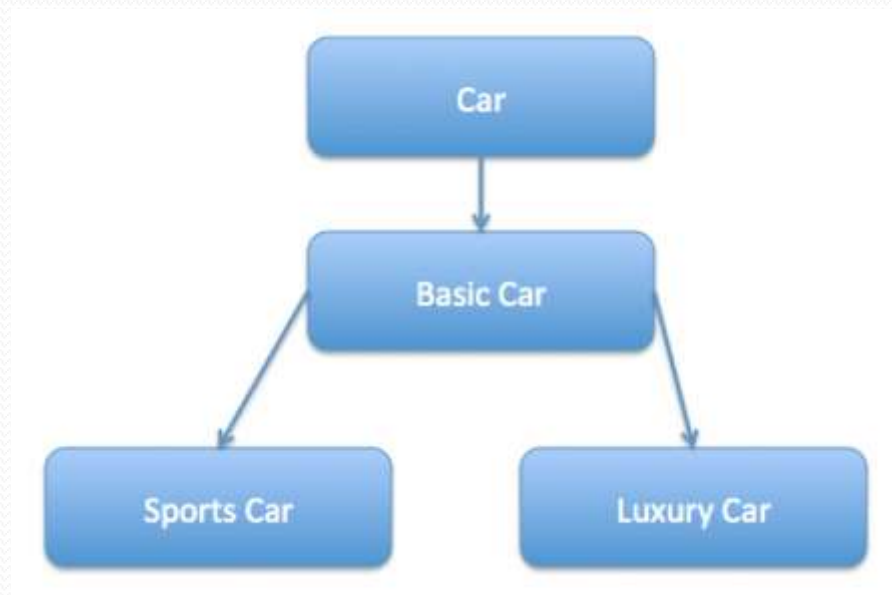
- } Now we get a requirement that we need to Compress the message.. but then simply write it to the log file. No encryption is needed.
- Using inheritance, what class would we need to inherit from? What problem do we have?

Problem! & Solution

- Inheritance statically defined an order of enhancing behavior.
- Issues occur when we need more flexibility in adding behavior.
- Using a Decorator, we can add (and remove) behavior at will!
- We use inheritance to extend the behavior of an object but this is done at compile time and its applicable to all the instances of the class. We can't add any new functionality or remove any existing behavior at runtime – this is when Decorator pattern comes into picture.

Decorator Example 1

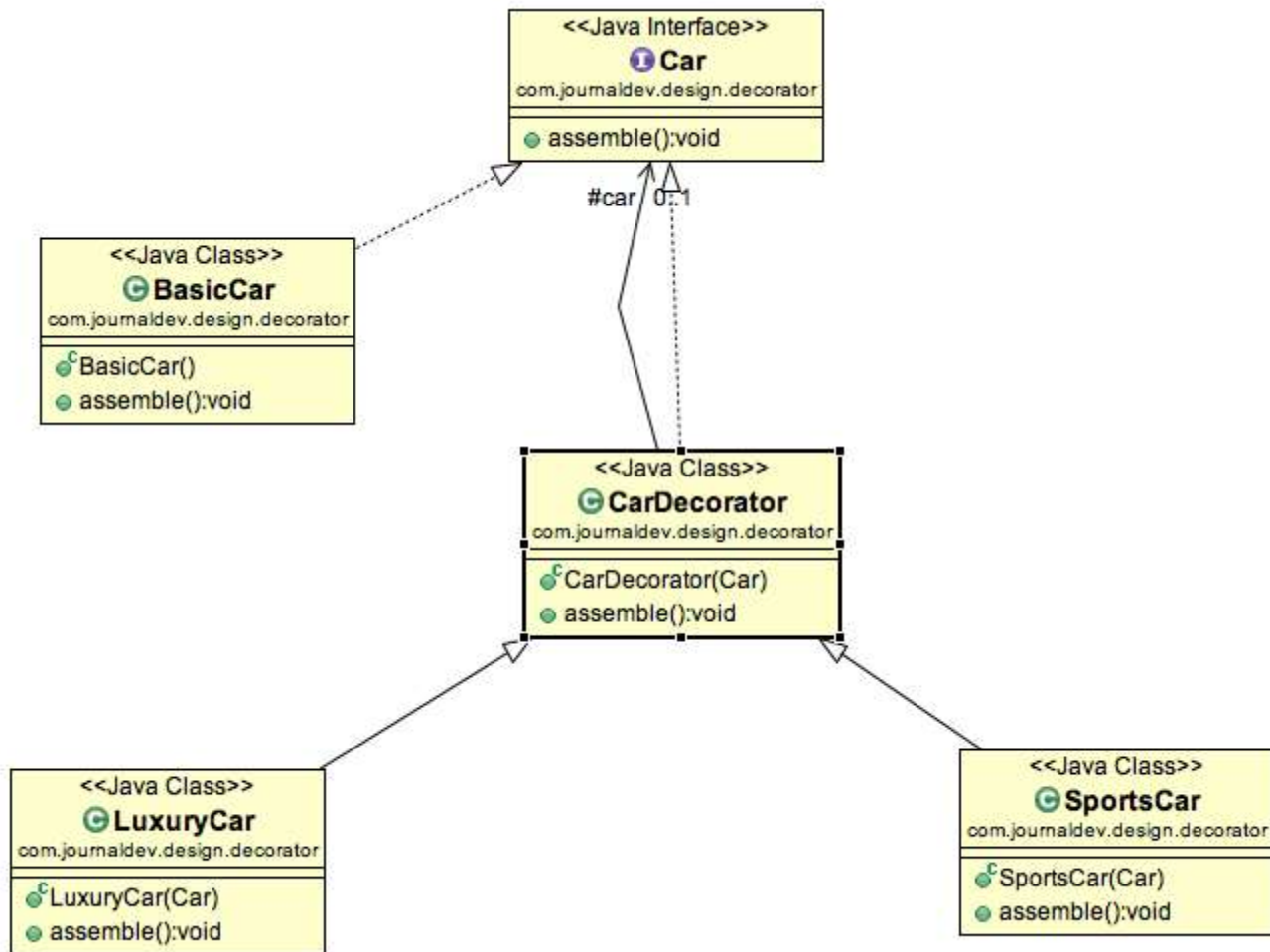
- Suppose we want to implement different kinds of cars
 - we can create interface Car to define the assemble method and then we can have a Basic car, further more we can extend it to Sports car and Luxury Car. The implementation hierarchy will look like below image.



Decorator Example 1

- But if we want to get a car at runtime that has both the features of sports car and luxury car, then the implementation gets complex and if further more we want to specify which features should be added first, it gets even more complex. Now imagine if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage. To solve this kind of programming situation, we apply decorator pattern in java.

Decorator Example 1



Decorator Example 1

- Step1 : create interface Car

```
public interface Car {  
    public void assemble();  
}
```

- Step2: create Implementation of Car (BasicCar)

```
public class BasicCar implements Car{  
    @Override  
    public void assemble() {  
        System.out.print("Basic Car.");  
    }  
}
```

Decorator Example 1

- Step3: Decorator

```
public class CarDecorator implements Car {  
    protected Car car;  
  
    public CarDecorator(Car c){  
        this.car=c;  
    }  
  
    @Override  
    public void assemble() {  
        this.car.assemble();  
    }  
}
```

Decorator Example 1

- Step4: Concrete Decorators

```
public class SportsCar extends CarDecorator {
    public SportsCar(Car c) {
        super(c);
    }
    @Override
    public void assemble(){
        super.assemble();
        System.out.print(" Adding features of Sports Car.");
    }
}

public class LuxuryCar extends CarDecorator {
    public LuxuryCar(Car c) {
        super(c);
    }
    @Override
    public void assemble(){
        super.assemble();
        System.out.print(" Adding features of Luxury Car.");
    }
}
```

Decorator Example 1

- Step5: Demo App

```
public class DecoratorPatternTest {  
    public static void main(String[] args) {  
        Car car = new BasicCar();  
        car.assemble();  
        System.out.println("\n*****");  
        Car sportsCar = new SportsCar(new BasicCar());  
        sportsCar.assemble();  
        System.out.println("\n*****");  
        Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new  
        BasicCar()));  
        sportsLuxuryCar.assemble();  
    }  
}
```

- Out put :

Basic Car.

Basic Car. Adding features of Sports Car.

Basic Car. Adding features of Luxury Car. Adding features of Sports Car.



Things to Note

- Composition, not Inheritance, was used to extend behavior.
- No changes were made to the original class (DefaultLogger).
- We could have provided a GUI so that the end user could specify the behaviors to use.

Comparison to Other Patterns

■ Strategy

- ☐ Good to use when we need to change the “guts” of an object.
- ☐ Think of decorator as a way to change the “skin” of an object.
Also with decorator, we can apply multiple enhancements on top of each other.

■ Adapter

- ☐ Just changes the interface of an object.
- ☐ A decorator will enhance behavior of an object.

The Decorator Pattern from GoF

- Intent
 - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing to extend flexibility
- Also Known As Wrapper
- Examples : add properties to an existing object.
 - Add borders or scrollbars to a GUI component
 - Add headers and footers to an advertisement
 - Add stream functionality such as reading a line of input or compressing a file before sending it over the wire

Applicability

- Use Decorator
 - To add responsibilities to individual objects dynamically without affecting other objects
 - When extending classes is unfeasible
 - Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination (this inheritance approach is on the next few slides)

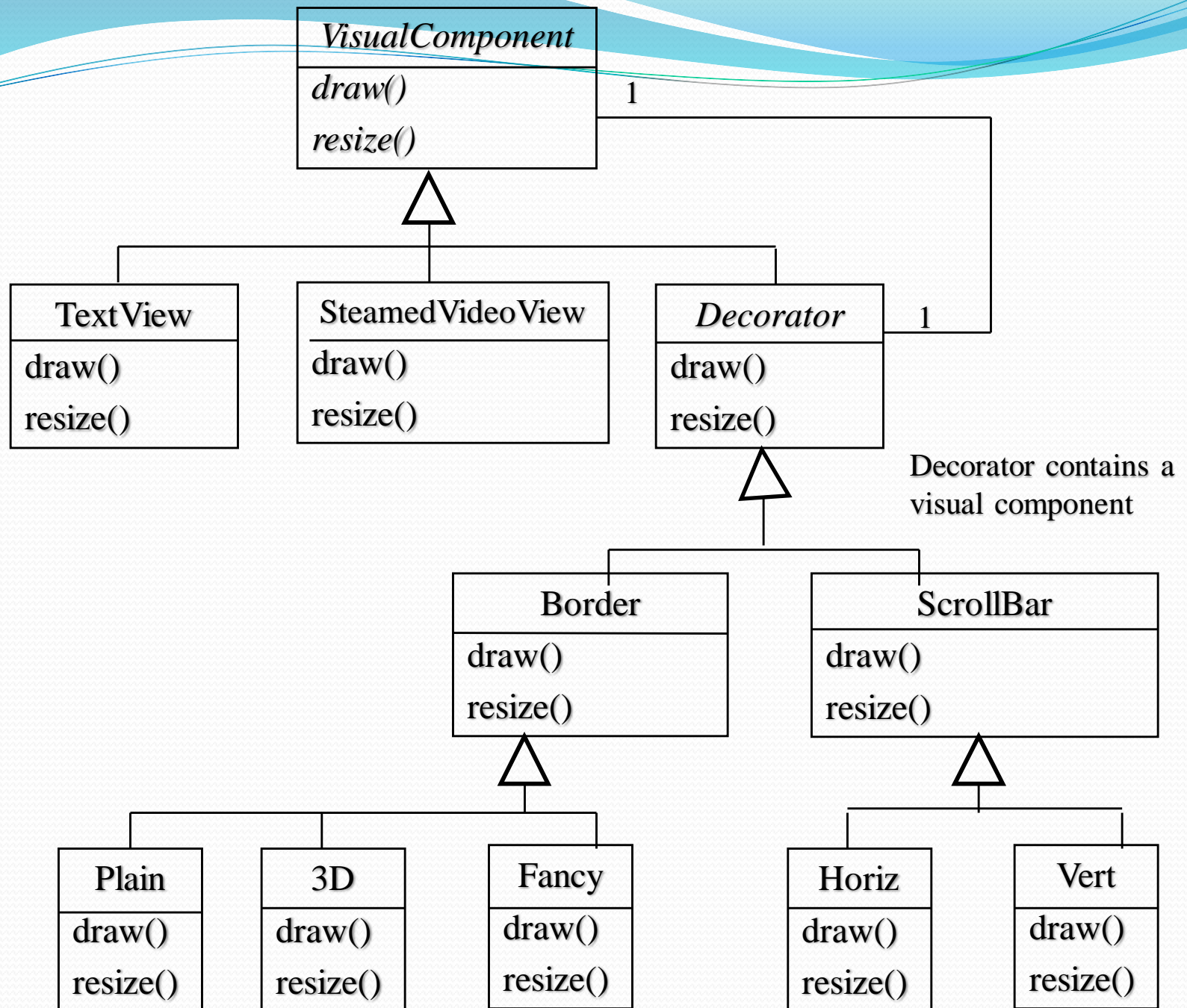
An Application

- Suppose there is a TextView GUI component and you want to add different kinds of borders / scrollbars to it
- You can add 3 types of borders : Plain, 3D, Fancy
- and , 1, or 2 two scrollbars : Horizontal and Vertical
- An inheritance solution requires 15 classes for one view

- 1.TextView_Plain
- 2.TextView_Fancy
- 3.TextView_3D
- 4.TextView_Horizontal
- 5.TextView_Vertical
- 6.TextView_Horizontal_Vertical
- 7.TextView_Plain_Horizontal
- 8.TextView_Plain_Vertical
- 9.TextView_Plain_Horizontal_Vertical
- 10.TextView_3D_Horizontal
- 11.TextView_3D_Vertical
- 12.TextView_3D_Horizontal_Vertical
- 13.TextView_Fancy_Horizontal
- 14.TextView_Fancy_Vertical
- 15.TextView_Fancy_Horizontal_Vertical

Disadvantages

- Inheritance solution has an explosion of classes
- With another type of border added, many more classes would be needed with this design?
- If another view were added such as StreamedVideoView, double the number of Borders/Scrollbar classes (i.e., 30 classes)
- Use the Decorator Pattern instead



Java Borders

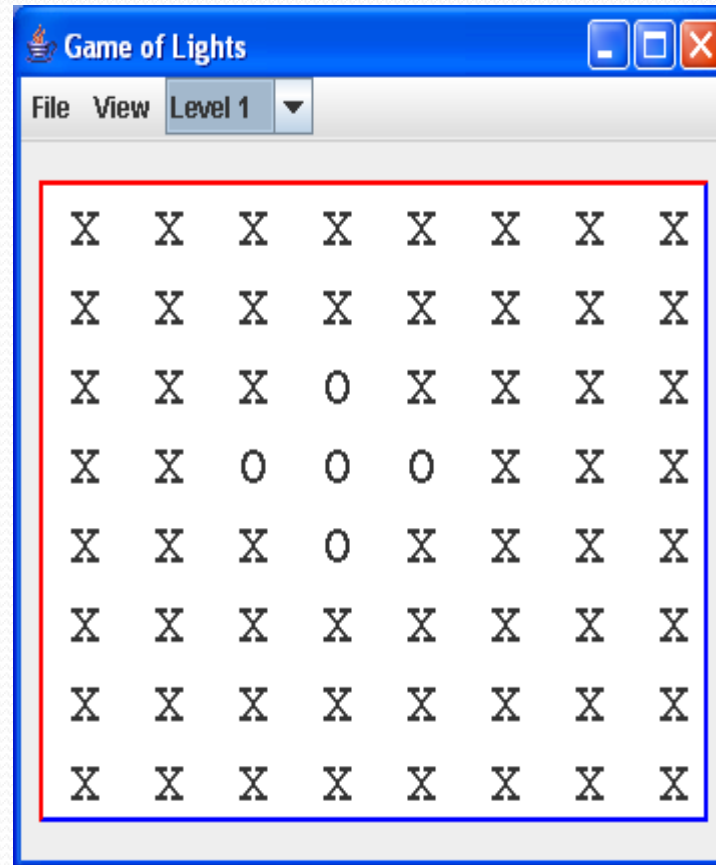
- Any JComponent can have 1 or more borders
- Borders are useful objects that, while not themselves components, know how to draw the edges of Swing components
- Borders are useful not only for drawing lines and fancy edges, but also for providing titles and empty space around components

For more on Borders, The Java Tutorial

<http://java.sun.com/docs/books/tutorial/uiswing/components/border.html>

Java Code: Add a Beveled Border

```
toStringView.setBorder(new BevelBorder(  
    BevelBorder.LOWERCED, Color.BLUE, Color.RED));
```



Motivation Continued

- The more flexible containment approach encloses the component in another object that adds the border
- The enclosing object is called the **decorator**
- The decorator conforms to the interface of the component so its presence is transparent to clients
- The decorator forwards requests to the component and may perform additional actions before or after any forwarding

Decorator Pattern in Java

- `InputStreamReader(InputStream in)` *System.in is an InputStream object*
 - ... bridge from byte streams to character streams: It reads bytes and translates them into characters using the specified character encoding. Java™API
- `BufferedReader`
 - Read text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. Java™API
- What we has to do for console input up to Java 1.4
before Scanner

```
BufferedReader keyboard =  
    new BufferedReader(new  
        InputStreamReader(System.in) );
```

Example of decorator pattern use

BufferedReader *decorates* InputStreamReader

BufferedReader

`readLine()`

InputStreamReader

`read() close()`

Java streams

- With > 60 streams in Java, you can create a wide variety of input and output streams
 - this provides flexibility *good*
 - it also adds complexity *bad*
 - Flexibility made possible with inheritance and classes that accept classes that extend the parameter type
- You can have an `InputStream` instance or any instance of a class that extends `InputStream`

```
public InputStreamReader(InputStream in)
```

One Constructor for many subclasses

- `InputStream` has these direct known subclasses:
`ByteArrayInputStream`, `FileInputStream`,
`FilterInputStream`, `ObjectInputStream`,
`PipedInputStream`, `SequenceInputStream`,
`StringBufferInputStream`
- `System.in` is an instance of `InputStream`

Another Decorator Example

- We also decorated a **FileInputStream** with an **ObjectInputStream** so you can read objects that implement **Serializable**

Another Example as a Code Demo

- Read a plain text file and compress it using the GZIP format ZIP.java
- Read a compress file in the GZIP format and write it to a plain text file UNGZIP.java
- Sample text [iliad10.txt](#) *from Project Gutenberg*

bytes

875,736 iliad10.txt bytes

305,152 iliad10.gz

875,736 TheIliadByHomer

(after code on next slide)

```
// Open the input file
String inFilename = "iliad10.txt";
FileInputStream input = new FileInputStream(inFilename);

// Open the output file
String outFilename = "iliad10.gz";
GZIPOutputStream out = new GZIPOutputStream(
    new FileOutputStream(outFilename));

// Transfer bytes from the output file to the compressed file
byte[] buf = new byte[1024];
int len;
while ((len = input.read(buf)) > 0) {
    out.write(buf, 0, len);
}

// Close the file and stream
input.close();
out.close();
```

```
// Open the gzip file
String inFilename = "iliad10.gz";
GZIPInputStream gzipInputStream =
    new GZIPInputStream(new FileInputStream(inFilename));

// Open the output file
String outFilename = "TheIliadByHomer";
OutputStream out = new FileOutputStream(outFilename);

// Transfer bytes from the compressed file to the output file
byte[] buf = new byte[1024];
int len;
while ((len = gzipInputStream.read(buf)) > 0) {
    out.write(buf, 0, len);
    for (int i = 0; i < len; i++)
        System.out.print((char) buf[i]);
    System.out.println();
}

// Close the file and stream
gzipInputStream.close();
out.close();
```