Vorbemerkung zum 6.Aufgabenzettel und allen weiteren Aufgabenzetteln

- Und wieder der Hinweis:
  Mögliche gegebene Tests dürfen nicht verändert, wohl aber am Ende ergänzt werden. Auch dürfen Sie die Testmethoden überladen und so um weiter eigene Parameter ergänzt werden.
  Sofern Sie die Testmethoden ergänzen, trennen Sie Ihren bzw. den ergänzten Teil deutlich ab. Z.B. bei der Ausgabe auf dem Bildschirm durch 2 Leerzeilen, einen großen Querstrich und wieder 2 Leerzeilen.

Aufgabe A6.1 CardProcessor – Finde den ersten Drilling

Implementieren Sie eine Klasse **`CardProcessor`**, die sich vom gegebenen Interface **`CardProcessor_I`** ableitet und

- die (Spiel-)Karten verarbeitet.
  Die Idee ist: Es sollen "einkommende" Karten untersucht und zwischengespeichert werden. Sobald der erste Drilling vorliegt, soll dieser zurückgegeben werden – andernfalls **`null`**.
- einen parameterlosen Konstruktor unterstützt
- (u.a.) die folgenden Methoden aufweist:

  `Object process( Card )`
  verarbeitet eine (Spiel-)Karte. Die als Parameter übergebene Karte wird zunächst "intern" gespeichert (bzw. den "bisher übergebenen" Karten hinzugefügt). Sobald der erste Drilling (3 Karten vom gleichen Rang) vorliegt, soll dieser Drilling (also die entsprechenden 3 Karten) als Rückgabewert der Methode zurückgegeben werden – andernfalls ist **`null`** zurückzugeben.

  `void reset()`
  löscht alle (intern) gespeicherten Karten. Ein möglicherweise zuvor vorhandener Drilling wird ebenfalls gelöscht. Nach **`reset()`** befindet sich der **`CardProcessor`** im Ausgangszustand.

Aufgabe A6.2 Wörter im Text zählen

Idee/Aufgabe: Analysieren Sie wie oft ein/jedes Wort in einer Textdatei auftritt.

Im PUB liegt der Java Code für eine **`WordGrabber`**-Klasse, die es Ihnen ermöglicht eine Datei Wort für Wort zu durchlaufen. Jedes Wort wird von der **`WordGrabber`**-Klasse als String "geliefert". Die **`WordGrabber`**-Klasse implementiert **`Iterator<String>`** und **`Iterable<String>`**.

Beispiel für die Nutzung des WordGrabbers:

```
String fileNameWithPath = ...
WordGrabber wg = new WordGrabber( fileNameWithPath );
if ( wg.hasNext() ) {                          // falls noch ein ungelesenes Wort in der Datei ist
    System.out.printf( "[ %s ]", wg.next() );  // das entsprechende Wort ausgeben
}//if
```

Alles was **`wg.next()`** als Wort abliefert, zählt als Wort. Merken Sie sich jedes dieser Wörter und zählen Sie, wie oft diese in der Datei auftreten. Groß-/Klein-Schreibung der Wörter soll hierbei ignoriert werden.

Im Rahmen dieser Aufgabe müssen Sie die Klassen: **`WordCounter`**, **`Word`** und **`Counter`** implementieren.

Um die Inhalte der Vorlesung einzuüben, sollen die Wörter von Ihnen nicht als **`String`**, sondern als Objekte der Klasse **`Word`** verarbeitet werden. Die Klasse **`Word`** ist von Ihnen zu implementieren und Objekte dieser Klasse sollen das jeweilige Wort als internen String aufnehmen. Innerhalb der Klasse **`Word`** können Sie die Klasse **`String`** ohne Einschränkungen nutzen.

Ok, eigentlich ist die Klasse Word unsinnig, aber wir wollen den Umgang mit einer Map an einem einfachen Beispiel üben.

Die Klasse **`Word`** ist zwingend eingefordert.

Implementieren Sie weiterhin eine Klasse **`Counter`**. Die Klasse soll als Elemente den Zählerstand (wie oft kam das jeweilige Wort vor) und eine Inkrementier-Methode **`void inc()`** aufweisen.

Auch um Test von Applikation zu trennen, schreiben Sie eine Klasse **`WordCounter`**, die eine **`printStatistic()`**-Methode aufweisen soll. Die **`printStatistic()`**-Methode soll als **`String`** eine Pfad-Angabe zu der zu untersuchenden Text-Datei entgegennehmen, die darin enthaltenen Wörter zählen und dann deren Anzahl auf dem Bildschirm ausgeben. Verwenden Sie in diesem Zusammenhang als Datenstruktur eine **`Map`** und für Ihr konkretes **`Map`**-Objekt eine **`HashMap<Word,Counter>`** (dies ist Pflicht - auch wenn die meisten vermutlich nach dem Lesen dieser Aufgabe eine TreeMap vorziehen würden.

Frage: Warum überhaupt eine Map? Und was unterscheidet eine TreeMap von einer HashMap?

Nachdem Sie alle Wörter gezählt haben, geben Sie diese lexikografisch sortiert nach den Wörtern und mit der jeweils zugehörigen Anzahl ihres Auftretens auf dem Bildschirm aus.

Beispiel-Ausgabe Ihrer **`printStatistic()`**-Methode (für Beispieltext "A Princess of Mars" mit "Trara" also Gutenberg Lizenz usw.):

```
...
[ zitidar ] : 2
[ zitidars ] : 6
[ zodanga ] : 54
[ zodangan ] : 21
[ zodangans ] : 14
```

Kommentar-&Leerzeilen-bereinigt lässt sich diese Aufgabe mit wenigen (etwa 40) Zeilen gut lesbaren Code lösen.

Größere Texte zum Testen können Sie unter www.gutenberg.org finden. Z.B. “A Princess of Mars” von Edgar Rice Burroughs in us-ascii.

Kurz-Dokumentation zur **`WordGrabber`** -Klasse:

Der Konstruktor ohne Parameter öffnet eine Datei "input.txt" in Ihrem Projekt-Verzeichnis.
Als Parameter können Sie aber auch eine Datei vorgeben. Ohne Pfadangabe muss diese Datei im Projekt-Verzeichnis liegen.

Beispiel:

```
WordGrabber wg = new WordGrabber( "c:\\test.txt" );
```

öffnet die Datei "c:/test.txt".
Sie können auch **`WordGrabber( "c:/test.txt" )`** schreiben.

`wg.hasNext()`
liefert **`true`** falls noch ein ungelesenes Wort verfügbar ist, sonst **`false`**.
(Die **`WordGrabber`**-Klasse implementiert **`Iterator<String>`**)

`wg.next()`
liefert das nächste ungelesene Wort (vom Typ String).
(Die **`WordGrabber`**-Klasse implementiert **`Iterator<String>`**)

Aufgabe A6.3 Multi-Purpose List

Vermutlich ist der gestellte **`TestFrame`** nicht direkt compilierbar und es müssen die **`import`**-Zeilen angepasst werden.

Implementieren Sie eine doppelt verkettete Liste für einen universellen Einsatz.

In einer Klasse **`MyList`** soll eine universelle verkettete Liste von Ihnen selbst implementiert werden. Sie dürfen hierfür also nicht auf die "Collections" zurückgreifen (und auch nicht auf "Arrays" ;-). Verwenden Sie Generics in sinnvoller Weise, so dass Sie nachher beliebige Datentypen mit der Liste verwalten können. Analog zu den Positionsnummern in einem Array soll das erste Element in der Liste die Positionsnummer 0, das zweite die Positionsnummer 1 usw. aufweisen. Sie müssen Doppelte nicht unterstützen bzw. können voraussetzen, dass keine Doppelten auftreten.

Die folgenden Methoden sollen unterstützt werden:

Informations-Objekt **`getNo( int )`**
Das Informations-Objekt mit der entsprechenden Positionsnr. in der Liste abliefern.

Informations-Objekt **`getF()`**
Das erste Informations-Objekt der Liste abliefern.

Informations-Objekt **`getL()`**
Das letzte Informations-Objekt der Liste abliefern.

Informations-Objekt **`extractNo( int )`**
Das Informations-Objekt mit der entsprechenden Positionsnr. in der Liste abliefern und aus der Liste mitsamt dem zugehörigen Knoten entfernen.

Informations-Objekt **`extractF()`**
Das erste Informations-Objekt der Liste abliefern und aus der Liste mitsamt dem zugehörigen Knoten entfernen.

Informations-Objekt **`extractL()`**
Das letzte Informations-Objekt der Liste abliefern und aus der Liste mitsamt dem zugehörigen Knoten entfernen.

`boolean putNo( int , ` Informations-Objekt **`)`**
Bei gültiger Positonsangabe, das Informations-Objekt an der übergegeben Position in der Liste einfügen und alle (alten) Informations-Objekte ab der übergebenen Position entsprechend nach "hinten" verschieben und **`true`** zurückgeben, sonst **`false`** zurückgeben (und nichts einfügen).
Wenn die Liste n Informations-Objekte aufweist, dann sind 0, ..., n gültige Positionsangaben.

`void putF(` Informations-Objekt **`)`**
Das Informations-Objekt am Anfang der Liste einfügen.

`void putL(` Informations-Objekt **`)`**
Das Informations-Objekt am Ende der Liste einfügen.

Informations-Objekt **`setNo( int , `** Informations-Objekt **`)`**
Bei gültiger Positonsangabe, das übergebene (neue) Informations-Objekt an die übergegebene Position in der Liste schreiben und das alte dort befindliche Informations-Objekt zurückgeben.
Wenn die Liste n Informations-Objekte aufweist, dann sind 0, ..., n-1 gültige Positionsangaben.

`void removeNo( int )`
Sowohl den Knoten als auch das Informations-Objekt an der übergebenen Position aus der Liste löschen.
Wenn die Liste n Informations-Objekte aufweist, dann sind 0, ..., n-1 gültige Positionsangaben.

`boolean remove(` Informations-Objekt **`)`**
Das Informations-Objekt mit zugehörigem Konten aus der Liste löschen - *sollte es mehrfach vorkommen, dann das 1.Auftreten löschen.* Rückgabewert ist **`true`** bei Erfolg, sonst **`false`** (z.B. falls nicht in der Liste enthalten).

`void clear()`
Die Liste zurücksetzen.

`boolean isEmpty()`

Liefert eine Rückmeldung, ob die Liste leer ist.

`int getSize()`
Liefert die Anzahl der in der Liste enthaltenen Informations-Objekte bzw. Elemente.

`boolean contains(` Informations-Objekt **`)`**
Liefert eine Rückmeldung, ob ein Informations-Objekte in der Liste enthalten ist.

Testen Sie nun Ihre Liste mit einer "Disc-Verwaltung". Eine **`Disc`** kann eine **`CD`** oder **`DVD`** sein.
Eine **`Disc`** hat einen Titel und enthält **`AUDIO`** (Musik), **`MOVIE`** (Film) oder **`VIDEO`**.
Für eine **`CD`** ist auch der Interpret von Interesse.
Bei einer **`DVD`** ist das Format von Interesse (**`PAL`**, **`NTSC`**).

Es müssen die folgenden Konstruktor-Aufruf-Beispiele unterstützt werden:

```
new CD( "Titel", AUDIO, "Interpret" )
new DVD( "Titel", MOVIE, PAL )
new DVD( "Titel", MOVIE, NTSC )
new DVD( "Titel", VIDEO, PAL )
new DVD( "Titel", VIDEO, NTSC )
```

Nicht eingefordert, aber sicherlich sinnvoll:

Ein **`package list`** für die Klasse **`MyList`** und eine nötige Knoten-Klasse.
Ein **`package media`** für die Referenztypen **`Disc`**, **`CD`**, **`DVD`** und möglichen weiteren Hilfs-Referenztypen.

Zur Reduzierung der Implementierungsaufwände könnten u.U. die folgenden internen Hilfsmethoden sinnvoll sein - bzw. mögliche Varianten davon (je nachdem wie die Freiheitsgrade der Aufgabenstellung ausgelegt wurden):

Knoten **`iSearchNode(`** Informations-Objekt **`)`**
Interne Hilfsmethode, die eine Referenz auf den jeweils zugehörigen Knoten für ein Informations-Objekt abliefert. Diese Methoden sollte nicht für Anwender der Klasse zur Verfügung stehen und daher nicht **`public`** sein.

Knoten **`iGetNodeNo( int )`**
Interne Hilfsmethode, die eine Referenz auf den jeweiligen Knoten an der entsprechenden Position in der Liste abliefert. Diese Methoden sollte nicht für Anwender der Klasse zur Verfügung stehen und daher nicht **`public`** sein.

`boolean iRemoveNode(` Knoten **`)`**
Interne Hilfsmethode, die eine Referenz auf den jeweiligen Knoten in der Liste entfernt und eine Erfolgsmeldung abliefert. Diese Methoden sollte nicht für Anwender der Klasse zur Verfügung stehen und daher nicht **`public`** sein.

Für die Fehlersuche sind u.U. die folgenden Methoden nützlich:

`void printElemF2L()`
Die in der Liste enthaltenen Informations-Objekte der Reihe nach vom Ersten zum Letzten ausgeben. Diese Methoden macht auch für Anwender der Klasse Sinn und kann daher **`public`** sein.

`void printElemL2F()`
Die in der Liste enthaltenen Informations-Objekte der Reihe nach vom Letzten zum Ersten ausgeben. Diese Methoden macht auch für Anwender der Klasse Sinn und kann daher **`public`** sein.

`void printNodeF2L()`
Die in der Liste enthaltenen Knoten der Reihe nach vom Ersten zum Letzten ausgeben. Diese Methoden sollte nicht für Anwender der Klasse zur Verfügung stehen und daher nicht **`public`** sein.

`void printNodeL2F()`
Die in der Liste enthaltenen Knoten der Reihe nach vom Letzten zum Ersten ausgeben. Diese Methoden sollte nicht für Anwender der Klasse zur Verfügung stehen und daher nicht **`public`** sein.

Entwickeln Sie auch eigene Tests. Die gestellten Tests sind primär Abnahmetests und haben nicht die Intention bei der Fehlersuche zu unterstützen.