SE1, Aufgabenblatt 11

Softwareentwicklung I – Wintersemester 2016/17

Sammlungen benutzen: Listen und Mengen

Moodle-URL: uhh.de/se1

Feedback zum Übungsblatt: uhh.de/se1-feedback

Projektraum ..... Softwareentwicklung 1- WiSe 2016/17

Ausgabewoche ..... 12. Januar 2017

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 Position (wie bei einer Liste) oder ist ihre Position 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 späteren Aufgabenblättern.

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. Wenn in einer Klasse keine spezielle Gleichheitsmethode implementiert ist (etwa ein Wertevergleich der Exemplarvariablen), ist in Java diese Methode standardmäßig als Prüfung auf Identität implementiert. Klassen, deren Exemplare auf Wertgleichheit überprüft werden sollen, müssen selber eine `equals`-Methode implementieren.

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

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

Das Interface `java.util.List` legt nur die Operationen eines Typs fest, ohne implementierende Methoden anzugeben. Es definiert die Schnittstelle eines Listenkonzepts (manipulierbare Reihenfolge, Zugriff über Positionen, etc.), das durch die Klassen `ArrayList` und `LinkedList` implementiert wird. Die beiden Implementierungen unterscheiden sich nur in der Geschwindigkeit, in der sie die Listenoperationen ausführen (wir werden das auf einem der nächsten Blätter genauer untersuchen). Verwendet werden beide Klassen jedoch immer nur unter dem Typ `List`, die konkrete Implementierung muss ausschließlich bei der Erzeugung des Listenobjekts angegeben werden.

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());
}
```

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 möchten, müssen diese mit einer *Import-Anweisung* importieren (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/7/docs/api/. Dort findet sich u.a. die Dokumentation der Sammlungsbibliothek und die der Operation `equals` aus der Klasse `Object` (im Paket `java.lang`).

Lernziele

Dynamische Sammlungen (Interfaces und implementierende Klassen) des JCF benutzen können; die Unterschiede zwischen einer Menge (`Set`) und einer Liste (`List`) kennen; Gleichheit und Identität unterscheiden können; die erweiterte for-Schleife für Sammlungen benutzen können.

Aufgabe 11.1 Eine Liste von Dateien

Öffnet das Projekt *DateiListe* und schaut euch die Klasse `DateiListe` an. Die Rümpfe der Methoden sind noch nicht sinnvoll implementiert. Das wollen wir im Folgenden nachholen!

11.1.1 Der Rumpf des Konstruktors ist noch leer. Welche Belegung hat das Zustandsfeld `_dateien`, nachdem der leere Konstruktor durchgelaufen ist? Welchen dynamischen Typ hat `_dateien`? **Schriftlich**.

11.1.2 Sorgt dafür, dass das Zustandsfeld `_dateien` eine sinnvolle Belegung erhält.

11.1.3 Vervollständigt den Rumpf der Methode `verarbeite`, so dass der Parameter der Liste hinzugefügt wird.

11.1.4 Vervollständigt den Rumpf der Methode `schreibeAufDieKonsole` mit einer klassischen for-Schleife.

Testet euren Code, indem ihr einen `VerzeichnisWanderer` und eine `DateiListe` erzeugt. Ruft dann die Methode `start` auf dem `VerzeichnisWanderer` auf und übergebt die `DateiListe`. Ruft zuletzt die Methode `schreibeAufDieKonsole` auf der `DateiListe` aus. Dann sollte man etwas sehen können.

11.1.5 Vervollständigt den Rumpf der Methode `loescheAlleEintrage`.

11.1.6 Vervollständigt den Rumpf der Methode `gesamtLaenge` mit einer erweiterten for-Schleife.

Aufgabe 11.2 Kontobewegungen protokollieren

Um mit den Sammlungen in Java noch besser vertraut zu werden, bauen wir ein bereits bekanntes Beispiel um: Öffnet das Projekt *Girokonto* mit der gleichnamigen Klasse. Die Möglichkeiten dieser Girokonten sind begrenzt, da sie zwar einen Kontostand (*Saldo*) haben, aber keine Auskunft über ihre bisherigen *Kontobewegungen* (Einzahlungen und Auszahlungen) geben können. Diesen Missstand wollen wir jetzt beheben.

11.2.1 Ändert die Klasse `Girokonto` so ab, dass sie intern ihre Kontobewegungen in einer Liste verwaltet. Benutzt hierzu Exemplare der Klasse `Kontobewegung` und speichert sie in einer `List`.

11.2.2 Erweitert die Schnittstelle der Klasse `Girokonto` um die Möglichkeit, alle Kontobewegungen auf die Konsole auszugeben. Erstellt hierzu eine Methode `druckeKontobewegungen`.

Aufgabe 11.3 Duplikate im Dateisystem aufspüren

Wenn man ein Dateisystem über viele Jahre benutzt, ist es fast unvermeidlich, dass sich Duplikate einschleichen und unnötigerweise Speicherplatz belegen. Gerade bei Musik- und Videodateien ist dieser Umstand ärgerlich, da diese Dateien besonders viel Speicherplatz fressen. Was liegt da also näher, als ein Java-Programm zu schreiben, dass ein Dateisystem nach ebensolchen Duplikaten durchsucht?

11.3.1 Öffnet das Projekt *Duplikate* und probiert das Programm erst mal aus. Erzeugt dazu ein neues Exemplar der Klasse `DuplikatSucher` und führt die `start`-Methode aus.

In der Konsole sollten jetzt Originale und Duplikate angezeigt werden. Die große Zahl am Ende einer Zeile ist ein sogenannter *Fingerabdruck*, der einen Datei-Inhalt fast eindeutig identifiziert. Die Wahrscheinlichkeit, dass inhaltsverschiedene Dateien denselben Fingerabdruck generieren, ist sehr viel kleiner als die Wahrscheinlichkeit, dass z.B. die Festplatte kaputt geht, der Strom ausfällt oder das Betriebssystem abstürzt. Wir können also im Folgenden davon ausgehen, dass der Fingerabdruck praktisch immer eindeutig ist.

Achtung: den Fingerabdruck bitte nicht mit dem `hashCode` verwechseln, dort sind Kollisionen aufgrund des beschränkten Wertebereichs von `int` und den simplen Berechnungsverfahren quasi unvermeidlich.

11.3.2 Schaut euch die Klasse `Datei` an und beantwortet folgende Fragen **schriftlich**:

Wie setzt sich der Zustand eines `Datei`-Exemplars zusammen?

Welcher Teil des Zustands wird in der überschriebenen `equals`-Methode berücksichtigt, d.h. wann werden zwei Exemplare von `Datei` als gleich angesehen? Kommentiert die Methode entsprechend.

Welcher Teil des Zustands hat einen Einfluss auf den `hashCode` einer Datei? Welcher Zusammenhang gilt allgemein zwischen `equals` und `hashCode`? Hält sich die Klasse `Datei` an diesen Vertrag?

http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals%28java.lang.Object%29

11.3.3 Schaut euch die Methode `equals` an. Was bedeutet der Ausdruck `(obj instanceof Datei)`? Hat `instanceof` etwas mit dem statischen Typ oder mit dem dynamischen Typ zu tun? **Schriftlich**.

11.3.4 Was bedeutet die Anweisung `Datei zweite = (Datei) obj;`? Welchen statischen Typ und welchen dynamischen Typ haben die hier beteiligten Variablen? Wird hier ein Objekt kopiert? **Schriftlich**.

11.3.5 Schaut euch die beiden Methoden `verarbeite` und `duplikatGefunden` an. Warum werden die Dateien in einem Set abgelegt, aber die Duplikate in einer Liste? Braucht man diese Liste überhaupt?

11.3.6 Wenn man nicht gerade eine digitale Mediathek nach Duplikaten untersucht, wird das Programm sehr viele kleine Dateien als Duplikate identifizieren, die zu völlig verschiedenen Programmen / Projekten gehören. Für solche Fälle wäre es sinnvoll, wenn Duplikate zusätzlich denselben Dateinamen haben müssten. An welchen Stellen müsste das Programm dafür angepasst werden? Nehmt diese Änderungen vor.

Aufgabe 11.4 Das Geburtstagsparadoxon

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 Schulklassen oder SE1-Übungsgruppen 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.

11.4.1 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 interaktiv mehrere Exemplare erzeugt und die equals-Methode aufruft.

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

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

Schreibt eine Klasse `MengenParty`, 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.

11.4.3 Schätzt zunächst durch Nachdenken, 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.

11.4.4 Kommentiert die Methoden `equals` in der Klasse `Tag` aus (oder benennt sie einfach um). Führt nun eine Simulation mit 100 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).

11.4.5 *Zusatzaufgabe:* Erweitert das Interface Party um eine Operation `anzahlKollisionen`, die darüber Aufschluss gibt, wie viele Kollisionen es auf der Party gibt. Schreibt eine passende Methode in `MengenParty`.