HSLU Hochschule Luzern

Objektorientierte Programmierung (OOP)

Übung SW08: Collections Einsatz / Enumerationen

Themen: Collections Einsatz (D02), Enumerationen, Static und Final (O10).
Zeitbedarf: ca. 240min.

Roland Gisler, Version 1.6.0 (FS25)

1 Schlüsselwörter `final` und `static`

1.1 Lernziele

- Schlüsselwörter **`final`** und **`static`** verwenden.
- Bedeutung von **`static`** für Attribute und Methoden kennen.
- Wirkung von **`final`** auf Variablen, Methoden und Klassen verstehen.

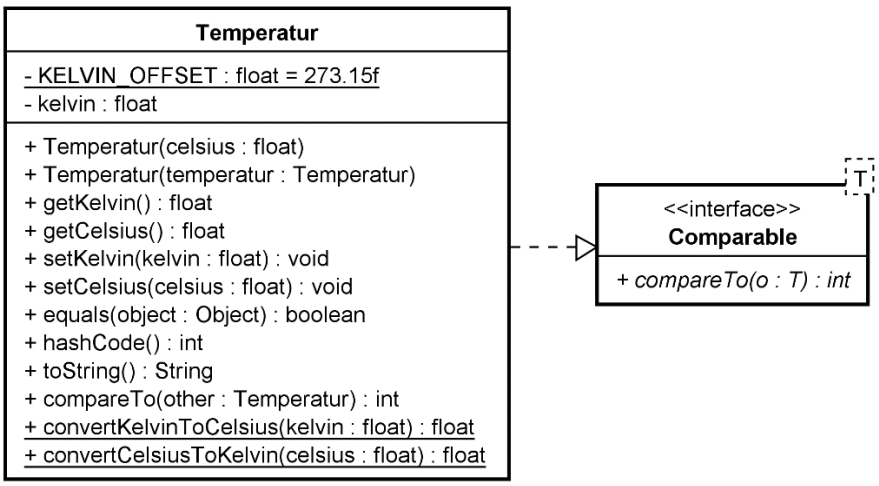
1.2 Grundlagen

Im Verlaufe der letzten Wochen haben wir immer wieder an der Klasse **`Temperatur`** gearbeitet und diese Stück für Stück erweitert. Mit dieser Aufgabe ergänzen wir einige der letzten Puzzlesteine, so dass wir die Klasse nun auf einen sehr soliden Stand bringen können!

1.3 Aufgaben

a.) Kopieren Sie den aktuell besten Stand Ihrer **`Temperatur`**-Klasse in das Package der aktuellen Semesterwoche. Vergessen Sie dabei nicht, auch die Testfälle zu kopieren und diese auszuführen (bzw. gegebenenfalls zu ergänzen). Bevor wir die Klasse weiterentwickeln, müssen alle Tests grün (also erfolgreich) sein - das ist sehr wichtig!

b.) Folgend sehen Sie ein mögliches Design der **`Temperatur`**-Klasse, so wie sie nach der Bearbeitung dieser Aufgabe aussehen sollte bzw. könnte:



Natürlich kann die Namensgebung in Details abweichen, das sind individuelle Freiheiten in der Modellierung. In den folgenden Teilaufgaben erhalten Sie Hinweise, welche Änderungen und Ergänzungen wir noch machen wollen.

c.) Wir rechnen innerhalb der Klasse zwischen Celsius und Kelvin um. Dafür benötigen wir einen Offsetwert von `273.15f`. Das ist ein idealer Kandidat für eine finale, statische Konstante, welche wir sinnvoll benennen und mit `static final` auszeichnen.

Hinweis: Aktuelle IDEs helfen Ihnen dabei, vorhandene literale Werte als Konstanten zu refaktorieren. Nutzen Sie diese Refactoring-Funktionen, denn sie vereinfachen und beschleunigen nicht nur die Arbeit, sondern reduzieren auch massgeblich die (Flüchtigkeits-)Fehler. Sie können es aber auch bewusst «von Hand» machen.

Nach diesem ersten Refactoring führen Sie wieder alle Testfälle aus, um zu prüfen ob nicht versehentlich etwas vergessen, oder ein Fehler eingebaut wurde!
➔ Die Unit-Tests übernehmen hier die Rolle von Regressionstests!

d.) Vielleicht haben Sie die (sehr einfache) Umrechnung zwischen Celsius und Kelvin an allen notwendigen Stellen direkt eingefügt? Im Sinne der Wiederverwendung ist es aber viel besser, diese als Methoden zu implementieren. Und weil diese beiden Methoden «stateless» (also nur von aktuellen Parametern, aber nicht von Attributen des Objektes abhängig) sind, können wir diese sogar als **statische**, öffentliche Hilfsfunktionen definieren!

e.) Ergänzen Sie, wenn nicht schon längst passiert, auch hier wieder entsprechende Testfälle für diese beiden Methoden und testen Sie sie, **bevor** Sie sie an anderen Codestellen aufrufen!

f.) Ersetzen Sie nun die allenfalls direkt implementierten Berechnungen konsequent durch Aufrufe der zwei neuen, statischen Funktionen. Auch hier können Sie – sofern Sie Testfälle haben – wieder deren unmittelbaren Nutzen erleben: Sie können nach der Veränderung des Quellcodes **sofort** und **automatisiert** testen, ob sich die Klasse trotz des Refactorings noch immer gleich verhält wie vorher!

g.) Es macht wenig Sinn, dass die Klasse `Temperatur` spezialisiert (vererbt) werden kann, oder? Zumindest haben wir sie nicht explizit dafür vorgesehen. Darum: Finalisieren Sie die Klasse! Wenn Sie wollen, erstellen Sie eine neue Klasse und probieren Sie aus was passiert, wenn Sie es trotzdem versuchen (`Dummy extends Temperatur`)?

h.) In einer `final`en Klasse sind implizit auch die Methoden `final`. Gehen Sie darum kurz zurück zur Übung mit `Shape` (abstract), `Circle` und `Rectangle`. Welche dieser Klassen können bzw. sollten Sie finalisieren und welche nicht? Überlegen Sie besonders bei `Shape` genau, welche Methoden in einer Spezialisierung **nicht** verändert werden sollten und finalisieren Sie diese!

Wir haben es fast geschafft: Prüfen Sie zum Abschluss auf der Klasse `Temperatur` noch einmal kritisch die Implementationen der Methoden `equals()`, `hashCode()` und `compareTo()` (Implementation des Interfaces `Comparable`). Sie sind für die weiteren Übungen eine wichtige Grundlage!
Machen Sie mit einer/m MitstudentIn (oder einem Dozenten) einen kurzen Review! Und wenn Sie auch `toString()` sinnvoll überschrieben haben, erspart Ihnen das später ebenfalls viel Arbeit bzw. erleichtert Ihnen die Fehlersuche.

2 Einsatz von Collections

2.1 Lernziele

- Auswahl einer geeigneten Datenstruktur (Collection).
- Verwenden von Datenstrukturen mit polymorphen Typen.
- Nutzen von vorhandenen Funktionen zur Iteration und Bearbeitung von Collections.

2.2 Grundlagen

Achten Sie darauf, dass Sie bei jeder Collection die Sie verwenden (z.B. **`ArrayList`**), für die Variable einen möglichst allgemeinen (Interface-)Typ (z.B. **`List`**, oder evt. sogar **`Collection`**) verwenden! Das ist angewandte Polymorphie und erlaubt es Ihnen, später mit minimalen Codeänderungen die Implementation einfach austauschen zu können.

2.3 Aufgaben

a.) Entwerfen Sie eine Klasse **`TemperaturVerlauf`**. Skizzieren Sie dazu ein einfaches UML-Klassendiagramm! Die Anforderungen sind: Die Klasse soll eine Menge von **`Temperatur`**-Objekten in einer Collection speichern zu können. Temperaturen sollen mit einer **`add(…)`**-Funktion hinzugefügt werden können. Eine Methode **`clear()`** soll alle enthaltenen Temperaturen entfernen und mit **`getCount()`** will man abfragen können, wie viele Temperaturwerte aktuell enthalten sind. Welche Datenstruktur ist dazu gut geeignet?

b.) Implementieren Sie die Klasse **`TemperaturVerlauf`**, und testen Sie diese gewohnt mit Unit-Tests! Wenn Sie für die Klasse zuerst ein Interface definiert haben: Super! Sehr gut!

c.) Nun möchten wir eine Methode ergänzen, welche uns den Maximalwert aller enthaltenen Temperaturen liefert. Implementieren und testen Sie sie! Hinweis: Es gibt unterschiedliche Varianten dies zu bewerkstelligen! Überlegen Sie sich die verschiedenen Varianten und tauschen Sie sich mit anderen Studierenden (oder einem Dozenten) aus. Vergessen Sie dabei nicht die mächtigen Methoden der Klasse `Collections`!

d.) Wir möchten auch den Minimalwert aller Temperaturen wissen. Implementieren Sie!

e.) Eine weitere neue Anforderung: Wir möchten die Durchschnittstemperatur aller enthaltenen Werte berechnen. Ergänzen Sie auch diese Funktion in Ihrer Klasse **`TemperaturVerlauf`** und testen Sie mit einfachen Werten. Vorsicht: Achten Sie darauf, dass Sie die Implementation möglichst unabhängig von der (intern) verwendeten Datenstruktur vornehmen (Stichwort: **`Iterator`**)!

f.) Probieren Sie einfach mal aus, was es bedeutet, die interne Datenstruktur auszutauschen! Wir könnten ja (ungeachtet dessen, ob in diesem Fall sinnvoll oder nicht) verlangen, dass keine doppelten Temperaturwerte enthalten sein dürfen!

g.) Haben Sie auch getestet was passiert, wenn die Methoden von c) bis e) aufgerufen werden, **ohne** dass überhaupt Werte enthalten sind?
Wenn nein: Implementieren Sie nun **zuerst** die Testfälle, welche diese Fehler aufdecken und das dann erwartete Verhalten überprüfen! Erst danach korrigieren Sie Ihre Fehler!
Wenn ja: **Sehr gut!** Sie sind auf dem besten Weg ein(e) umsichtige(r) Entwickler*in zu werden!

3 Enumerationen

3.1 Lernziele

- Potential der Enumerationen verstehen.
- Implementation einfacher Enumerationen.
- Enumerationen mit zusätzlichen Attributen und Methoden versehen.

3.2 Grundlagen

Für diese Aufgaben greifen wir wieder auf unsere chemischen Elemente zurück. Wir haben uns für diese ja jeweils die Schmelz- und die Siedetemperatur gemerkt. Und damit sind indirekt natürlich auch die Aggregatszustände (**`SOLID`**, **`LIQUID`**, **`GAS`**)[1] dieser Elemente (für eine bestimmte Temperatur) definiert.

3.3 Aufgaben

a.) Entwerfen Sie eine Enumeration für die drei Aggregatszustände und setzen Sie diese um.

b.) Ergänzen Sie Ihren Entwurf für die Elemente an geeigneter Stelle mit Methoden, damit wir den Aggregatszustand eines Elementes abhängig von einer aktuellen Temperatur abfragen können!

c.) Verwenden Sie die Enumeration, um nach einer Abfrage des Zustandes damit eine Ausgabe in der Art `"Blei ist bei 20°C fest."` (als Beispiel) erzeugen zu können.
Tipp: Beachten Sie, dass wir Enumerationen auch als Datentyp in einem **`switch`**-Statement verwenden können!

d.) Ergänzen Sie die Enumeration mit einem **`String`**-Attribut (und die damit notwendigen restlichen Methoden), welches «schöne» Bezeichnung (wie «fest», «flüssig» und «gasförmig») ergänzt.

e.) Die Teilaufgabe c) können wir mit der Erweiterung von d) nun schon wieder ganz anders (viel eleganter und mit weniger Code) lösen, oder? Probieren Sie es aus!

Optional:

f.) Es gibt eine spezielle Implementation einer **`Map`**: Die sogenannte **`EnumMap`**. Sie verwendet als Key eine Enumeration. Man könnte die verschiedenen Temperaturen für Schmelz- und Siedepunkt als Enum modellieren, und diese wiederum in der **`Element`**-Klasse in einer entsprechenden **`EnumMap`** ablegen. Wenn Sie Lust haben: Probieren Sie es doch mal aus!

Hinweis: Hier exemplarisch und beispielhaft zu Übungszwecken, lässt sich diese Teilaufgabe f) sehr kontrovers diskutieren. Sie darf auf keinen Fall als mustergültig betrachtet werden, sondern nur als eine mögliche Variante!

➔ Objektorientiertes Design ist keine absolute Wissenschaft, sondern hat viel mit Erfahrung, Vernunft und den individuell unterschiedlichen Anforderungen im jeweiligen Kontext zu tun.

[1] Den vierten Aggregatszustand **`PLASMA`** lassen wir hier zur Vereinfachung einfach mal weg, weil ionisieren kann man ja nicht nur thermisch. ☺