Description

- The Decorator Pattern is used to extend the functionality of an object dynamically
  - No inheritance
  - No changes in the class source
  - How? Object Composition
- Accomplished creating an object wrapper around the actual object
Characteristics

- The Decorator Object:
  - Has the same interface as the underlying object
    - Clients interact in the same way with the new object
  - Contains a reference to the actual object
  - Receive and forward calls from a client to the underlying object
  - Adds additional functionality before or after forwarding requests

Decorator Pattern vs. Inheritance

<table>
<thead>
<tr>
<th>Decorator Pattern</th>
<th>Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used to extend the functionality of a particular object.</td>
<td>Used to extend the functionality of a class of objects.</td>
</tr>
<tr>
<td>Does not require subclassing.</td>
<td>Requires subclassing.</td>
</tr>
<tr>
<td>Dynamic.</td>
<td>Static.</td>
</tr>
<tr>
<td>Runtime assignment of responsibilities.</td>
<td>Compile time assignment of responsibilities.</td>
</tr>
<tr>
<td>Prevents the proliferation of subclasses leading to less complexity and confusion.</td>
<td>Could lead to numerous subclasses, exploding class hierarchy on specific occasions.</td>
</tr>
<tr>
<td>More flexible.</td>
<td>Less flexible.</td>
</tr>
<tr>
<td>Possible to have different decorator objects for a given object simultaneously. A client can choose what capabilities it wants by sending messages to an appropriate decorator.</td>
<td>Having subclasses for all possible combinations of additional capabilities, which clients expect out of a given class, could lead to a proliferation of subclasses.</td>
</tr>
<tr>
<td>Easy to add any combination of capabilities. The same capability can even be added twice.</td>
<td>Difficult.</td>
</tr>
</tbody>
</table>
EXAMPLE: Message Logging Utility

Initially:

```
<<interface>>
Logger
log(msg: String)
```

```
ConsoleLogger
log(msg: String)
```

```
FileLogger
log(msg: String)
```

NOTE: LoggerFactory not shown in this figure

EXAMPLE (Inheritance):

Suppose new functionalities:
- Transform an incoming message to an HTML document
- Apply simple encryption on an incoming message

```
<<interface>>
Logger
log(msg: String)
```

```
ConsoleLogger
log(msg: String)
```

```
FileLogger
log(msg: String)
```

```
HTMLConsoleLogger
log(msg: String)
```

```
HTMLFileLogger
log(msg: String)
```

Class Explosion!!!
EXAMPLE (Decorator Pattern):

- Define a Default root decorator “LoggerDecorator”:
  - Contains a reference to a Logger instance that points to a Logger object it wraps.
  - Implements the Logger interface and provides the basic default implementation for the log method, forwarding calls to the Logger object it wraps.

- Define two subclasses: HTMLLogger and EncryptLogger

```
public class LoggerDecorator implements Logger {
    private Logger logger;
    public LoggerDecorator(Logger imp_logger) {
        logger = imp_logger;
    }
    public void log(String dataLine) {
        /*
         * Default implementation to be overridden by subclasses.
         */
        logger.log(dataLine);
    }
}
```

EXAMPLE:

- LoggerDecorator Class
EXAMPLE:

- Concrete Logger Decorators:
  - Both HTMLLogger and EncryptLogger:
    - Overrides Default Log method
    - Implement new functionalities

- Client needs to:
  - Create an appropriate Logger instance (FileLogger or ConsoleLogger using factory method)
  - Create an appropriate LoggerDecorator instance by passing the Logger instance created in Step 1 to its constructor
  - Invoke methods on the LoggerDecorator instance (Step 2) as it would on the Logger instance (Step 1)

EXAMPLE (Client Code):

```java
class DecoratorClient {
    public static void main(String[] args) {
        LoggerFactory factory = new LoggerFactory();
        Logger logger = factory.getLogger();
        HTMLLogger hLogger = new HTMLLogger(logger);
        //the decorator object provides the same interface.
        hLogger.log("A Message to Log");
        EncryptLogger eLogger = new EncryptLogger(logger);
        eLogger.log("A Message to Log");
    }
}
```
EXAMPLE: HTML Decorator Class

```java
public class LoggerDecorator implements Logger {
    Logger logger;
    public LoggerDecorator(Logger imp_logger) {
        logger = imp_logger;
    }
    public void log(String Dataline) {
        // Default implementation to be overriden by subclasses.
        logger.log(Dataline);
    }
}
```

```java
public class HTMLLogger extends LoggerDecorator {
    public HTMLLogger(Logger imp_logger) {
        super(imp_logger);
    }
    public void log(String Dataline) {
        // Added functionality
        String Dateline = makeHTML(Dataline);
        logger.log(Dataline);
    }
    public String makeHTML(String Dataline) {
        // Make it into an HTML document.
        String Dateline = "<html><body>" + Dataline + "</body>";
        return Dateline;
    }
}
```

EXAMPLE:

- What if we add a new Message Logger (DB Logger)?
  - A client only needs to do the steps explained on the last slide
  - DB Logger would be of Logger type and it can be sent to any of the HTMLLogger or EncryptLogger as an argument while invoking their constructors

- And adding new Decorators?
  - Decorators inside Decorators!
EXAMPLE:

- Association between Different Logger Classes and Logger Decorator