Abschnitt 17: Beispiel: Die Klasse String (Teil 2), das Interface Comparable<T>

Überblick

Zur Erinnerung: Eigenschaften von Relationen

Eine Relation $R \subseteq A \times A$ ist

- *reflexiv*, wenn für alle $a \in A$ gilt: $R(a, a)$
- *symmetrisch*, wenn für alle $a, b \in A$ gilt: $R(a, b) \Rightarrow R(b, a)$
- *transitiv*, wenn für alle $a, b, c \in A$ gilt: $R(a, b) \wedge R(b, c) \Rightarrow R(a, c)$.
- *antisymmetrisch*, wenn für alle $a, b \in A$ gilt: $R(a, b) \wedge R(b, a) \Rightarrow a = b$.
- *total*, wenn für alle $a, b \in A$ gilt: $R(a, b) \vee R(b, a)$.

Exakter Vergleich von Objekten

- Wie wir gesehen haben, ist in Java die Gleichheit (`==`) von Objekten als Überprüfung der Identität der Speicheradressen definiert.
- Die Klasse `Object` definiert aber auch eine Methode `public boolean equals(Object obj)`.
- Für Objekte vom Typ `Object` gibt `x.equals(y)` den gleichen Wert zurück wie der Vergleich `x==y`.
- Da jede Klasse eine Unterklasse von `Object` ist, verfügt auch jedes Objekt in Java über die `equals`-Methode.
- Oft wird diese Methode überschrieben, um eine allgemeinere Gleichheit zu testen.
- Wie bereits diskutiert, unterscheidet man zwischen `==` und `equals` auch als Identität vs. Gleichheit von Objekten.

Gleichheit von Objekten

Die Methode `equals` ist in der Klasse `Object` wie folgt spezifiziert:
Die `equals`-Methode implementiert eine Äquivalenz-Relation auf nicht-**`null`** Objektreferenzen mit folgenden Eigenschaften:

- *Reflexivität:* Für jede nicht-**`null`** Referenz `x` gilt: `x.equals(x)`.
- *Symmetrie:* Für nicht-**`null`** Referenzen `x` und `y` ist `x.equals(y)` **`true`** g.d.w. `y.equals(x)` **`true`** ist.
- *Transitivität:* Für nicht-**`null`** Referenzen `x`, `y` und `z` gilt: wenn `x.equals(y)` und `y.equals(z)` **`true`** sind, dann ist auch `x.equals(z)` **`true`**.
- *Konsistenz:* Für nicht-**`null`** Referenzen `x` und `y` sind mehrfache Aufrufe von `x.equals(y)` entweder immer **`true`** oder immer **`false`**, solange keine Information in den Objekten `x` oder `y` verändert wurde, die von der Methode `equals` überprüft wird.
- Außerdem gilt für eine nicht-**`null`** Referenz `x`: `x.equals(`**`null`**`)` ergibt **`false`**.

Gleichheit von Objekten

- Diese Vorschrift *soll* eingehalten werden, wenn man in einer Klasse die Methode `equals` überschreibt. Das ist aber eine *semantische* Vorschrift, die nicht vom Compiler überprüft, sondern nur vom Programmierer bewiesen werden kann.
- Die Implementierung der Methode `equals` in der Klasse `Object` selbst ist die schärfste Möglichkeit, diese Vorschrift umzusetzen.

Exakter Vergleich von Zeichenketten

- Die Klasse String überschreibt `equals` so, dass für einen String `s` und ein Objekt `o` gilt:
  `s.equals(o)` g.d.w.:
  - `o` ist nicht **`null`**
  - `o` ist vom Typ String und
  - `o` repräsentiert genau die gleiche Zeichenkette wie `s` (d.h. `s` und `o` haben gleiche Länge und an jeder Stelle steht der gleiche Character.
- Genügt die Implementierung von `equals` in der Klasse `String` der Vorschrift für die Implementierung, die in der Klasse `Object` gegeben ist?

Abgeschwächte Gleichheit von Zeichenketten

- Eine abgeschwächte Gleichheit auf Strings achtet nicht auf den Unterschied zwischen Klein- und Großbuchstaben:
  `public boolean` `equalsIgnoreCase(String anotherString).`
- Gleichheit am Anfang bzw. am Ende:
  `public boolean` `startsWith(String prefix)` bzw.
  `public boolean` `endsWith(String suffix)`
- Gleichheit des Inhalts mit anderen Objekten vom Typ `CharSequence` (also z.B. ein `StringBuilder`, ein `StringBuffer` oder auch ein anderer `String`):
  `public boolean` `contentEquals(CharSequence cs)`

Überblick

Natürliche Ordnung auf Objekten

- `equals` implementiert eine *Äquivalenzrelation*, wie wir gesehen haben, d.h. eine Relation, die reflexiv, symmetrisch und transitiv ist.
- Eine *(partielle) Ordnung* ist eine Relation, die reflexiv, transitiv und antisymmetrisch ist.
- Eine *totale Ordnung* ist eine partielle Ordnung, die außerdem total ist.
- Das Interface `Comparable<T>` definiert eine *totale Ordnung* R auf der Klasse T durch die Methode `int compareTo(T o)` wie folgt:

$$R \subseteq T \times T \quad \text{mit} \quad (x, y) \in R \iff \texttt{x.compareTo(y) <= 0}$$

- Wenn eine Klasse das Interface `Comparable` implementiert, nennt man diese durch `compareTo` definierte totale Ordnung auch die *natürliche Ordnung* (*natural ordering*) der entsprechenden Klasse.

Natürliche Ordnung auf Objekten

Für zwei Objekte `x` und `y` vom Typ `T implements Comparable<T>` gilt:

- Der Ausdruck `x.compareTo(y)` ist vom Typ **`int`**.
- Der Wert des Ausdrucks `x.compareTo(y)` soll kleiner 0 sein, wenn `x` "kleiner" ist als `y`.
- Der Wert des Ausdrucks `x.compareTo(y)` soll größer 0 sein, wenn `x` "größer" ist als `y`.
- Der Wert des Ausdrucks `x.compareTo(y)` soll gleich 0 sein, wenn `x` "gleich" ist zu `y`.
- Diese "Gleichheit" (`x.compareTo(y)==0`) soll normalerweise die gleiche Gleichheit sein, wie die von `x.equals(y)` angegebene, dies ist aber nicht notwendigerweise so.
- **Achtung:** Falls eine Klasse `Comparable` so implementiert, dass `x.compareTo(y)==0` $\not\Leftrightarrow$ `x.equals(y)`,
  sollte in der Dokumentation deutlich darauf hingewiesen werden. Die empfohlene Bemerkung ist: *Note: this class has a natural ordering that is inconsistent with equals.*

Überblick

Lexikographischer Vergleich von Zeichenketten

- Die Klasse `String` implementiert `Comparable<String>`.
- Als natürliche Ordnung über `String`s wird dabei die lexikographische Ordnung angenommen.
- Grundlage des Vergleichs ist der `<`-Operator, angewendet auf die einzelnen **`char`**s der beiden Strings.
- Wenn zwei `String`s, `s1` und `s2`, an einem Index oder mehreren Indices aus unterschiedlichen Charactern bestehen, dann ist der Wert von `s1.compareTo(s2)` der Wert von `s1.charAt(k)-s2.charAt(k)`, wobei *k* der kleinste der unterschiedlichen Indices ist.
- Unterscheiden sie sich an keiner Stelle, dann ist der Wert von `s1.compareTo(s2)` der Wert von `s1.length()-s2.length()`.

Lexikographischer Vergleich von Zeichenketten

- Auch für den lexikographischen Vergleich gibt es die Möglichkeit, große und kleine Buchstaben als gleich anzusehen.
- **`public int`** `compareToIgnoreCase(String str)` vergleicht zwei Strings wie `compareTo`, nachdem auf jeden Character `character` die Anweisung `Character.toLowerCase(Character.toUpperCase(character))` angewendet wurde.

Überblick

Vergleich von Mustern

- Um ein Muster in einer Zeichenkette zu suchen, gibt es in der Klasse `String` z.B. die Methode
  `public String replaceAll(String regex, String replacement).`
- Der Parameter `regex` definiert dabei einen sogenannten *regulären Ausdruck*.
- Eine andere hilfreiche Methode, die auf regulären Ausdrücken basiert, ist `public String[] split(String regex).`

Exkurs: Reguläre Ausdrücke

- Reguläre Ausdrücke sind ein mächtiges Werkzeug, um Muster in Zeichenketten zu beschreiben.
- Reguläre Ausdrücke sind auch von theoretischer Bedeutung für die Beschreibung bestimmter Typen von Sprachen.
- Als Werkzeug in konkreten Programmierproblemen sind reguläre Ausdrücke auch von immenser praktischer Bedeutung. In dieser Hinsicht werden sie aber im Studium kaum vertieft.
- Wir geben an dieser Stelle ein paar einführende Hinweise.

Exkurs: Reguläre Ausdrücke

Definition

Reguläre Ausdrücke sind wie folgt für ein Alphabet Σ (d.h. eine Menge von Symbolen, also z.B. alle möglichen Werte vom Typ `char`) wie folgt *induktiv* definiert:

- $\emptyset$ (= die leere Menge) ist ein regulärer Ausdruck.
- ε (= das leere Wort) ist ein regulärer Ausdruck.
- Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck.
- Wenn a und b reguläre Ausdrücke sind, dann sind auch
  - ab (Konkatenation)
  - $a|b$ (Vereinigung)
  - $(a)^*$ (Stern-Operator), d.h. die durch den regulären Ausdruck a definierte Zeichenkette kann beliebig oft (auch 0-mal) hintereinander vorkommen.

  reguläre Ausdrücke.

Konstrukte für reguläre Ausdrücke

Viele Programmiersprachen bieten Konstrukte an, um reguläre Ausdrücke zu definieren, die dann verwendet werden können, um spezielle Zeichenketten innerhalb von Zeichenketten zu finden. In Java gibt es z.B.:

- `X` für das Zeichen `X`.
- `[abc]` für die Zeichen `a`, `b` oder `c`. Mit den eckigen Klammern wird eine sogenannte *Zeichenklasse* definiert, entspricht der Vereinigung. Möglich ist z.B. auch `[a-z]` für alle Kleinbuchstaben.
- Abkürzungen für bestimmte Zeichenklassen, z.B. `.` für ein beliebiges Zeichen, `\d` für eine Ziffer (= `[0-9]`).
- Quantifier, die beschreiben, wie oft eine Zeichenkette hintereinander auftreten darf, z.B.:
  - `X*` `X` kommt beliebig oft vor (0-mal oder öfter)
  - `X+` `X` kommt einmal oder öfter vor (= `X(X)*`)
  - `X{n}` `X` kommt genau *n*-mal vor
  - `X{n,m}` `X` kommt mindestens *n*-mal, höchstens *m*-mal vor
  - `X?` `X` kommt 0 oder 1-mal vor

Exkurs: Reguläre Ausdrücke

Pattern bilden

- Ein Konstrukt, das einen regulären Ausdruck beschreibt, kann man in einem String bilden.
- Aus einem solchen String kann man ein Objekt der Klasse `Pattern` bilden (durch die statischen Methoden `Pattern.compile`).
- In der Klasse `Pattern` sind die Möglichkeiten für reguläre Ausdrücke in Java ausführlich dokumentiert.
- Worauf matchen folgende Pattern?
  - `Pattern.compile("Kr(i|o)ege(l|r)")`
  - `Pattern.compile("Info (I{1,3}|IV)")`

Pattern Matching

- Aus einem `Pattern`-Objekt `p`, das durch den String `regex` (Darstellung eines regulären Ausdrucks) definiert wurde, kann man ein `Matcher`-Objekt `mStr` spezifisch für einen String `str` erzeugen, um das Pattern `p` in `str` zu finden, zu ersetzen usw.
- Nun können Sie das Verhalten der String-Methode
  `public String replaceAll(String regex, String replacement)`
  verstehen, die eine Abkürzung für folgende Befehle ist:

```
Pattern p = Pattern.compile(regex);
Matcher mStr = p.matcher(str);
String neuerString = mStr.replaceAll(replacement);
// oder auch:
String neuerString2 =
  Pattern.compile(regex).matcher(str)
         .replaceAll(replacement);
```

- Damit ist nicht-exakte Suche mit Mustern in Strings möglich, das sogenannte *Pattern Matching*.

Überblick

String-Darstellung von Objekten

- Die Klasse `Object` definiert eine Instanzen-Methode `public String toString()`, die eine String-Repräsentation des Objektes zurückgibt.
- In `Object` ist dies eine Zeichenfolge zusammengesetzt aus dem Klassen-Namen, dem `@`-Zeichen und der Hexadezimal-Darstellung des Hash-Codes des Objektes, der für Objekte vom Typ `Object` aus der internen Speicherdarstellung erzeugt wird. (Die eigentliche Bedeutung des Hash-Codes lernen wir später kennen.)
- Da jede Klasse eine Unterklasse von `Object` ist, hat auch jedes Objekt von beliebigem Typ diese Fähigkeit, eine String-Repräsentation von sich zu geben.
- Wie wir in den Übungen bereits gesehen haben, kann man für eine bestimmte Klasse eine andere (und möglicherweise sinnvollere) Repräsentation implementieren.

String-Darstellung von Objekten

- Beispiele:
  - Ein Objekt vom Typ `StringBuilder` gibt den Inhalt der internen Zeichenkette als String zurück.
  - Ein Objekt vom Typ `Double` gibt eine String-Repräsentation der dargestellten Zahl zurück.
- Diese Methode wird oftmals implizit aufgerufen, z.B., wenn man der Methode `System.out.println(Object x)` ein Objekt übergibt.