# *Aufgabe A6.1 Thingy Collector – (immer) 5 unterschiedliche Dinge sammeln*

Diese Aufgabe umfasst 2 Teile:
- Umgang mit dem Tool: **JUnit**5 – Ein erstes Mal Erstellen eigener Unit-Tests
- Die eigentliche Aufgabe – **5 Ungleiche** zusammenstellen

**Die "eigentliche Aufgabe":**

Vorbemerkung:
Vorweg: Wir erinnern uns an V4.2 😉
Vorweg: Wieder gilt, dass es "nur" funktioniert reicht nicht. Sie sollen diejenigen Collection(s) finden und verwenden, die Sie optimal unterstützt/unterstützen. Sie müssen die Auswahl begründen können.

Implementieren Sie eine Klasse **`Collector`**, die sich von **`Collector_I`** ableitet und die **`Item`**s verarbeitet. Die Idee ist: Items gemäß den jeweiligen Anforderungen zu sammeln. Die Items haben Eigenschaften. Diese Eigenschaften sind:

- Farbe — Sie haben Zugriff auf dieses Eigenschaft mit: `Color getColor()`
- Größe — Sie haben Zugriff auf dieses Eigenschaft mit: `Size getSize()`
- Gewicht — Sie haben Zugriff auf dieses Eigenschaft mit: `Weight getWeight()`
- Wert — Sie haben Zugriff auf dieses Eigenschaft mit: `Long getValue()`

Alles weitere müssen **Sie selbst** den gestellten Referenztypen `Color`, `Item`, `Size`, `Weight` entnehmen.

Ein **`Collector`** soll einkommende Items sammeln. Immer sobald 5 verschiedene Items vorliegen, sollen diese "abgeliefert" werden. Falls mehrere Items zur Auswahl stehen, soll das jeweils älteste gewählt werden. Innerhalb der als Ergebnis abgelieferten Collection gibt es keine Anforderung bzgl. der Ordnung. Die abgelieferten Items sind nach der Herausgabe nicht mehr im Bestand bzw. Gedächtnis des Collectors. Ein Item kann also nur einer Sammlung (Zusammenstellung von 5 Verschiedenen) angehören.

Die Klasse **`Collector`** soll das Interface **`Collector_I`** unterstützen und die folgenden Elemente aufweisen:

**`Collector()`**
erzeugt einen Collector.

**`Collection<Item> process( Item )`**
verarbeitet ein Item. Das als Parameter übergebene Item wird den "bisher übergebenen" Items hinzugefügt.
Immer wenn fünf unterschiedliche(!)[1] Items vorliegen, sollen diese als Rückgabewert der Methode (in Form einer geeigneten Collection) abgeliefert werden – andernfalls soll **`null`** zurückgegeben werden.
Items, die bereits Teil einer (Ergebnis-)Collection waren, dürfen nicht für die Bildung einer weiteren Collection verwendet werden - solche Items sind also - *nachdem sie als Teil eines Ergebnisses herausgegeben wurden* - aus dem "Gedächtnis" zu entfernen[2]. Ein Item darf nur jeweils maximal einer (Ergebnis-) Collection angehören[3]. (*Mögliche interne Collections sind hiervon ausgeschlossen.*)
Bemerkung: Es wird zugesichert, dass (bei den gestellten Tests) nur "sinnvolle" Parameter übergeben werden.

---

[1] Mit "Unterschiedliche" sind "Ungleiche" gemeint. Doppelte werden als doppelt gewertet, weil sie gleich sind bzw. nicht voneinander unterschieden werden können. Eine andere Formulierung ist also: Immer wenn 5 Items zusammengestellt werden können, ohne dass Doppelte dabei sind.
[2] Die Aussagen bezüglich des Entfernens aus dem Gedächtnis betreffen die Identität. Gleiche bzw. Doppelte, die noch nicht für eine Ergebnis-Collection verwendet wurden, dürfen (bzw. müssen bei Bedarf) noch verwendet werden.
[3] Außer, es ist neu dem Bestand hinzugefügt wurden. Ein Item, dass aus dem Bestand entfernt wurde, kann danach erneut dem Bestand hinzugefügt werden. Solange eine Item korrekter Weise im Bestand ist, wird zugsichert, dass es nicht erneut angeboten wird.

`void reset()`
löscht das "Gedächtnis". Ein möglicher (interner) Zustand wird auf den Ausgangswert bzw. die Starteinstellung zurückgesetzt.

**Ergänzung des UnitTestFrames um eigene Tests:**

Auch um den Umgang mit JUnit zu üben, sollen Sie ein erstes Mal eigene Unit-Tests erstellen und konkret den gestellten UnitTestFrame um min. 2 weitere Tests ergänzen.
Sie nutzen schon seit Längerem gestellte Unit-Tests. Diese enthalten viele Code-Beispiele an denen Sie sich orientieren können.

Die JUnit5-Tests sollen in Methoden implementiert werden, die dem folgenden Template genügen:

```
@Test
public void guterSelbsterklärenderNameFürDenTest(){
    …
}
```

Der UnitTestFrame startet dann die einzelnen JUnit-Test-Methoden an.

Eine JUnit-Test-Methode repräsentiert eine "Testeinheit". Sie gilt als erfolgreich abgeschlossen, wenn Sie bis zum Ende durchlaufen wird bzw. die Methode "regulär" verlassen wird.
Vergleichbar dem assert-Statement gibt es assert-Methoden, die etwas zusichern indem sie eine Erwartung überprüfen. Sollte die Zusicherung nicht gelten, wird die Ausführung der Methode beendet und der Test wird als fehlgeschlagen gewertet.
Es gibt unterschiedliche assert-Methoden. Nachfolgend eine Auflistung der wichtigsten assert-Methoden:

**assertEquals** überprüft zwei Variablen-Werte (Objekte oder primitive Datentypen) auf Gleichheit.
**assertSame** überprüft, ob zwei Referenz-Variablen dasselbe Objekt referenzieren (oder beide null) sind.
**assertNotSame** überprüft, ob zwei Referenz-Variablen unterschiedliche Objekte referenzieren.

**assertNull** überprüft, ob eine Referenz-Variable null referenziert.
**assertNotNull** überprüft, ob eine Referenz-Variable nicht null referenziert.

**assertTrue** prüft, ob ein boolscher Ausdruck wahr ist.
**assertFalse** prüft, ob ein boolscher Ausdruck unwahr ist.

Sofern von den assert-Methoden 2 Werte (i.d.R. Objekte) als Parameter entgegengenommen werden, um diese zu vergleichen, wird zuerst der korrekte bzw. der erwartete Wert und danach der tatsächlich berechnete Wert übergeben.

Die API für JUnit5 findet sich unter:
https://junit.org/junit5/javadoc/latest/index.html
bzw. konkret für die Assert-Methoden
https://junit.org/junit5/javadoc/latest/org/junit/Assert.html

Grundsätzlich gilt:

- Code-Zeilen der zu testenden Klassen, die nie während des Tests ausgeführt werden, werden auch nicht getestet.
- "Alles" sollte getestet werden - d.h. jedes Statement sollte zumindest einmal ausgeführt werden.
- Es ist wichtig neben den "Normal"-Fällen unbedingt auch die "Grenz"- bzw. "Extrem"-Fälle zu prüfen.
- Es wird unterschieden zwischen Positiv-Tests und Negativ-Tests.
  Ein **Positiv-Test** (auch verifizierender Test) überprüft dahingehend, dass ein korrektes Resultat bei sinnvoller Anwendung/Eingabe erzielt wird.
  Ein **Negativ-Test** (auch Provokations- oder Robustheits-Test) überprüft dahingehend, dass sinnvoll auf eine sinnlose/falsche Anwendung/Eingabe reagiert wird. Es wird getestet, dass die SW robust gegenüber Benutzungsfehlern ist. Beispielsweise `assert` aktuelle Parameter ist sinnvoll `: "Illegal Argument";` wäre gemäß Vorlesungsstand eine typische Implementierung, die Robustheit gewährleistet. Uns fehlt noch das Wissen, wie ein derartiger Test zu implementieren ist.
- Es wird unterschieden zwischen Blackbox-Tests und Whitebox-Tests.
  Ein **Blackbox-Test** testet, ob ein (z.B. durch ein Interface) eingefordertes Verhalten vorliegt. Das Erstellen des Blackbox-Tests erfordert keine/niemals Kenntnisse des Codes, der getestet wird, und ist vor der Implementierung möglich. *Jede Implementierung, die das von außen Geforderte leistet, ist zu lässig.*
  Ein **Whitebox-Test** erfordert Kenntnisse des Codes, der getestet wird, bzw. der inneren Funktionsweise des zu testenden SW-Moduls/Systems. Es wird getestet, dass die Eigenschaften der Implementierung im Inneren selbst den Vorstellungen entsprechen.
- Mit einem Test kann nur die Anwesenheit von "bestimmten Fehlern" überprüft werden. Die Abwesenheit (fehlerfreier Code) würde einen "erschöpfenden Test" erfordern. Ein erschöpfender Test ist in der Praxis (für ernsthaften Code und unter realistischen Bedingungen) i.d.R. weder erstellbar noch ausführbar.

Der gegebene UnitTestFrame weist einige Testlücken auf.
Es wird bei dieser Aufgabe **nicht** erwartet, dass Sie Alles testen bzw. alle Testlücken schließen – schreiben Sie einfach min. zwei eigene Tests. Es gibt mehr als genügend Testlücken, die sich anbieten  sie zu schließen ;-)

## *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>`**.

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.

Am Ende gibt es eine Kurz-Dokumentation zur **`WordGrabber`**-Klasse.

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. Die Klasse Word darf in **keiner** wie auch immer gearteten Weise von String abgeleitet werden.
*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 und Obiges zur 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.

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? Und warum wird eine HashMap eingefordert?

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>` )

Es folgt ein Beispiel für die Nutzung des WordGrabbers. Es werde alle Wörter der jeweiligen Datei ausgegeben. Damit dies Sinn macht, muss die Datei vereinfacht eine (Plain-ASCII-)Textdatei sein.

```
final String fileNameWithPath = ...
final 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
```

## *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 **`MultiPurposeList<>`**, die das Interface **`MultiPurposeList_I<>`** unterstützt, soll eine universelle verkettete Liste von Ihnen selbst implementiert werden. Wenn Ihnen der Name nicht gefällt, dann dürfen Sie diesen Klassennamen gerne per Refactoring in einen sinnvolleren Namen abändern. Methodennamen müssen unverändert bleiben. Sie dürfen für Ihre Implementierung 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 können voraussetzen, dass keine Doppelten im Zusammenhang mit der Methode **`boolean remove(`** Informations-Objekt **`)`** auftreten.

Die folgenden Methoden sollen unterstützt werden:

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

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

**`void 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.
Wenn die Liste n Informations-Objekte aufweist, dann sind 0, ..., n gültige Positionsangaben. D.h. Positionsangabe 0 bedeutet "ganz am Anfang" und n "ganz am Ende".

Informations-Objekt **`setNo( int ,`** Informations-Objekt **`)`**
Bei gültiger Positionsangabe, 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 in den Ausgangs- bzw. Initialisierungs-Zustand. In HW wäre es eine RESET.

**`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.

Zur Reduzierung der Implementierungsaufwände sind die folgenden internen Hilfsmethoden zu implementieren. Denken Sie an DRY-Code ::= Don't Repeat Yourself - dies gilt insbesondere, wenn der entsprechende Code nicht trivial ist.

> 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.

Es gibt viele Freiheitsgrade bei dieser Aufgabe. Von daher ist die folgende Aussage "unscharf". Bei einer "üblichen Implementierung" würde **iSearchNode** 2x, **iGetNodeNo** 4x und **iRemoveNode** 3x aufgerufen / genutzt werden.

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

> **void printElemFirstToLast()**
> 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 printElemLastToFirst()**
> 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 printNodeFirstToLast()**
> 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 printNodeLastToFirst()**
> 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.

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 )
```

Die Werte der Felder/Exemplarvariablen einer Disc sind unveränderlich. Zwei **Disc**-Objekte, die sich nicht bzgl. ihrer Werte der Felder/Exemplarvariablen unterscheiden lassen, sollen von equals() als gleich erkannt werden.

Sofern keine Struktur vorgegeben ist, organisieren Sie Ihr Projekt wie folgt:

Ein (Sub-)**package multiPurposeList** für die Klasse **MultiPurposeList** und eine nötige Knoten-Klasse.
Ein (Sub-)**package media** für die Referenztypen **Disc**, **CD**, **DVD** und möglichen weiteren Hilfs-Referenztypen.

**Freiwillige Zusatz-(Teil-)Aufgabe:**

Erweitern Sie die Funktionalität der MultiPurposeList um:

> **`MultiPurposeList<> subList(`** ***Informations-Objekt*** **`)`**
> Liefert eine Unterliste ab dem Informations-Objekt. Wird es nicht in der Liste gefunden, soll eine **leere Liste** abgeliefert werden. Die Informations-Objekte mit zugehörigen Knoten verbleiben in der Liste.
> Achten Sie auf den Aufwand Ihrer Implementierung – durchlaufen Sie die Originalliste **nur einmal**.
>
> **`MultiPurposeList<> subList(`** ***Informations-Objekt*** **`,`** ***Informations-Objekt*** **`)`**
> Liefert eine Unterliste ab dem "ersten" Informations-Objekt (das 1. gefundene der beiden übergebenen vom Listenanfang aus gesehen) bis zum (einschließlich) anderen Informations-Objekt. Wenn beide als Parameter übergebene Informations-Objekte gleich sind, dann besteht die Ergebnis-Sub-List aus dem einen Informations-Objekt (sofern es in der Liste enthalten ist).
> Falls nur (genau) ein Informations-Objekt gefunden wird, soll eine Unterliste ab diesem Informations-Objekt abgeliefert werden.
> Die Informations-Objekte mit zugehörigen Knoten verbleiben in der Liste.
> Sofern beide Informations-Objekte nicht gefunden werden, soll in eine **leere Liste** abgeliefert werden.
> Achten Sie auf den Aufwand Ihrer Implementierung – durchlaufen Sie die Originalliste **nur einmal**.

# Freiwillige Zusatzaufgaben

Es folgen freiwillige Zusatzaufgaben. D.h. diese Aufgabe ist freiwillig ;-).
Wenn Sie diese freiwillige Zusatzaufgabe freiwillig lösen, dann haben Sie den "Gewinn", dass Sie mehr geübt haben und dass Sie Ihre Lösung für diese freiwillige Zusatzaufgabe im Labor besprechen können (sofern Zeit ist – Pflichtaufgaben haben Vorrang).

## *Freiwillige Zusatzaufgabe Z6.1 Fibonacci (mit BigInteger)*

Wir erinnern uns an Aufgabe A1.3 ;-)
Die Fibonacci-Reihe ist eine unendliche Folge von ganzen positiven Zahlen $f_0$, $f_1$, $f_2$, ... .
(Siehe http://de.wikipedia.org/wiki/Fibonacci-Folge ) Die ersten beiden Zahlen sind:

$f_0 = 0$,
$f_1 = 1$,

Jede weitere Zahl ist die Summe der beiden Vorgängerzahlen :

$f_n = f_{n-2} + f_{n-1}$ für n >= 2

Der Anfang der Fibonacci-Reihe lautet also: 0, 1, 1, 2, 3, 5, 8, ...
Geben Sie die ersten n Werte aus – n soll in einer Variablen `n` vom geeigneten Typ abgelegt werden.
Die ausgegeben Zahlen sind durch Komma zu trennen. Vor der ersten Zahl und nach der letzten Zahl darf jedoch kein Komma stehen.

Bedenke: Was passiert z.B. bei Eingaben/Startwerten wie 1, 0 oder gar -1 ?

Es sei bemerkt, dass -warum auch immer- die "ursprüngliche" Fibonacci-Reihe wie folgt beginnt:

$f_1 = 1$,
$f_2 = 1$,

Es steht Ihnen frei ob Sie die "ursprüngliche" oder die in Wikipedia beschriebene Fibonacci-Folge implementieren. Nur sagen Sie dies vor der Abnahme deutlich an und setzen Sie es konsequent um.

Schreiben Sie eine Klasse `FibonacciNumberComputer` (die in einem Package `fibonacci` liegen soll) und das Interface `FibonacciNumberComputer_I` implementiert mit den Methoden:

- `void printFirstFibonacciNumbers( int )`
  Der Parameter bestimmt jeweils die Anzahl der (ersten) Werte, die ausgeben werden sollen. Der Parameter ist also der Wert von n (siehe oben).
- `BigInteger computeFibonacciNumber( int )`
  Der Parameter bestimmt den gewünschten Wert in der Fibonacci-Folge.

Es soll intern mit dem Datentyp BigInteger gerechnet werden.
Hinweis/Test: Der letzte ausgegebene Wert bei Aufruf von `printFirstFibonacciNumbers( x )` muss der gleiche Wert sein, den der Aufruf von `computeFibonacciNumber( x )` liefert.

Für diese Aufgabe wird kein Code gestellt. Sie sind weit genug um "alles" selber scheiben zu können.

## *Freiwillige Zusatzaufgabe Z6.2 FrequencyComputer*

Schreiben Sie eine Klasse ListAnalyzer, die das Interface ListAnalyzer_I unterstützt. Der ListAnalyzer bestimmt insbesondere für eine als Parameter übergebene Liste, wie oft die jeweiligen in der Liste enthaltenen Elemente konkret in der Liste enthalten sind.
Für alle weiteren Details siehe gegeben Code bzw. Interfaces.

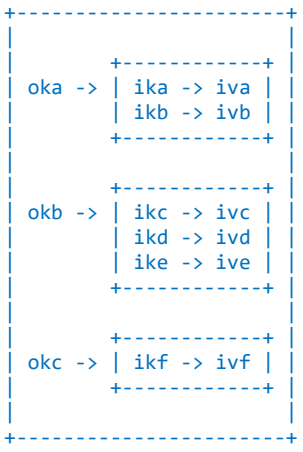
## *Freiwillige Zusatzaufgabe Z6.3  Collection Converter*

Beim Datentransfer bzw. Datenaustausch zwischen unterschiedlichen Rechnersystemen macht es Sinn komplexere Datenstrukturen in einfachere Datenstrukturen wandeln zu können und umgekehrt einfachere Datenstrukturen wieder in komplexere Datenstrukturen zurückwandeln zu können. Z.B. die Objektserialisierung, die beliebig komplexe Objekte in einen einfachen Datenstrom wandeln sowie daraus zurückgewinnen kann, ist in "diesem Zusammenhang motiviert". Dies wird vermutlich Thema im 2.Semester, aber **nicht** jetzt!

In dieser Aufgabe sollen Sie nun einen CollectionConverter implementieren, der
eine geschachtelte Map<Object,Map<Object,Object>> in eine flache List<Object> wandeln kann und
eine flache List<Object> in eine geschachtelte Map<Object,Map<Object,Object>> zurückwandeln kann.

Beispiel / "Idee":

Map<Object,Map<Object,Object>> konvertiert nach List<Object>

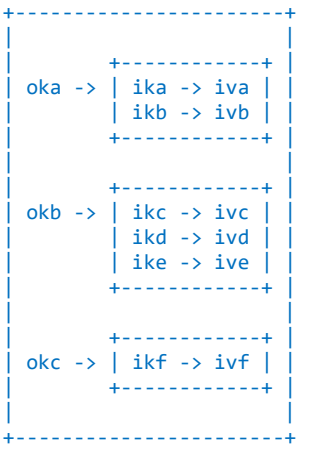



```
oka,ika,iva,oka,ikb,ivb,okb,ikc,ivc,okb,ikd,ivd,okb,ike,ive,okc,ikf,ivf
```

und

List<Object> konvertiert nach Map<Object,Map<Object,Object>>

```
oka,ika,iva,oka,ikb,ivb,okb,ikc,ivc,okb,ikd,ivd,okb,ike,ive,okc,ikf,ivf
```




Das Präfix
`ok` steht für **o**`uter` **k**`ey`
`ik` steht für **i**`nner` **k**`ey`
`iv` steht für **i**`nner` **v**`alue`

Bemerkung: Jetzt wechselt die Bedeutung der Farben.

Die **Map<**Object**,**Map<Object,Object>**>** ist eine (äußere/outer) Map, die eine (innere/inner) **Map<**Object**,**Object**>** als Daten enthält. Die äußere Map enthält Schlüssel ok… die jeweils eine innere Map identifizieren. Die innere Map enthält Schlüssel ik… die jeweils einen inneren Datensatz iv… identifizieren. Wenn die äußere Map in eine Liste gewandelt wird, dann sind Tripel (ok…,ik…,iv…) zu bilden. Die Verkettung der Elemente dieser Tripel bildet die Liste.

Bei der Wandlung in die umgekehrte Richtung ist eine entsprechende List<Object> (bestehend aus einer Verkettung von entsprechenden Tripeln (ok…,ik…,iv…) ) in eine **Map<**Object**,Map<**Object**,**Object**>>** zu konvertieren.

Die Abbildung zuvor soll dies visualisieren. Aber, Achtung: Sofern keine sortierte Datenstruktur verwendet wird, kann natürlich keine Aussage bzgl. der Sortierung getroffen werden. Dies ist bei der Abbildung zu berücksichtigen.
Jedoch gilt bzgl. des Aufbaus der Liste, dass sich deren Elemente als Verkettung der Elemente der Tripel (ok…,ik…,iv…) auffassen lassen – andernfalls würde es nicht funktionieren.