Vorbemerkung zum 5.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 um weitere eigene Parameter ergänzen (für zusätzliche eigene Tests).
  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.
- Ein **wichtiges (Lern-)Ziel der folgenden Aufgaben** ist auch, dass Sie die **richtige Datenorganisation** an der richtigen Stelle verwenden. Also diejenige Datenorganisation, die für den Zweck optimal geeignet ist und die nötigen darauf wirkenden Operationen optimal unterstützt – im Sinne von zu möglichst einfachen Methoden führt. **Dass es "nur" funktioniert reicht nicht!** Auch müssen Sie die Auswahl begründen können.
  Achtung! Wir haben verschiedene ADT, Collections, Map, … mehr oder weniger intensiv kennengelernt. Dies sind Datenorganisationen.

Zunächst wieder Vorbereitungsaufgaben mit denen Sie sich beschäftigen sollen. Auch wenn die Bearbeitung dieser Aufgabe Pflicht ist, wird Ihre Bearbeitung (vermutlich) nicht im Labor kontrolliert. Sie können aber gern Verständnisfragen im Labor stellen.

Aufgabe V5.1 Vorbereitungsaufgabe: N gleicher Farbe

Lesen Sie zunächst in der Klasse `CardProcessor` im Package `cardProcessor` den Methodenkopf-Kommentar der Methode `drawSameColour()` durch. Schauen Sie sich dann die Implementierung an und verstehen Sie diese.

Aufgabe V5.2 Simple (Hash-)Set Implementation

In der Vorlesung wurde die Idee des Hashing besprochen. Im Package `hashSetBasedOnFixedSizedArray` finden Sie nun eine Implementierung des Interfaces Set vor. Wie der Name schon andeutet handelt es sich hierbei um eine HashSet, die intern ein Array konstanter Größe als "HashTable" nutzt. Jeweils im Package:

`dataForTests` sind unterschiedliche Varianten einer "Personen-Klasse" zu finden
`hashSetBasedOnFixedSizedArray` ist die konkrete Implementierung der HashSet zu finden
`testsForHashSetBasedOnFixedSizedArray` sind einige Tests zu finden.

Objekte unterschiedlicher Personen-Klassen werden als Daten in der HashSet gespeichert. Z.B. in der Klasse `Pi` werden `equals()` und `hashCode()` von der Klasse `Object` geerbt, während sie in der Klasse `Po` überschrieben werden. Die Klasse `Px` ist für eigene Experimente gedacht.

Schauen Sie sich den gestellten Code an und versuchen Sie das in der Vorlesung besprochene nachzuvollziehen. In der Klasse `HashSetBasedOnFixedSizedArray` sollte (fast) alles oberhalb des markierten Bereichs (ca. Zeile 480) mit dem Vorlesungswissen verständlich sein. Lediglich die Generics wurde (u.U.) für das hier abgeforderte Code-Verständnis unzureichend besprochen. Das betrifft die Stellen mit den "spitzen Klammern" `<…>`. Also z.B. `public boolean addAll( final Collection<? extends T> coll ){…` Dies sollte aber für das grundsätzliche Verständnis der Algorithmen kein Problem darstellen.

In der Personen-Klasse `Px` können Sie eigene Experimente bzgl. der Methoden `equals()` und insbesondere `hashCode()` wagen und deren Auswirkung z.B. auf das Eintragen in die Set und das Entfernen aus der Set austesten. Sie haben die Klassen `Pi` und `Po` als Vergleich.
Denken Sie dran, dass das `equals()` der Klasse Object einen Identitätstest macht.

Aufgabe V5.3 Simple (Hash-)Map Implementation

Im Package **hashMapBasedOnFixedSizedArray** finden Sie nun eine Implementierung des Interfaces Map vor. Wie der Name schon andeutet handelt es sich hierbei um eine HashMap, die intern ein Array konstanter Größe als "HashTable" nutzt. Jeweils im Package:

dataForTests sind u.a. unterschiedliche Varianten einer "Personen-Klasse" zu finden
hashMapBasedOnFixedSizedArray ist die konkrete Implementierung der HashMap zu finden
testsForHashMapBasedOnFixedSizedArray sind einige Tests zu finden.

Objekte der unterschiedlichen Personen-Klassen werden als Key in der HashMap gespeichert. Z.B. in der Klasse **Pi** werden `equals()` und `hashCode()` von der Klasse `Object` geerbt, während sie in der Klasse **Po** überschrieben werden. Die Klasse **Px** ist für eigene Experimente gedacht.

Schauen Sie sich den gestellten Code an und versuchen Sie das in der Vorlesung besprochene nachzuvollziehen.

In der Personen-Klasse **Px** können Sie eigene Experimente bzgl. der Methoden `equals()` und insbesondere `hashCode()` wagen und deren Auswirkung z.B. auf das Eintragen in die Map und das Entfernen aus der Map austesten. Sie haben die Klassen **Pi** und **Po** als Vergleich.
Denken Sie dran, dass das `equals()` der Klasse Object einen Identitätstest macht.

Aufgabe A5.1 In umgekehrter Reihenfolge ausgeben

Ergänzen Sie im Package `cardProcessor` im Klassen-Template `CardProzessor` eine Methode `reverseOrder()`, die zunächst solange Karten von einem gegebenen Kartenstapel zieht bis eine gewünschte Karte gezogen wurde und danach alle bisher gezogenen Karten in umgekehrter Reihenfolge ausgibt.
Also, die letzte gezogene Karte (das ist die gewünschte Karte) zuerst ausgeben und die erste gezogene Karte zuletzt ausgeben. (Ja, wir denken "LIFO" bzw. Last In First Out ;-)
Es darf vorausgesetzt werden, dass der Kartenstapel 52 Karten enthält bzw. die gewünschte Karte auch im Kartenstapel vorhanden ist.
Die Methode soll 3 Parameter aufweisen, die in der nachfolgenden Reihenfolge

- den gegebenen Kartenstapel,
- die gewünschte Karte und
- einen Wahrheitswert für eine optionale Ausgabe der jeweils gezogenen Karte unmittelbar nach der Ziehung entgegen nehmen.

Siehe hierzu auch den gestellten TestFrame.

Aufgabe A5.2 Keine doppelten Karten

Ergänzen Sie im Package `cardProcessor` im Klassen-Template `CardProzessor` eine Methode `removeDuplicates()`, die beliebig viele Karten entgegen nimmt (in Form eines Arrays). Aus diesen Karten sollen die Doppelten[1] entfernt werden. Die so bereinigten Karten (also die Karten befreit von Doppelten) sind als Ergebnis (konkret Array über Karten) zurückzugeben.
Siehe hierzu auch den gestellten TestFrame.

[1] Zwei Karten, die sich nicht bzgl. ihrer Attribute unterscheiden lassen, werden als gleich angesehen und zählen damit als Doppelte. Sofern Doppelte vorgefunden werden, ist es egal welche der Doppelten entfernt werden. Es gilt: Nach dem Entfernen, darf es keine Doppelten mehr geben.

Aufgabe A5.3 Mit Comparator sortieren lassen

Schreiben Sie im Package `cardComparator` einen Comparator `UsualOrder`, der Karten vergleicht und es Ihnen ermöglicht eine Liste über Karten (List<Card>) mit Collections.sort() zu sortieren.
Die Karten sollen nach Aufruf von Collections.sort() nach folgender Ordnung sortiert sein.
Mit **erste**r **Priorität** nach **Ränge**n (Rank) und zwar Ass vor König, König vor Dame, Dame vor Bube, Bube vor 10, 10 vor 9, 9 vor 8, 8 vor 7, 7 vor 6, 6 vor 5, 5 vor 4, 4 vor 3 und 3 vor 2.
Mit **zweite**r **Priorität** nach **Farbe**n (Suit) und zwar Kreuz vor Pik, Pik vor Herz, Herz vor Karo.
Siehe hierzu auch den gestellten TestFrame.

Aufgabe A5.4 CardProcessor – Drillinge finden

Implementieren Sie im Package `tripleFinder` 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 ein 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 ein 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. Diese Karten sind bzw. dieser Drilling ist danach **nicht** mehr im Bestand!
  Der Typ der Rückgabewerts ist bewusst Object um möglichst "wenig Hilfestellung" zu geben.

  `void reset()`
  löscht alle (intern) gespeicherten Karten. Nach `reset()` befindet sich der `CardProcessor` im Ausgangszustand.

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).

Aufgabe Z5.1 Thingy Collector – (immer wieder) 7 gleicher Farbe zusammenstellen

Aus der Vorbereitungsaufgabe V4.2 sind Thingies/Items bekannt.

Implementieren Sie nun eine Klasse **`ItemProcessor`**, die sich von einem zu erstellenden Interface **`ItemProcessor_I`** ableitet und die **`Item`**`s` verarbeitet. Die Idee ist: Items zu sammeln und sobald 7 gleicher Farbe vorliegen, diese abzuliefern. 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.

Der ItemProcessor soll einkommende Items sammeln und immer sobald 7 Items gleicher Farbe vorliegen, sollen diese "abgeliefert" werden. Abgelieferte Items sind mit der Ablieferung nicht mehr im Bestand. Ein Item darf also nur jeweils einer (Ergebnis-)Zusammenstellung angehören, sofern das Item nicht danach wieder dem ItemProcessor angeboten wird.

Die Klasse **`ItemProcessor`** soll das von Ihnen zu erstellenden Interface **`ItemProcessor_I`** unterstützen und die folgenden Elemente aufweisen:

`ItemProcessor()`
erzeugt/initialisiert einen ItemProzessor.

`Sevensome<Item> process( Item )`
verarbeitet ein Item. Das als Parameter übergebene Item wird den "bisher übergebenen" Items hinzugefügt.
Immer wenn 7 Items gleicher Farbe Items vorliegen, sollen diese 7 als Rückgabewert der Methode (in Form eines "geeigneten Sevensome") abgeliefert werden – andernfalls soll **`null`** zurückgegeben werden.
Items, die bereits Teil einer (Ergebnis-)Zusammenstellung (also eines abgelieferten Sevensomes) waren, dürfen nicht für die Bildung einer weiteren Zusammenstellung verwendet werden - solche Items sind also - *nachdem sie Teil eines Ergebnisses waren* - aus dem "Gedächtnis" zu entfernen. Ein Item darf nur jeweils maximal einer (Ergebnis-)Zusammenstellung angehören[2]. (*Mögliche interne Collections sind hiervon ausgeschlossen.*)

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

Von Ihnen sind sollen also mindestens die Referenztypen: **`ItemProcessor_I`**, **`ItemProcessor`** und der **generische** Referenztype **`Sevensome`** zu erstellen.

Bemerkung:
Ein **`Sevensome`** ist ein "Sevensome". D.h. ein Sevensome enthält immer exakt 7 "Dinge".
Das Sevensome ist generisch – soll also auch für andere Zwecke verwendbar sein.
Ein **`Sevensome<Item>`** soll u.a. das folgende Element aufweisen:

`T get( int )`
liefert das jeweilige Element des Sevensome*s*.

Gern können Sie sich auch ein Interface **`Sevensome_I`** schreiben, das von **`Sevensome`** implementiert wird. Sofern Sie dies tun, sollte der Rückgabetyp von process natürlich **`Sevensome_I`** sein.

[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.