

Programmieren II (Java)

5. Praktikum: Collections und Iteratoren





Sommersemester 2024

Christopher Auer



Abgabetermine

Lernziele

- ▶  Iterator und  Iterable
- ▶ *Java-Collections*: erstellen, befüllen, bearbeiten und abfragen
- ▶  Comparator und  Comparable

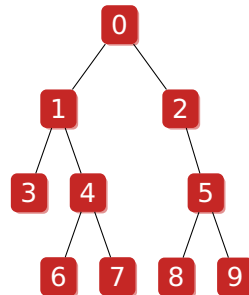
Hinweise

- ▶ 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*!



Aufgabe 1: Breitensuche als Iterator ★★

Sie kennen aus dem ersten Semester die Datenstruktur Binärbaum: Ein Elternknoten (*Vertex*) eines Binärbaums hat entweder *kein Kind*, ein *linkes Kind*, ein *rechtes Kind* oder *beide Kinder*. Der Knoten ohne Eltern heisst *Wurzel*, ein Knoten ohne Kinder heisst *Blatt*. Ein Beispiel eines Binärbaums, mit der Wurzel ganz oben, ist:



Zusätzlich kann ein Knoten einen Wert besitzen. In unserem Fall ist das ein `int`. Die Klasse `Vertex` modelliert einen Knoten in einem Binärbaum:

Vertex
- value : <code>int</code> - left : <code>Vertex</code> - right : <code>Vertex</code>
+ Vertex(value : <code>int</code> , left : <code>Vertex</code> , right : <code>Vertex</code>) + getValue(): <code>int</code> + getLeft/Right(): <code>Vertex</code>

<<interface>> Vertex
Iterable

Eine Breitensuchen-Durchlauf listet die Knoten eines Binärbaums *ebenenweise* von *oben nach unten* und je Ebene von *links nach rechts* auf. Im obigen Beispiel ergibt so ein Durchlauf die Reihenfolge 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Eine *Breitensuche* lässt sich mit einer Queue implementieren:

1. Die Queue wird mit dem *Wurzelknoten* initialisiert.
2. *Solange* die Queue *noch nicht leer ist*, wiederhole:
 - Entferne das *erste Element* der Queue. Dieser Knoten *v* ist der aktuelle Knoten des Durchlaufs.
 - Füge das *linke* und *rechte Kind* von *v*, falls vorhanden, hinten an die Queue an.

Im Folgenden machen wir die Klasse `Vertex` *iterierbar*, wobei der *Iterator* die Elemente, beginnend von der Wurzel, in der Reihenfolge der Breitensuche zurückgibt.

Das Interface `Iterable`

Betrachten Sie die bisherigen Implementierung der Klasse `Vertex` und ändern Sie die Klassendeklaration so ab, dass `Vertex` das Interface `Iterable` *implementiert*, wobei der *Typparameter* `Vertex` ist.

- Was sagt die *Fehlermeldung* aus? Welche Möglichkeiten haben Sie *prinzipiell* die Fehlermeldung zu beheben?

- ▶ **Überschreiben** Sie die Methode `iterator` aus dem Interface `Iterable`, indem Sie `null` zurückgeben!

Das Interface `Iterator`

Implementieren Sie eine *private, innere* (*nicht* statische) Klasse `VertexIterator` mit:

- ▶ `VertexIterator` implementiert `Iterator` mit `Vertex` als *Typparameter*.
- ▶ Als Attribut besitzt `VertexIterator` die Queue für die *Breitensuche*. Verwenden Sie dafür eine `LinkedList`.
- ▶ Der Konstruktor von `VertexIterator` erstellt die Queue und befüllt sie mit der Wurzel. *Hinweis:* Die Wurzel ist die `Vertex`-Instanz, auf der `iterator` aufgerufen wurde.
- ▶ `hasNext` liefert `true`, wenn die Queue noch Elemente besitzt, sonst `false`.
- ▶ `next` implementiert die *Breitensuche* (s. oben). Dabei gibt `next` den Knoten, der aus der Queue entfernt wird, zurück und hängt ggf. dessen Kinder an die Queue an.
- ▶ Achten Sie darauf, dass `next` sich korrekt nach Spezifikation verhält, wenn `hasNext` `false` liefert.
- ▶ Geben Sie in `Vertex.iterator` eine Instanz von `VertexIterator` zurück.

Testen

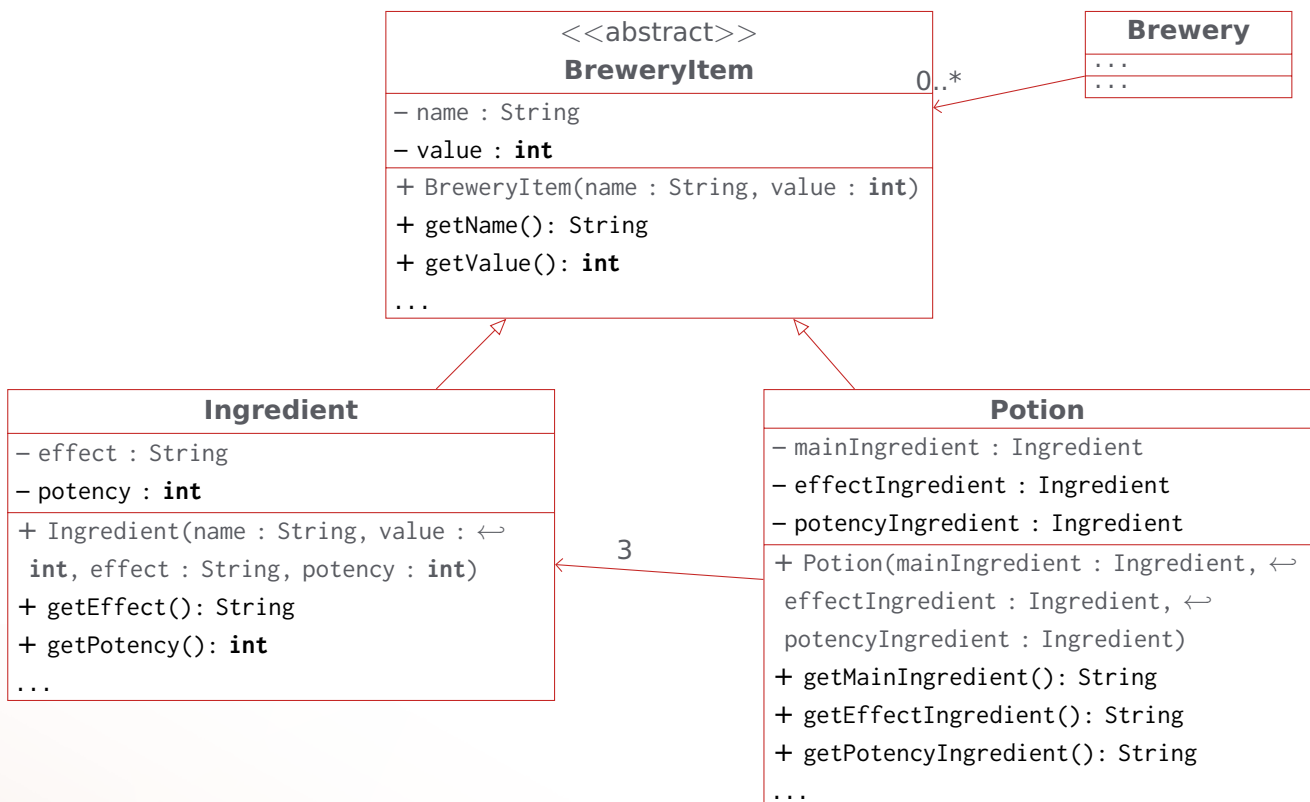
Testen Sie Ihre Implementierung:

- ▶ Führen Sie den JUnit-Test in `VertexTest` aus.
- ▶ `BfsMain.main` definiert den Binärbaum, aus obigem Beispiel. Dafür müssen Sie den Bereich mit `Vertex` *einkommentieren*. Ergänzen Sie die `main`-Methode um eine *foreach*-Schleife, die, beginnend bei `root`, alle Knoten des Binärbaums durchläuft und jeweils deren *Werte* ausgibt (`getValue`).



Aufgabe 2: Zaubertrank-Brauerei ★★

In diesem Praktikum verwenden wir die Java-Collections um eine *Zaubertrank-Brauerei* zu verwalten: In einer Zaubertrank-Brauerei kann man *Zutaten* (Ingredients) zusammenmischen um *Zaubertränke* (Potions) mit verschiedenen Eigenschaften herzustellen. Dazu sind bereits drei Klassen implementiert:



- ▶ **BreweryItem** — die *abstrakte Oberklasse* für alle Dinge, die in unserer Brauerei verwaltet werden. Ein **BreweryItem** hat einen *Namen* (name) und einen *Wert* (value).
- ▶ **Ingredient** — eine *Zutat* hat zusätzlich einen *Effekt* (effect) und eine *Potenz* (potency), die angibt wie stark die Zutat die Wirkung des Tranks erhöht. Beispiele für Zutaten sind:
 - ▶ name = "Ice Cube", value=1, effect = "Cooling", potency = 1
 - ▶ name = "Bird's Eye Chili", value=2, effect = "Burning", potency = 4
 - ▶ name = "Unicorn Hair", value=10, effect = "Rejuvenating", potency = 15
- ▶ **Potion** — ein Zaubertrank. Wie Sie aus der Grundlagenvorlesung „*Zaubertränke I*“ aus dem letzten Semester sicherlich noch wissen, besteht jeder Zaubertrank aus drei Zutaten (Ingredients):
 - ▶ **mainIngredient** — die „*Hauptzutat*“ aus der der Zaubertrank besteht.
 - ▶ **effectIngredient** — Zutat, die die Art der *Wirkung* definiert.
 - ▶ **potencyIngredient** — Zutat, die die *Potenz* (Wirkungsstärke) des Tranks bestimmt.

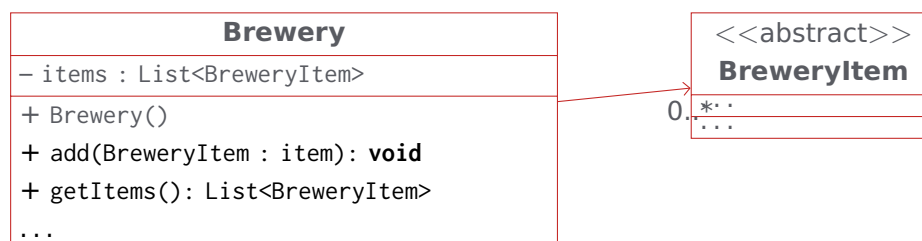
Der Wert des Zaubertranks wird dabei aus dem Wert von **mainIngredient** plus den multiplizierten Werten von **effectIngredient** und **potencyIngredient** berechnet (bereits in **Potion.java** *implementiert!*). Verwendet man z.B. "Ice Cube" als **mainIngredient**, "Bird's Eye Chili" für die Wirkung und "Unicorn hair" für die Potenz, so erhält man als Resultat "Burning Ice Cube" mit einer Potenz von 15 und einem Wert von $1 + 4 \cdot 15 = 61$.

Vertrautmachen mit bestehendem Code

Öffnen Sie das Verzeichnis `SupportMaterial/potion-brewery` in Ihrer IDE und machen Sie sich mit den bestehenden Klassen vertrauen! Die Klassen `BreweryItem`, `Ingredient` und `Potion` sind, wie oben beschrieben, bereits implementiert. Die Klasse `BreweryMain` beinhaltet ein Grundgerüst für ein `main`-Programm, das wir im Folgenden erweitern. Die Tests liegen im Verzeichnis `app/src/test`.

Die Klasse `Brewery` ★★

Mit der Klasse `Brewery` wollen wir unsere Zutaten und Zaubersprüche verwalten, sowie natürlich brauen.



Das Attribut `items` verweist auf eine `List`-Collection, in der alle *Zutaten* und *Zaubersprüche* verwaltet werden. Die Datenstruktur soll dabei *Duplikate* zulassen und *effizientes Erweitern* am Ende der Liste erlauben. Deklarieren Sie die Klasse `Brewery` mit folgendem Inhalt:

- ▶ Das Attribut `items`
- ▶ Der *Default-Konstruktor* initialisiert `items` mit einer geeigneten Java-Collection.
- ▶ Die Methode `add(BreweryItem item)` (`item != null`) fügt ein Element an `items` an.
- ▶ Die Methode `getItems()` gibt `items` als *unveränderliche Liste* zurück.

Testen Sie Ihre Implementierung mit den Testmethoden `testConstructor`, `testAdd` und `testGetItems` in `BreweryTest`!

`BreweryMain` ★★

Die Klasse `BreweryMain` enthält die `main`-Methode. Kommentieren Sie den Bereich mit der Markierung `### Brewery` ein. Erstellen Sie in der `main`-Methode eine `Brewery`-Instanz mit dem Namen `brewery` und rufen Sie damit die Methode `fillWithIngredients` auf (`TODO fillWithIngredients`). Diese befüllt Ihre Brauerei mit zahlreichen *Zutaten*. Implementieren Sie dann eine *statische private Methode* `void printBrewery(Brewery brewery)`, die die Elemente Ihrer Brauerei zeilenweise ausgibt, und rufen Sie die Methode auf (`TODO printBrewery`). *Hinweis*: `BreweryItem` und `Co.` implementieren bereits `toString()`.

`Brewery.contains(BreweryItem item)` ★★

Implementieren Sie die Methode `boolean contains(BreweryItem item)` (`item != null`), die ermittelt ob `item` in der Brauerei vorhanden ist! Erweitern Sie die `main`-Methode wie folgt (`TODO contains`):

- ▶ Prüfen Sie ob `unicornHair` in der Brauerei vorhanden ist.
- ▶ Erstellen Sie `Ingredient`-Instanz kugelschreibbar mit `name="Kugelschreibbar"`, `value=1`, `effect="Screaming"` und `potency=1`. Prüfen mit `contains` Sie ob kugelschreibbar in `brewery` enthalten ist und geben Sie das Ergebnis aus.

- ▶ Erstellen Sie `unicornHairClone`, das `unicornHair` *gleich* (`unicornHair.equals(unicornHairClone) == true`), und prüfen Sie mit `contains` ob `unicornHairClone` in `brewery` enthalten ist. Geben Sie das Ergebnis aus. Was ist das Ergebnis und *warum*? Wie würde sich das Verhalten ändern, wenn weder `Ingredient` noch `BreweryItem` die `equals`-Methode implementieren würden?

Testen Sie `contains` mit `testContains`!

Brewery.remove und BreweryException ★★

Auch in einer Zaubertrank-Brauerei geht mal was schief. Implementieren Sie eine *ungeprüfte Ausnahmeklasse* `BreweryException` mit den in der Vorlesung vorgestellten *Konstruktoren*!

Implementieren Sie dann die Methode `void Brewery.remove(BreweryItem item)` (`item != null`) wie folgt:

- ▶ Sollte `item` nicht in der Brauerei vorhanden sein, generieren Sie eine `BreweryException` mit entsprechender *aussagekräftiger* Fehlermeldung.
- ▶ Ansonsten, entfernen Sie `item` aus der Brauerei.

Macht es Sinn, dass `BreweryException` *ungeprüft* ist? Wenn ja, warum? Wenn nein, warum *nicht*? Erweitern Sie `main` wie folgt (TODO `remove`):

- ▶ Entfernen Sie `unicornHair` aus der Brauerei. *Welche Instanz* wurde aus der Brauerei entfernt?
- ▶ Versuchen Sie kugelschreibbar aus der Brauerei zu entfernen. Fangen Sie die dabei generierte `BreweryException` mit einem `try-catch`-Block.

Testen Sie `remove` mit `testRemove`!

Zaubertränke brauen mit brew() ★★

Implementieren Sie die Methode `Potion brew(Ingredient mainIngredient, Ingredient effectIngredient, Ingredient potencyIngredient)` (`Parameter != null`), die einen Zaubertrank wie folgt braut:

- ▶ Prüfen Sie für jede Zutat, ob Sie auch wirklich *vorhanden* ist. Wenn nicht, generieren Sie eine `BreweryException` mit entsprechender *aussagekräftiger* Fehlermeldung.
- ▶ *Entfernen* Sie die drei Zutaten aus der Brauerei.
- ▶ *Erstellen* Sie den Trank, indem Sie den Konstruktor von `Potion` mit den drei Parametern aufrufen.
- ▶ *Fügen* Sie den Trank in die Brauerei ein und geben Sie in anschließend zurück.

Erweitern Sie `main` so, dass Sie folgende Zaubertränke brauen und die Ergebnisse jeweils ausgeben:

mainIngredient	effectIngredient	potencyIngredient
booger	capri	birdsEyeChili
northpoleSnowflake	unicornHair	capri
springWater	unmatchedOpeningParenthesis	springWater
booger	capri	birdsEyeChili

Geben Sie die den Bestand der Brauerei nach dem Brauen aus. Testen Sie `brew` mit `testBrew`!

sortedItems und Comparable ★★

Die Klasse `BreweryItem` soll eine *natürliche Ordnung* implementieren. Implementieren Sie dazu das Interface `Comparable<BreweryItem>` in `BreweryItem`, wobei zuerst *aufsteigend* nach dem *Namen* und dann nach dem *Wert* verglichen werden soll. Implementieren Sie dann die Methode `List<BreweryItem> Brewery.sortedItems()`, die die Elemente in der Brauerei *aufsteigend nach der natürlichen Ordnung* zurückgibt. Geben das Ergebnis von `sortedItem` in `main` aus und testen Sie Ihre Implementierung mit `testSortedItems`!

getIngredients ★★

Implementieren Sie die Methode `List<Ingredient> Brewery.getIngredients()`, die eine Liste mit allen `Ingredient`-Instanzen zurückgibt! Testen Sie `getIngredients` mit `testGetIngredients`!

sortedIngredients ★★→★★

Implementieren Sie die Methode `List<Ingredient> sortedIngredients()`, die eine *sortierte Liste* aller `Ingredient`-Instanzen zurückgibt! Verwenden Sie für die Sortierung einen `Comparator`, den Sie als *private geschachtelte Klasse* (**static**) wie folgt implementieren: Zwei `Ingredient`-Instanzen werden zuerst nach dem *Effekt*, dann nach der *Potenz* und schließlich nach der *natürlichen Ordnung* von `BreweryItem` aufsteigend verglichen. Verwenden Sie die Methode `getIngredients` der vorherigen Aufgabe. Geben Sie das Ergebnis von `sortedIngredients` in `main` aus (TODO `← sortedIngredients`)! Testen Sie `sortedIngredients` mit `testSortedIngredients`! Wie sähe die Implementierung aus, wenn Sie anstatt einer *privaten geschachtelten Klasse* eine *anonyme Klasse* verwenden würden?

getTotalValue ★★

Implementieren Sie eine Methode `int getTotalValue()`, die den *Gesamtwert* aller Elemente unserer Brauerei berechnet, d.h. die Summe aller Werte von `BreweryItem.getValue()`. Geben Sie das Ergebnis in `main` aus (TODO `getTotalValue`) und testen Sie Ihre Implementierung mit `testGetTotalValue`!

getMostValuablePotion ★★→★★

Implementieren Sie eine Methode `Potion getMostValuablePotion()`, die den *wertvollsten Zaubertrank* ermittelt und zurückgibt, d.h. `getValue()` liefert den größten Wert. Sollten keine Zaubertränke vorhanden sein, so soll `null` zurückgegeben werden! Geben Sie den wertvollsten Trank in `main` aus (TODO `getMostValuablePotion`) und testen Sie Ihre Implementierung mit `testGetMostValuablePotion`!

getInventory ★★→★★

Es fehlt noch eine Methode: `Map<BreweryItem, Long> getInventory()` liefert eine *Zuordnung* der *unterschiedlichen* Elemente unserer Brauerei zu der Anzahl, wie oft Sie vorkommen. So kommt z.B. `squealingChalk` *dreimal* vor, während der Zaubertrank „Infuriating Spring Water“ nur *einmal* vorkommt. Implementieren Sie `getInventory` und geben Sie die Zuordnung in `main` wie folgt aus (TODO `getInventory`):

```
3 -> Ingredient: Ice Cube, value=1, effect=Cooling, potency=1
1 -> Ingredient: Unicorn Hair, value=10, effect=Rejuvenating, potency=15
3 -> Ingredient: Dragon's Breath, value=8, effect=Burning, potency=9
1 -> Potion: Infuriating Spring Water (2), value=6
...
```

Testen Sie Ihre Implementierung mit `testGetInventory!`