PR1, Aufgabenblatt 6

Programmieren I – Wintersemester 2018/19

Sammlungen benutzen

Ausgabedatum: ..... 5. Dezember 2018

Lernziele

Sammlungen (Interfaces und implementierende Klassen) des Java Collections Frameworks benutzen können; die Unterschiede zwischen einer Menge (`Set`) und einer Liste (`List`) kennen; Gleichheit und Identität unterscheiden können; Typtests und Typzusicherungen verstehen und programmieren können; die erweiterte for-Schleife für Sammlungen benutzen können; Abbildungen (`Map`) benutzen können.

Kernbegriffe

In praktisch allen Anwendungen werden *Sammlungen* gleichartiger Objekte manipuliert. Für die alltägliche Programmierung stellen Programmierbibliotheken meist Sammlungen als *dynamische Behälter* zur Verfügung, in die beliebig viele *Elemente* eingefügt und aus ihnen auch wieder entfernt werden können. Dabei sind zwei Eigenschaften für Klienten von solchen Sammlungen besonders relevant:

- Haben die Elemente in der Sammlung explizit eine *manipulierbare Reihenfolge* (wie bei einer Liste) oder ist ihre *Ordnung irrelevant* (wie beispielsweise bei einer allgemeinen Menge)?
- Sind *Duplikate* in der Sammlung zugelassen (wie bei einer Liste) oder darf ein Element nur einmal vorkommen (wie bei einer Menge)?

Wir konzentrieren uns hier zunächst auf diese Benutzungsaspekte von Sammlungen, indem wir Listen und Mengen benutzen. Techniken zur ihrer Implementierung kommen auf dem nächsten Aufgabenblatt.

Um für eine Sammlung entscheiden zu können, ob ein Element bereits enthalten ist, muss es ein Konzept von Gleichheit geben. Wir unterscheiden für Objekte *Gleichheit* von *Identität*: Zwei Objekte einer Klasse können *gleich* sein (etwa die gleichen Werte in ihren Exemplarvariablen haben), sind aber niemals *identisch* (ein Objekt ist nur mit sich selbst identisch). Gleichheit impliziert also nicht Identität, aber Identität impliziert Gleichheit: Wenn zwei Variablen/Referenzen auf *dasselbe* Objekt verweisen, verweisen sie automatisch auch auf *das gleiche* Objekt.

Alle Objekte in Java können auf Gleichheit miteinander verglichen werden, da an jedem Objekt die Operation `boolean equals(Object other)` aufgerufen werden kann. Sie ist in der Klasse `Object` definiert, die eine Handvoll Operationen definiert, die jedes Objekt in einem Java-System anbietet. Der Operation `equals` wird als Parameter das Exemplar mitgegeben, mit dem es verglichen werden soll. In Java-Klassen ist diese Methode standardmäßig als Prüfung auf Identität realisiert, sofern keine eigene `equals`-Methode implementiert wird.

Die *Sammlungsbibliothek* von Java (engl. *Java Collection Framework*, kurz JCF) stellt verschiedene Interfaces und Klassen für verschiedenartige Sammlungen zur Verfügung. Seit der Java-Version 5 ist es möglich, den Typ der Elemente einer Sammlung mit anzugeben. So deklariert beispielsweise

```
List<String> myList;
```

eine Variable `myList`, die nur auf Listen verweisen kann, die ausschließlich Strings enthalten, und `new ArrayList<String>()` erzeugt ein Exemplar der das Interface `List` implementierenden Klasse `ArrayList`, in dem nur Strings gespeichert werden können.

Ebenfalls seit der Java-Version 5 ermöglicht die *erweiterte for-Schleife* (engl.: for-each loop), die Elemente einer Sammlung zu durchlaufen. So gibt beispielsweise die folgende Schleife über die oben deklarierte Liste `myList` für jeden String in der Liste seine Länge aus:

```
for (String s : myList) // lies: für jeden String s in myList ...
{
    System.out.println(s.length());
}
```

Neben Set und List gibt es im JCF das Interface Map, das den Umgang mit *Abbildungen* modelliert. Eine Abbildung ist eine Sammlung von *Schlüssel-Wert-Paaren*, in der die Schlüssel eindeutig sein

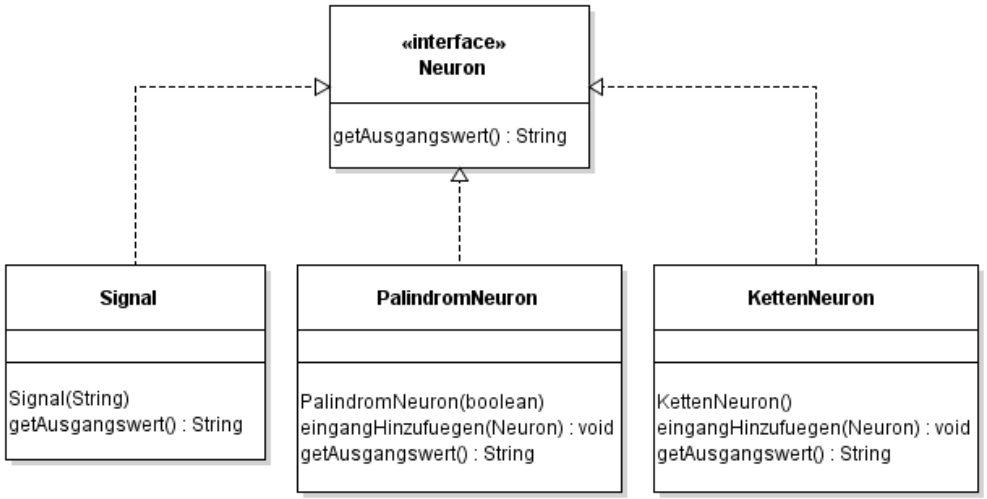
müssen. Als Wert kann jeder Referenztyp dienen, beispielsweise auch eine Sammlung. Über die Operation `get` kann mit einem Schlüssel bequem auf einen gespeicherten Wert zugriffen werden.

Die Bibliotheken der Sprache Java sind in so genannten *Paketen* (engl.: packages) organisiert. Das Paket der Sammlungsbibliothek heißt `java.util`. Klassen, die Bibliotheksklassen und -Interfaces benutzen, importieren diese mit einer *Import-Anweisung* (z.B. `import java.util.ArrayList;`). Die Programmier-Schnittstelle (engl: *API – Application Programming Interface*) der verschiedenen Bibliotheken ist in der *API Specification* beschrieben, siehe http://docs.oracle.com/javase/8/docs/api/. Dort findet sich u.a. die Dokumentation der Sammlungsbibliothek und die der Operation `equals` aus der Klasse `Object` (im Paket `java.lang`).

Aufgabe 6.1 Neuronales Netz für Zeichenketten (Termin 1)

In dieser Aufgabe entwickeln wir ein Netzwerk zur Verarbeitung von Zeichenketten, das durch den Aufbau digitaler *Neuronaler Netze* inspiriert ist. Im Kern eines solchen Netzes stehen *Neuronen*. Jedes Neuron hat mehrere Eingänge und einen Ausgang. Aus den Signalen an den Eingängen wird das Signal am Ausgang berechnet.

In unserer Anwendung werden Signale als Zeichenketten dargestellt. Außerdem drehen wir die Aktivierungsreihenfolge um: in der Realität sendet ein Neuron ein Signal am Ausgang, wenn ein Schwellwert von Eingangssignalen überschritten wird; wir hingegen „sammeln“ die Werte an den Eingängen und berechnen daraus den Ausgangswert, wenn dieser abgefragt wird. Das folgende Klassendiagramm gibt eine Übersicht, was insgesamt in 6.1 implementiert werden soll:

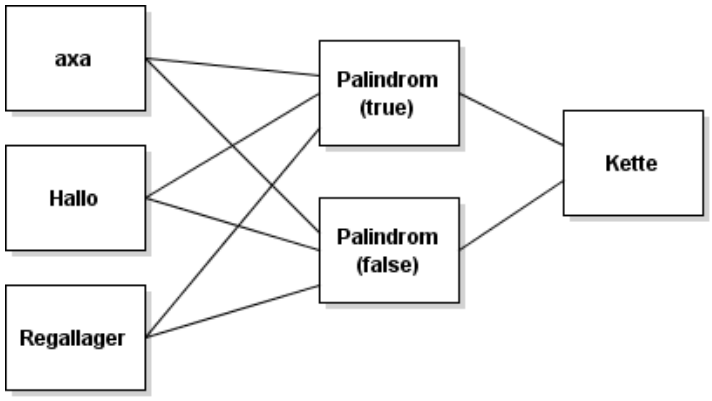


6.1.1 Wir benötigen zunächst ein Interface `Neuron` für alle Neuronen. Dieses bietet eine Operation `getAusgangswert` für den Zugriff auf den Wert am Ausgang (einen String).
Die Eingangswerte eines neuronalen Netzes werden durch spezielle Neuronen geliefert, die selber keine Eingänge haben: `Signal`. Diese liefern am Ausgang einen Wert, der über den Konstruktor gesetzt wird. Alle anderen Neuronen bieten an ihrer Schnittstelle eine Operation `eingangHinzufuegen`, mit der jeweils ein Neuron als weiterer Eingang hinzugefügt wird.

6.1.2 Ein `PalindromNeuron` ist in der Lage, Palindrome zu erkennen. Um seinen Ausgangswert zu bestimmen, prüft es seine Eingangswerte auf Palindrome. Ist nur einer der Eingangswerte ein Palindrom, wird dieser Wert zurückgegeben; liegen mehrere Palindrome an den Eingängen an, wird das längste geliefert; ansonsten wird `null` geliefert. Ein `PalindromNeuron` kann in einem von zwei Modi arbeiten: Entweder wird Groß- und Kleinschreibung unterschieden oder nicht; diese Eigenschaft wird über einen Konstruktorparameter gesetzt.
Welcher Sammlungstyp ist hier für das Speichern der Eingänge geeignet?

6.1.3 Ein `KettenNeuron` verkettet die Werte seine Eingänge, in der Reihenfolge, in der sie hinzugefügt wurden. Zwischen den Eingangswerten fügt es jeweils ein Leerzeichen ein. Es liefert mindestens eine leere Zeichenkette. Außerdem hat dieses Neuron ein Gedächtnis: es merkt sich seine letzte Ausgabe und schickt diese zusätzlich seiner nächsten Ausgabe vorweg.
Welcher Sammlungstyp ist hier für das Speichern der Eingänge geeignet?

6.1.4 Als gute Softwareentwickler habt ihr natürlich(!) **Testklassen** zu euren Klassen geschrieben. Testet nun auch das Zusammenspiel der Klassen. Beispielsweise sollte das folgende Netzwerk…



… mit dem ersten Abfragen der Ausgabe des Ketten-Neurons liefern:

axa Regallager

und mit dem zweiten Abfragen:

axa Regallager axa Regallager

Testet diesen Fall mit JUnit. **Zeichnet**, ähnlich wie oben, mindestens ein weiteres Netzwerk und testet auch dieses mit JUnit. Zeichnet dann eines der Diagramme auch als **Objektdiagramm**.

Aufgabe 6.2 Das Geburtstagsparadoxon (Termin 1)

Ein bekanntes mathematisches Rätsel, von dem ihr vielleicht schon einmal gehört habt, ist das *Geburtstagsparadoxon*. Dabei geht es um die Frage, wie wahrscheinlich es ist, dass in einer Gruppe von Personen mehrere Leute am gleichen Tag Geburtstag haben (wobei das Geburtsjahr keine Rolle spielt). Die Wahrscheinlichkeit ist schon für kleinere Gruppen wie etwa Partys und Schulklassen erstaunlich hoch. Das Geburtstagsparadoxon ist eine Veranschaulichung der allgemeinen Frage nach der Kollision von Zufallszahlen und spielt z.B. in der Kryptographie eine wichtige Rolle.

Es gibt mathematische Formeln, die die Wahrscheinlichkeit einer Kollision exakt berechnen. Diese werdet ihr in Veranstaltungen kennen lernen, die sich mit Kombinatorik und Stochastik beschäftigen. Als angehende Softwareentwickler wollen wir hier einen anderen Ansatz wählen. Wir simulieren eine große Anzahl von Partys mit Gästen und leiten aus den Messergebnissen einen empirischen Wert für die Wahrscheinlichkeit ab.

6.2.1 Öffnet das Projekt *Geburtstag* und schaut euch die *Dokumentation* der Klasse `Tag` an. Im Kommentar von `equals` steht, dass zwei `Tag`-Objekte, die den gleichen Tag darstellen, als gleich angesehen werden. `Tag`-Objekte, die nicht den gleichen Tag darstellen, sollen natürlich als ungleich angesehen werden.
Überprüft dies, indem ihr mindestens diese beiden Testfälle in der vorgegebenen JUnit-Testklasse implementiert! Wie weit lässt sich die Klasse `Tag` sinnvoll testen?

6.2.2 Schaut euch das Interface `Party` an. Hier werden zwei Operationen definiert: `fuegeGeburtstagHinzu` wird aufgerufen, wenn ein Gast seinen Geburtstag verraten hat.

`mindestensEinGeburtstagMehrfach` liefert `true`, sobald zwei Gäste am gleichen Tag Geburtstag haben.

Schreibt eine Klasse `PartyMenge`, die das Interface `Party` implementiert. Fügt dazu in der Methode `fuegeGeburtstagHinzu` den übergebenen Geburtstag in ein `HashSet` von Geburtstagen ein. Das Interface `Set` definiert die Schnittstelle einer Menge. Eine Menge enthält keine Duplikate und die Elemente in einer Menge haben keine explizite Reihenfolge (bzw. die gekapselte, interne Reihenfolge ist nicht relevant für den Umgang mit einer Menge). Wie bemerkt ihr, ob ein Geburtstag bereits enthalten ist?

Testet eure Implementation, indem ihr ein Exemplar von `Simulation` erstellt und daran die Methode `test()` aufruft. Falls das Ergebnis `false` ist, habt ihr einen Fehler gemacht.

6.2.3 Schätzt zunächst, wie viele Gäste ungefähr nötig sein müssten, um Kollisionswahrscheinlichkeiten von 50%, 75% und 95% zu bekommen. Testet eure Vermutungen anschließend mit der Methode `simuliere(int gaeste)`.

Überlegt, wann die Wahrscheinlichkeit einer Kollision *exakt* 100% sein muss.

6.2.4 Kommentiert die Methode `equals` in der Klasse `Tag` aus (oder benennt sie einfach um). Führt nun eine Simulation mit 366 Gästen aus. Was beobachtet ihr? Woran liegt das? Schaut euch ggf. die Dokumentation der Methode `equals` in der Klasse `Object` an. **Sorgt anschließend dafür, dass `equals` wieder korrekt funktioniert** (Auskommentierung bzw. Umbenennung rückgängig machen).

6.2.5 Die `equals` Methode in `Tag` ist nach einem Muster aufgebaut, das aus drei Schritten besteht. **Beschreibt diese drei Schritte mit Begriffen aus dem Skript.**

Aufgabe 6.3 Prägende Informatiker

6.3.1 Öffnet das Projekt *Informatiker* und studiert das Interface `Vergleicher`.

Erzeugt jeweils ein Exemplar der Klasse `PraegendeInformatiker` und der Klasse `PerNachname`. Ruft auf dem ersten Exemplar die Operation `schreibeGeordnet` auf und übergebt das zweite Exemplar als Parameter. Daraufhin sollten die Informatiker per Nachnamen geordnet auf der Konsole erscheinen.

Die Klasse `PerNachname` vergleicht offenbar anhand des Nachnamens. Wozu mag wohl die Klasse `PerAlter` gut sein? Probiert sie aus!

Möglicherweise ist euch aufgefallen, dass gleichaltrige Personen untereinander keine dem Benutzer sinnvoll erscheinende Reihenfolge haben. Wäre es nicht praktisch, wenn man mehrere Vergleicher miteinander kombinieren könnte? Also erst das Alter vergleichen, und bei gleichem Alter den Nachnamen?

Genau dafür gibt es die Klasse `Zweistufig`. Erstellt ein Exemplar und übergebt als Parameter jeweils ein Exemplar der Klassen `PerAlter` und `PerNachname`. Wenn ihr nun das Exemplar der Klasse `Zweistufig` an `schreibeGeordnet` übergebt, sollte die Liste deutlich sinnvoller aussehen.

6.3.2 Schreibt eine Klasse `PerVorname`, welche die Vornamen der Personen miteinander vergleicht.

6.3.3 Schreibt eine Klasse `PerGeschlecht`, welche Frauen vor Männern einstuft. Schreibt auch eine JUnit-Testklasse dazu. Warum bietet sich dies hier an?

6.3.4 Könnt ihr durch geschickten Einsatz der Klasse `Zweistufig` auch 3 Vergleicher hintereinander schalten?

6.3.5 Schreibt eine Klasse `Umgekehrt`, welche sich genau umgekehrt zu einem anderen Vergleicher verhält. Beispielsweise soll `new Umgekehrt(new PerAlter())` junge Personen vor alten einstufen. Als Vorlage kann hierbei die (deutlich kompliziertere) Klasse `Zweistufig` dienen.

6.3.6 *Zusatzaufgabe*: Das Vergleichen zweier Objekte wie im Interface `Vergleicher` ist ein solch zentraler Gedanke, dass dafür in Java bereits das Interface `Comparator` im Paket `java.util` mitgeliefert wird:

https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Damit `Comparator` für beliebige Typen von Objekten funktionieren kann, gibt man in spitzen Klammern an, um welchen Typ es sich jeweils handelt. In unserem Fall würde man also `Comparator<Person>` schreiben. Dieses Parametrisieren von Typen mit anderen Typen wird erst im nächsten Semester ausführlich erläutert.

Legt eine Kopie des Projekts *Informatiker* an, indem ihr in BlueJ *Projekt* / *Speichern unter…* auswählt und dann einen beliebigen Namen wählt, z.B. *Informatiker2*.

Entfernt das Interface `Vergleicher` und verwendet stattdessen den Typ `Comparator<Person>`. Dazu müsst ihr in jedem Quelltext, der diesen Typ verwendet, folgendes in die erste Zeile schreiben:

```
import java.util.Comparator;
```

Aufgabe 6.4 Sammlungen benutzen: Ausbau der Mau-Mau-Simulation

Die Mau-Mau-Simulation von Blatt 4 ist bisher noch recht schwach und soll nun an einigen Punkten verbessert werden. Dazu gibt es eine neue Version des Projektes für dieses Aufgabenblatt im pub.

6.4.1 Seht euch die Implementation der Klasse `Spieler` genauer an. Warum wurde die Hand des Spielers als Liste und nicht als Set modelliert? **Schriftlich!**

6.4.2 Implementiert in der Klasse `Spieler` eine neue Operation `anzahlBuben`, die für die Karten, die der Spieler auf der Hand hat, die *Anzahl der Buben* ermittelt (beim Mau-Mau wichtig).

6.4.3 Neue Regel für Mau-Mau: Neben dem Gewinner wollen wir auch den Verlierer ermitteln. Verlierer sei, wer bei Spielende die meisten Punkte auf der Hand hat. Implementiert in der Klasse `Spieler` eine neue Operation `zaehlePunkte`, die für die Karten, die der Spieler auf der Hand hat, die Summe ihrer Punktwerte liefert. Dazu muss ein Spieler „wissen“, wie viele Punkte eine Karte wert ist (die 7, 8 und 9 beispielsweise haben den Wert 3, während ein As 11 Punkte wert ist, siehe die Readme-Datei im BlueJ-Projekt). Diese *Abbildung* von Karten(-rängen) auf Punktwerte sollt ihr mit einer geeigneten Java-Map umsetzen. Dazu sollt ihr eine bestehende Map-Implementation *benutzen* (nicht selbst eine implementieren!) Erweitert die Klasse `Spieler` um einen Konstruktorparameter, der die Abbildung auf die passenden Werte entgegennimmt. Erzeugt und befüllt in der Klasse `MauMauRunde` eine geeignete Map mit den entsprechenden Werten und übergebt diese an die drei Spieler. Mit diesem Wissen könnt ihr nun in der Klasse `Spieler` die Operation `zaehlePunkte` korrekt implementieren.

6.4.4 Und zum Schluss wieder etwas Abwechslung vom Mau-Mau spielen: Implementiert in der Klasse `MauMauRunde` eine Methode `zieheErstenDrilling`. Diese soll sich einen frischen Kartenstapel erzeugen und von diesem so lange jeweils eine Karte ziehen, bis ein *Drilling* (drei Karten desselben Ranges) gezogen wurde. Die Methode soll diese drei Karten als *Menge von Karten* zurückliefern.

6.4.5 *Zusatzaufgabe:* Die Schnittstelle der Klasse `Spieler` ist inzwischen etwas umfangreicher geworden. Unser Vertrauen in die korrekte Umsetzung der Dienstleistungen wäre höher, wenn wir eine Testklasse hätten. Schreibt eine JUnit-Testklasse für die Klasse `Spieler`, die mindestens die von euch neu implementierten Methoden testet. Welche Grenzfälle solltet ihr jeweils berücksichtigen?