Objektorientierte Programmierung

Kapitel 17: Strings

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/oop18/

Inhalt

1. Einführung

2. StringBuilder

Strings (1)

- String-Konstanten wie `"abc"` wurden in Kapitel 4 behandelt.

  Insbesondere wurden da auch die Escape-Sequenzen (wie `\n`) behandelt, mit denen man spezielle Zeichen in die Zeichenketten einbetten kann.

- Der Compiler erzeugt für Zeichenketten-Konstanten Objekte der Klasse `String` (genauer `java.lang.String`).

- Diese Bibliotheks-Klasse hat eine große Anzahl von Methoden, die unter unter folgender Webadresse dokumentiert sind:
  [http://docs.oracle.com/javase/7/docs/api/java/lang/String.html]

- Außer den Konstanten sind noch der Konkatenations-Operator `+` und die Methode `toString` in die Sprache Java eingebaut.

  Für eigene Klassen kann man keine Operatoren definieren (in C++ schon).

- Ansonsten ist es eine normale Klasse.

Strings (2)

Folgen von Zeichen:

- Strings sind recht ähnlich zu `char[]`-Arrays.
- Man kann die Länge eines Strings `s` abfragen mit

  `s.length()`

  Achtung: Hier Methode (erfordert "`()`" für Parameter), bei Arrays Attribut.
  Mögliche Erklärung: Man wollte gleiches Interface wie `StringBuilder` (s.u.), dort Länge variabel, also ging `final` Attribut nicht, Attr. ohne `final` unsicher.
- Ein Zeichen des Strings bekommt man mit

  `s.charAt(i)`

  Dabei muss `i` wie bei Arrays einen Wert haben zwischen `0` (für das erste Zeichen) und `s.length() - 1` (für das letzte Zeichen).
- `"abc".charAt(i)` wäre legal (mit $0 \leq$ `i` ≤ 2).

  `"abc"` ist ja ein Objekt der Klasse `String`.

Strings: Vergleich (1)

- Wenn man Strings mit `==` vergleicht, wird wie üblich nur getestet, ob es sich um das gleiche Objekt handelt.

  Es wird nicht der Inhalt des Strings verglichen, d.h. die Zeichenfolge.

- Z.B. gibt folgendes Programmstück "Verschieden" aus:

```
String s = "abc";
String t1 = "ab", t2 = "c", t = t1 + t2;
if(s == t)
    System.out.println("Gleich");
else
    System.out.println("Verschieden");
```

- Der Vergleich zweier Strings funktioniert sicher mit

```
s.equals(t)
```

  Diese Methode liefert einen Wahrheitswert (Typ `boolean`), und zwar `true`, wenn beide `String`-Objekte die gleiche Zeichenfolge repräsentieren, sonst `false`.

Strings: Vergleich (2)

- Für im Programm direkt als Konstanten vorkommende Zeichenketten (`String`-Literale) ist garantiert, dass der Compiler für jede Zeichenkette nur ein `String`-Objekt erzeugt. Dort würde der Vergleich mit `==` funktionieren.

  Dies gilt auch für Ausdrücke, die sich nur aus Konstanten und Operatoren wie + zusammensetzen. Sie werden zur Compilezeit berechnet, nicht erst zur Laufzeit. Daher ist z.B. `"abc" == "a" + "bc"` (bei `javac 1.7.0_75`).

- Wenn man einen String `s` berechnet hat, kann man mit `s.intern()` ein kanonisches Objekt dafür bekommen.

  Das System hat eine Liste von String-Objekten, die mit allen direkt als String-Literal im Programm vorkommenden Strings initialisiert ist. `intern()` schaut nach, ob es die vom übergebenen Objekt repräsentierte Zeichenkette darin schon gibt. Falls ja, liefert sie das Objekt aus der Liste. Falls nein, trägt sie das Objekt ein, und liefert das Objekt.

Strings: Sortierung

- Der Aufruf `s1.compareTo(s2)` liefert
  - einen negativen Wert, wenn `s1` lexikographisch vor `s2` kommt,
  - `0`, wenn beide gleich sind, d.h. `s1.equals(s2)`
  - einen positiven Wert, wenn `s1` nach `s2` kommt.
- Dabei werden die einzelnen Zeichen nach Unicode-Werten geordnet, was nicht die übliche alphabetische Anordnung ist.

  Diese ist Länder-spezifisch. Für eine deutsche Sortierung verwendet man

  ```
  java.util.Locale german = new java.util.Locale("de");
  java.text.Collation coll = java.text.Collatoion.getInstance(german);
  ```

  (Alternativ gibt es `java.util.Locator.GERMAN`, und `getInstance()` verwendet die default Locale.) Dann Vergleich mit `coll.compare(s1, s2)`.

Strings: Erzeugung

- Man kann einen `String` aus einem Zeichen-Array erzeugen:

```
char[] array = { 'a', 'b', 'c' };
String s = new String(array);
```

  Man kann einen String auch aus `byte[]` erzeugen, dann muss man aber einen Zeichensatz angeben, oder sich auf den Default-Zeichensatz verlassen.

- Man kann beliebige Typen mit der statischen Methode `valueOf` umwandeln, z.B.

```
String s = String.valueOf(123);
```

- Ein bestimmtes Format bekommt man mit

```
String s = String.format("%4.1f", 1.23);
```

  Vier Zeichen insgesamt (inklusive ","), eine Dezimalstelle nach dem Komma.

Strings: Weitere nützliche Methoden (1)

- `int indexOf(int c)`:
  Index des ersten Vorkommens des Zeichens `c`.

  Z.B. würde nach `String s = "abc";` der Aufruf `s.indexOf('b')` den Wert `1` liefern. `-1` wird geliefert, wenn das Zeichen nicht vorkommt.
  Der Typ ist `int`, weil es auch mit Unicode Code Points funktioniert (s.u.).
  Alle diese Methoden gibt es auch in einer Variante mit ganzen Strings statt einzelnen Zeichen (Suche nach Substrings): `"abcdcd".indexOf("cd") == 2`

- `int indexOf(int c, int i)`:
  Index des nächsten Vorkommens von `c` ab Position `i`.

  Die Suche beginnt an der Index-Position `i` (einschließlich).
  Es gibt keinen Fehler, wenn `i` zu groß ist, dann wird `-1` geliefert.

- `int lastIndexOf(int c)`:
  Index des letzten Vorkommens.

Strings: Weitere nützliche Methoden (2)

- `String substring(int start, int end)`:
  Teilzeichenkette extrahieren.

  Die Zeichenkette beginnt bei Index `start` und endet bei `end-1`. Bis zum Ende der Eingabezeichenkette `s` kommt man also mit `end = s.length()`.

- `String toLowerCase()`:
  Umwandlung in Kleinbuchstaben.

  Entsprechend liefert `toUpperCase()` den String in Großbuchstaben.
  Die Methode `trim()` entfernt Leerplatz an Anfang und Ende.

- `String replace(char c1, char c2)`:
  Ersetzt alle Vorkommen von c1 durch c2.

  Es gibt auch eine Variante mit Strings (oder, noch allgemeiner, `CharSequence`-Werten). Bei `replaceAll` und `replaceFirst` kann man ein Muster (regulären Ausdruck) für den zu ersetzenden Text angeben.

Strings: Unicode-Unterstützung (1)

- Strings sind genauer Folgen von UTF-16 “Code Units”.
- Unicode-Zeichen haben “Code Points” zugeordnet, die `int`-Werte sind (bis `0x10FFFF`).

  Dies ist der eigentliche Zeichencode. Viele Programmierer kümmern sich nicht um diese Unterscheidung, für die häufigsten Sprachen funktioniert es, und wenn man nur Texte ausgibt, ist die genaue Anzahl Zeichen meist nicht wichtig.

- In UTF-16 werden viele Zeichen mit 16 Bit codiert (z.B. alle ASCII-Zeichen und deutsche Umlaute), aber es gibt auch Zeichen, die zwei aufeinanderfolgende 16-Bit Einheiten (`char`-Werte) benötigen.

  Diese Teile heissen “High-Surrogate” (`'\uD800'` bis `'\uDBFF'`) und “Low-Surrogate” (`'\uDC00'` bis `'\uDFFF'`). In diesen Bereichen liegen keine normalen Zeichen. Jeder Teil enthält 10 Bit, man muss sie wieder zusammenfügen und $2^{16} = 65536$ addieren, um den Code Point zu bekommen.

Strings: Unicode-Unterstützung (2)

- Folgende Methode liefert den echten Zeichencode an einer Position (ggf. zusammengesetzt aus zwei `char`-Werten):

  ```
  int codePointAt(int index)
  ```

- Die Anzahl tatsächlicher Zeichen zwischen zwei Index-Positionen in einem String erhält man mit

  ```
  int codePointCount(int begin, int end)
  ```

  Wenn man z.B. `begin = 0` und `end = s.length()` setzt, erhält man die tatsächliche Länge des Strings in Zeichen (im Gegensatz zur Länge in 16-Bit Code Units, die `length()` liefert).

- Die Klasse `Character` enthält einige (statische) Hilfsfunktionen für die Arbeit mit Code Points, z.B. liefert

  ```
  static int charCount(int codePoint)
  ```

  die Anzahl der `char`-Werte für einen Zeichencode (1 oder 2).

Inhalt

1 Einführung

2 StringBuilder

StringBuilder (1)

- Objekte der Klasse `String` sind unveränderlich.
- Will man einen String konstruieren, so kann man dazu ein Objekt der Klasse `StringBuilder` verwenden.

  [http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html]

  Die Klasse `StringBuffer` hat die gleiche Schnittstelle, und ist sicher für parallele Zugriffe aus mehreren Threads, deswegen aber nicht ganz so effizient. Wenn nur ein Thread darauf zugreifen muss, wird die Klasse `StringBuilder` empfohlen, die mit Java 5 eingeführt wurde.
- Natürlich kann man einen String auch Zeichen für Zeichen durch Konkatenation mit `+` aufbauen, das wäre aber nicht effizient (Verschwendung von Laufzeit und Speicherplatz).

  Es wird jedesmal ein neues `String`-Objekt konstruiert, das der Garbage Collector später wieder einsammeln muss. Es ist damit zu rechnen, das jedes Mal der bisherige String kopiert wird.

StringBuilder (2)

- Man kann ein Objekt dieser Klasse z.B. erzeugen mit

```
StringBuilder s = new StringBuilder(80);
```

- Dabei ist 80 die initiale Kapazität, also die Größe des Zeichen-Arrays in dem `StringBuilder`-Objekt.
- Wenn die Größe später nicht reichen sollte, wird sie automatisch erweitert, aber das kostet natürlich Laufzeit.

  Es muss ein neues, größeres Array angefordert werden und die Zeichen müssen vom alten in das neue Array umkopiert werden. Der Garbage Collector muss das alte Array wieder einsammeln.
  Andererseits kostet ein zu großes Array Speicherplatz.

- Nach der Erzeugung enthält der `StringBuilder` zunächst den leeren String `""`.

StringBuilder (3)

- Der in einem `StringBuilder` `s` gespeicherte String kann geändert werden z.B. mit

```
s.append("abc");
```

  Hiermit wird der String `"abc"` hinten an den bereits gespeicherten String (initial `""`) angehängt. Die Methode `append` akzeptiert beliebige Typen, die ggf. in Strings umgewandelt werden (formal handelt es sich um mehrere Methoden mit gleichem Namen aber unterschiedlichen Argument-Typen).

- Man kann einen String (oder einen beliebigen Wert) in den gespeicherten String auch an beliebiger Stelle einfügen:

```
s.insert(1, "de")
```

  Wenn in `s` vorher der String `"abc"` gespeichert war, steht dort jetzt der String `"adebc"`, d.h. `"de"` wurde an der Index-Position `1` eingefügt, was dort vorher stand, wurde nach hinten verschoben.

StringBuilder (4)

- `StringBuilder` hat auch die von `String` bekannten Methoden `charAt` und `length()`.

  Das ist kein Zufall, sondern liegt an dem Interface `CharSequence`, das von beiden Klassen implementiert wird. Interfaces wurden in Kapitel 13 behandelt.

- Dabei liefert die Methode `length()` die aktuell in dem `StringBuilder` gespeicherte Zeichenzahl (`char`-Werte).

  Und nicht die Kapazität. Diese erhält man mit `capacity()`.

- Mit `toString()` erhält man ein `String`-Objekt zum aktuellen Inhalt des `StringBuilders`.

  Der String ist unveränderlich und damit unabhängig von eventuellen zukünftigen Änderungen des `StringBuilder`-Objektes. Die Klasse `String` hat auch einen Konstruktor mit einem `StringBuilder` als Parameter.