

Programmieren II (Java)

2. Praktikum: Grundlagen Objektorientierung



Sommersemester 2025

Christopher Auer

Lernziele

- ▶ Implementieren nach einer Spezifikation
- ▶ Aufbau von Klassen: Attribute und Methoden
- ▶ Konstruktoren: Verkettung, statische Factory Methoden
- ▶ Getter und Setter
- ▶ Java-Standard-Methoden: equals, toString
- ▶ Dokumentation: javadoc
- ▶ Testen mit JUnit
- ▶ **enums**: definieren, erweitern und verwenden
- ▶ Schauen Sie sich die Tutorial-Videos auf der *Moodle-Seite* mit *wichtigen Hinweisen* zum Praktikum und VS-Code an!
- ▶ Sie dürfen die Aufgaben *alleine* oder zu *zweit* bearbeiten und abgeben
- ▶ Sie müssen *4 der 5* Praktika bestehen
- ▶ *Kommentieren* Sie Ihren Code
 - ▶ Jede *Methode* (wenn nicht vorgegeben)
 - ▶ *Wichtige* Anweisungen/Code-Blöcke
 - ▶ *Nicht kommentierter* Code führt zu *Nichtbestehen*
- ▶ Bestehen Sie eine Abgabe *nicht* haben Sie einen *zweiten Versuch*, in dem Sie Ihre Abgabe *verbessern müssen*.
- ▶ *Wichtig*: Sie sind einer *Praktikumsgruppe* zugewiesen, *nur* in dieser werden Ihre Abgaben *akzeptiert*!

Bevor es losgeht

Bevor wir starten, lesen Sie sich *aufmerksam* folgende Hinweise durch:

- *Importieren* Sie zunächst das beigefügte *Gradle-Projekt* unter SupportMaterial/music in Ihre bevorzugte Entwicklungsumgebung. Erstellen Sie Ihre Klassen im Projekt-Verzeichnis unter:
src/main/java

Wenn Sie Ihre Klassen *woanders* erstellen, wird Ihr Projekt *nicht funktionieren*!

- *Dokumentieren* Sie *alle Klassen* und *öffentlichen Methoden* mit *JavaDoc*! Zusätzlich müssen Sie weiterhin Ihren Quellcode *kommentieren*.
- Verwenden Sie zum Prüfen auf *Gleichheit* von `String`s und anderen Objekten die Methode `equals`!
- Prüfen Sie die Parameter jeder Methode auf *Gültigkeit*! Sollte ein Parameter einen ungültigen Wert haben, erzeugen Sie eine `IllegalArgumentException` wie folgt:

```
throw new IllegalArgumentException("Aussagekräftige (!) Fehlermeldung");
```

Geben Sie eine *aussagekräftige Fehlermeldung* an!

- Achten Sie auf die korrekte *Sichtbarkeit* (`public`, `private`, `protected`) der Klassen, Attribute und Methoden! Es ist im Folgenden *nicht korrekt* einfach *keine Sichtbarkeit* anzugeben — und der Übungsleiter wird Sie darauf ansprechen!
- Testen Sie Ihre Klassen mit den mitgelieferten *JUnit*-Tests in den beigefügten Gradle-Projekten! Führen Sie dazu den Gradle-Task `test` aus. Die JUnit-Tests finden Sie im Projektverzeichnis unter:

src/test/java

Machen Sie sich außerdem mit der Unterstützung von JUnit-Tests in Ihrer Entwicklungsumgebung vertraut.

- *Tipp*: Haben Sie die Implementierung einer Methode abgeschlossen, kommentieren Sie die Tests mit entsprechenden Namen ein. Diese sind nach dem Schema



MethodName_Bedingungen_ErwartetesErgebnis

benannt.

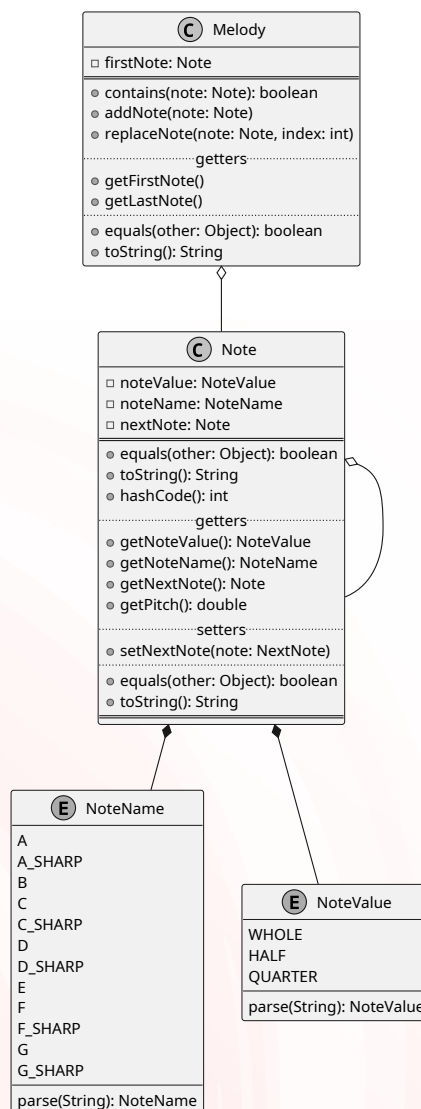
- Solange ein Test scheitert, ist *Ihre Implementierung nicht korrekt*. Betrachten Sie in diesem Fall die Fehlermeldung und den Quellcode des gescheiterten Tests. Für die Abgabe müssen *alle Testfälle* erfolgreich durchlaufen — der Übungsleiter wird Sie dazu auffordern, die Tests laufen zu lassen und die Abgabe *erst akzeptieren* wenn alle Tests *erfolgreich* sind!
- *Verändern Sie nicht die Inhalte der JUnit-Tests* um das Problem zu „lösen“!

Aufgabe: Kompositionssoftware

In dieser Aufgaben implementieren wir eine einfache *Kompositionssoftware um Melodien* zu erstellen. Dazu implementieren wir im Folgenden vier Klassen:

- Note ist eine Klasse, die eine gespielte Note repräsentiert.
- Melody dient zur *Verwaltung* von Noten als eine  einfach-verkettete Liste.
- Main beinhaltet das *Hauptprogramm* zur Bearbeitung von Melodien. Diese Klasse ist bereits für Sie vorbereitet und muss **nicht** von Ihnen ergänzt werden.
- NoteName ist ein Enum, welches verwendet wird, um Noten entsprechend des  Englischen Standards zu benennen. Der Einfachheit halber verzichten wir auf enharmonische Verwechslung, d.h. in unserem Programm existiert nur *SHARP* für die Benennung schwarzer Tasten. Unsere Noten heißen also A, A#, B, C, C# usw.
- NoteValue ist ein Enum welches die Länger einer Note beschreibt. Wir kennen drei Notenwerte: WHOLE, HALF und QUARTER.

Das folgende UML Klassendiagramm soll Ihnen zur Orientierung dienen:



Hauptprogramm

Die `main`-Methode befindet sich in der Klasse `Main`, die die User Interaktion bereits implementiert. Es ist empfehlenswert, dass Sie eine *weitere main-Klasse* erstellen in dessen `main`-Methode Sie Ihre bisherigen Implementierung testen.

Das Enum `NoteName` ★★

Noten werden mit Buchstaben bezeichnet. Wir verwenden das enum `NoteName` zur internen Repräsentation der Tonhöhe. Zusätzlich fügen wir das Attribut `abbreviation` hinzu, um den für Menschen verständlichen Noten Namen ausgeben zu können:

Konstante	abbreviation
A	"A"
A_SHARP	"A#"
B	"B"
C	"C"
C_SHARP	"C#"
D	"D"
D_SHARP	"D#"
E	"E"
F	"F"
F_SHARP	"F#"
G	"G"
G_SHARP	"G#"

Für die Benutzerinteraktion ist es nötig, eine statische `parse(String)`-Factory-Methode zu implementieren. Dies soll die Umkehrung der obigen Tabelle darstellen.

Tipp: Ein `enhanced switch` ist praktisch für die Implementierung.

Achtung: Gemeine User übergeben auch nicht definierte Eingabewerte in Ihre `parse(String)`-Methode. Reagieren Sie entsprechend auf undefinierte Eingaben.

Das enum `enum NoteValue` ★★

Noten haben eine bestimmte Länge, in unserer einfachen Kompositionssoftware beschränken wir uns auf *ganze*, *halbe* und *Viertelnoten*. Jeden dieser Notenwerte modellieren wir als Enum Konstante. Zusätzlich fügen wir das Attribut `abbreviation` hinzu, um den für Menschen verständlichen Noten Namen ausgeben zu können:

Konstante	abbreviation
WHOLE	"W"
HALF	"H"
QUARTER	"Q"

Für die Benutzerinteraktion ist es ebenfalls nötig, eine statische `parse(String)`-Factory-Methode zu implementieren. Dies soll die Umkehrung der obigen Tabelle darstellen.

Tipp: Ein `enhanced switch` ist praktisch für die Implementierung.

Achtung: Gemeine User übergeben auch nicht definierte Eingabewerte in Ihre `parse(String)`-Methode. Reagieren Sie entsprechend auf undefinierte Eingaben.

Die Klasse `Note` ★★

Die Klasse `Note` hat drei Felder:

- ▶ **name:** `NoteName`
- ▶ **value:** `NoteValue`
- ▶ **nextNote:** `Note`

Implementieren Sie folgendes:

- ▶ Einen Konstruktor mit den Parametern `name` vom Typ `NoteName` und `value` vom Typ `NoteValue`
- ▶ Einen Konstruktor mit den Parametern `name` vom Typ `String` und `value` vom Typ `String`. Dieser Konstruktor soll intern die statischen `parse(String)`-Methoden der Klassen `NoteValue` und `NoteName` verwenden. Dieser Konstruktor muss den ersten Konstruktor, den Sie implementiert haben, aufrufen.
- ▶ Getter für alle Felder
- ▶ Einen Setter für das Feld `nextNote`
- ▶ eine `equals()` Methode. Eine `Note` ist gleich, wenn `name` und `value` gleich sind.
- ▶ Die Methode `toString()`. Sie wandelt die `Note` in einen einfachen, menschenlesbaren `String` um, die Felder `name` und `value` sind mit einem Komma getrennt.
Beispiel: `C#:W`.

Tipp: Benutzen Sie die `getAbbreviation()` Methoden der Enums `NoteName` und `NoteValue`.

- ▶ Die Property Methode `getPitch()`. Diese berechnet die Frequenz der `Note` in Hertz, gemäß dieser Formel:

$$f = 2^{\frac{h}{12}} \cdot 440\text{Hz}$$

h ist der Abstand in Halbtönen zur Note A. Beispiel: A hat 0 Halbtöne Abstand zu A und C hat 3 Halbtöne Abstand zu A.

- ▶ Die Methode `toString()`. Sie wandelt die `Note` in einen einfache, menschenlesbaren `String` um. Nutzen Sie die Methoden `getAbbreviation()` der Klassen `NoteName` und `NoteValue`.

Die Klasse Melody ☆☆

Die Klasse Melodie enthält Noten und speichert diese in Form einer ↗ einfach-verkettete Liste. Hierfür besitzt jede Note das Feld nextNote. Die Aufgabe der Klasse Melody ist es, diese Melodie zu speichern und ggf. zu verändern. Die Klasse Melody hat ein Feld:

- ▶ **firstNote:** Note

Implementieren Sie folgendes:

- ▶ Einen Konstruktor mit den Parameter firstNote vom Typ Note
- ▶ Getter für alle Felder
- ▶ Die Methode contains(Note). Sie überprüft, ob die Melodie **exakt** dieses Notenobjekt enthält und gibt das Ergebnis als **boolean** zurück.
- ▶ Die Methode addNote(Note) fügt der Melodie eine Note am Ende hinzu. Zusätzlich überprüft diese Methode, ob die Melodie schon **exakt** dieses Notenobjekt enthält, ist dies der Fall muss eine Exception geworfen werden.
- ▶ Die Methode replaceNote(newNote, index): sie tauscht eine Note mit dem Index i aus. Der Wert des Parameters index ist auf Sinnhaftigkeit zu überprüfen; Hat index keinen sinnvollen Wert, ist eine Exception zu werfen. Zusätzlich überprüft diese Methode, ob die Melodie schon **exakt** dieses Notenobjekt enthält, ist dies der Fall muss eine Exception geworfen werden.
- ▶ Die Methode toString(). Sie wandelt die Melodie in einen einfachen, menschenlesbaren String um, die einzelnen Noten werden mit einem Komma getrennt.
Beispiel: A:W,B:W,C#:W,D#:H,F#:Q,C#:Q.
Eine leere Melodie (firstNote ist **null**) gibt einen leeren String zurück.
Tipp: Nutzen sie die toString() Methode der Klasse Note, die sie bereits implementiert haben.
- ▶ equals() Eine Melody ist dann gleiche einer anderen, wenn alle Noten gleich sind.