HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN
WÜRZBURG-SCHWEINFURT

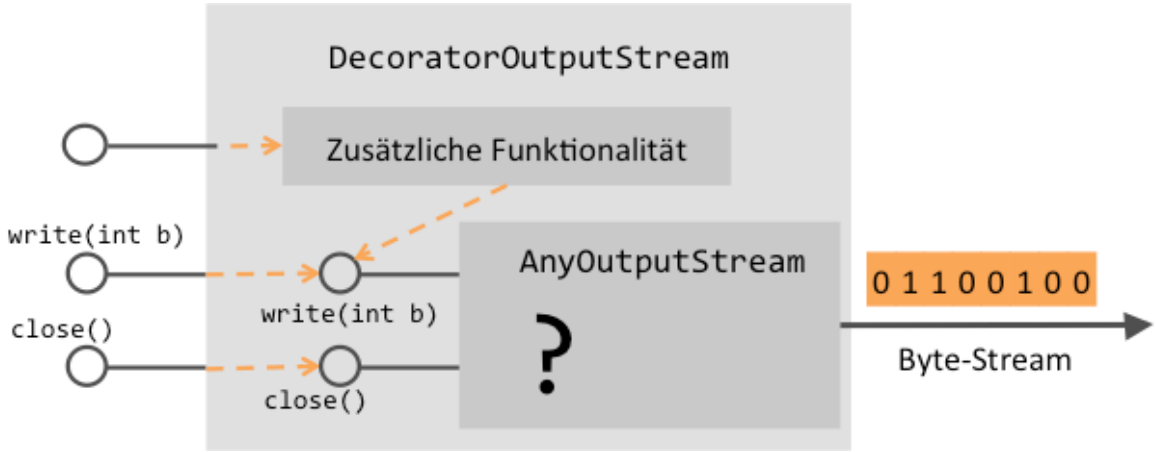
FH·W·S


Programmieren II
Sommersemester 2016

**Übungsblatt 8 (2 Punkte)**

## Decorator

Das Prinzip eines Decorators ist es, eine existierende Klasse zu erweitern, indem die bestehende Klasse von einer neuen Klasse (dem Decorator) mit Hilfe von Komposition genutzt wird, um zusätzliche Funktionalität anzubieten. Die Methoden der bestehenden Klasse werden dabei oft vom umhüllenden Objekt „durchgeschleift“ und nur punktuell ergänzt.




Dieses Prinzip wird innerhalb der Java-Klassenbibliothek insbesondere bei den Streams intensiv eingesetzt. Auf diese Weise können unterschiedlichste Streamklassen (z.B. für das Lesen und Schreiben von Dateien, Kommunikation über das Netzwerk etc.) von den Ergänzungen eines Decorators profitieren.

## Aufgabe „exercise08“ (Abgabe vom 24.5. - 30.05.2016, bZv-relevant)

Im Zuge der weitreichenden Überwachung durch Geheimdienste möchten Sie Ihre Streams in Zukunft grundsätzlich verschlüsseln. Da die Verschlüsselung sowohl z.B. beim Speichern und Laden von Daten in einem Filesystem als auch beim Senden und Empfangen von Daten über das Netzwerk zum Einsatz kommen soll, möchten Sie

- einen Decorator zum Verschlüsseln von OutputStreams mit dem Namen `EncryptOutputStream` und
- einen Decorator zum Entschlüsseln von InputStreams mit dem Namen `DecryptInputStream`

entwickeln.

Während dem `EncryptOutputStream` beim Konstruktoraufruf ein OutputStream-Objekt und ein Integer-Wert als Schlüssel mitgegeben werden, erwartet der Konstruktor des `DecryptInputStream` einen InputStream und ebenfalls einen Integer-Wert als Schlüssel:

public `EncryptOutputStream`(OutputStream os, int key)
public `DecryptInputStream`(InputStream is, int key)

Um einen verschlüsselten Stream wieder lesen zu können, müssen der generierende `EncryptOutputStream` und der lesende `DecryptInputStream` mit dem gleichen (geheimen) Schlüssel (key) erzeugt werden.

Die Verschlüsselung selbst ist recht einfach (und damit auch sehr unsicher): Jedes Bit eines zu verschlüsselnden Bytes wird unverändert gelassen, wenn der Schlüssel an dieser Bitstelle eine 0 ausweist. Sollte beim Schlüssel die Bitstelle mit einer 1 markiert sein, wird das entsprechende Bit des zu verschlüsselnden Bytes invertiert. Dieses Verhalten entspricht der bitweisen XOR-Funktion und ist in Java mit Hilfe des ^-Operators realisierbar:

```
int encrypted = unencrypted ^ key;
```

**Ihre Aufgabe:**

a) Entwickeln Sie im Package `exercise08` eine Klasse `EncryptOutputStream` mit dem oben aufgeführten Konstruktor. Die Klasse soll Unterklasse der abstrakten Klasse `OutputStream` sein und als Decorator für den beim Konstruktoraufruf übergebenen `OutputStream` dienen. Auszugebende Bytes werden mit Hilfe des ebenfalls dem Konstruktor übergebenen Schlüssels gemäß der oben erläuterten Vorgehensweise verschlüsselt. Es sind jedoch nur Schlüsselwerte zwischen 0 und 255 zulässig; andere Werte sollen zu einer `IllegalArgumentException` im Konstruktor führen. Beachten Sie, dass per Konvention ein Schließen des `EncryptOutputStream` auch zum Schließen des eingebetteten Streams führen muss.

b) Entwickeln Sie analog zur Teilaufgabe a) eine Klasse `DecryptInputStream`, mit deren Hilfe die verschlüsselten Streams wieder entschlüsselt werden können. Leiten Sie sich die notwendige Entschlüsselungsvorgehensweise aus den Vorgaben selbst her.

c) Die entwickelten Stream-Decorator sollen für den Austausch verschlüsselter Textdateien verwendet werden. Der Schlüsselwert sei 100. Vollenden Sie dazu die im Package `exercise08` bereits begonnene Klasse `FileDecryptor`, indem Sie die statische Methode `decryptFile` implementieren. Dabei soll die Datei mit dem als Parameter übergebenen Dateinamen geöffnet und entschlüsselt werden. Bitte geben Sie die entschlüsselten Bytes in der Methode `decryptFile` als Zeichen (char) auf der Konsole aus:

```
System.out.print  ((char)decrypted);
```

Überprüfen Sie die Korrektheit Ihrer Lösung mit Hilfe der main-Methode und der mitgelieferten verschlüsselten Datei `data.crypt`.