LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Skript zur Vorlesung:

Einführung in die Programmierung

WiSe 2009 / 2010

Prof. Dr. Christian Böhm
Annahita Oswald
Bianca Wackersreuther

Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Gliederung

Zeichenketten

- Ein wichtiges Element für jede Programmiersprache sind Zeichenketten (z.B. zur Modellierung von Ein- / Ausgabe, Verarbeitung von Texten).
- Als grundlegenden Typ für (einzelne!) Zeichen kennen wir den primitiven Typ **`char`** und die Wrapper-Klasse `Character`.
- Zeichen*ketten*, also Sequenzen von mehreren Zeichen, werden durch die Klasse `String` repräsentiert.

Was sind Strings für den Compiler?

- Der Compiler kennt einige elementare Eigenschaften von Zeichenketten:
  - Er erkennt String-Literale (z.B. `"Hello, World!"`) und erzeugt daraus String-Objekte.
  - Er kann Strings verknüpfen (+-Operator).
- Intern ist ein String eine Reihung von **`char`**s.
- Ein Großteil der Implementierung der Eigenschaften von Zeichenketten ist der Laufzeitbibliothek überlassen (z.B. Vergleich von Zeichenketten, Extraktion von Teilstrings).

Die Klasse String

- Als Klasse hat String einige Besonderheiten:
  - Ein String kann aus Literalen erzeugt werden (s.o.).
  - Auf Strings ist ein Operator definiert (s.o.).
  - String-Objekte sind nicht dynamisch (dazu später).
- Dennoch betrachten wir Strings als erstes Beispiel für Objekte ausführlicher, denn als Modellierung von Zeichenketten ist die Klasse String von zentraler Bedeutung für die effiziente Programmentwicklung.
- Im Folgenden werfen wir einen Blick auf einige wichtige Methoden der Klasse String. Sie sollten sich mit Hilfe der Java-Dokumentation

  (http://java.sun.com/javase/6/docs/api/)

  weiter mit der Klasse String vertraut machen.

Statische Methoden

- Die Klasse String bietet statische Methoden an, um aus primitiven Typen Strings zu erzeugen:
  - `static String valueOf(boolean b)`
  - `static String valueOf(char c)`
  - `static String valueOf(char[] c)`
  - `static String valueOf(int i)`
  - `static String valueOf(long l)`
  - `static String valueOf(float f)`
  - `static String valueOf(double d)`
- Bei der Konkatenation (z.B. `"Note: "+1.0`) werden diese Methoden (hier: `static String valueOf(double d)`) implizit verwendet.
- **Warum sind diese Methoden statisch?**

Konstruktoren

Außer aus Literalen kann man Strings auch durch verschiedene Konstruktoren erzeugen. Die wichtigsten sind:

- `String()` – erzeugt ein neues String-Objekt, das den leeren String repräsentiert (eine **`char`**-Sequenz der Länge 0).
- `String(String original)` – erzeugt einen neuen String, der die gleiche **`char`**-Sequenz repräsentiert wie das angegebene Original. Es wird also eine Kopie (des Strings, nicht der Referenz!) erzeugt.
- `String(char[] value)` – erzeugt einen String aus dem gegebenen Array von **`chars`**. Das neue String-Objekt ist danach unabhängig vom **`char`**-Array, d.h. Änderungen am **`char`**-Array haben keinen Auswirkung auf den String.
- `String(char[] value, int offset, int count)` – wie `String(char[] value)`, wobei man aber noch einen Ausschnitt des Arrays spezifiziert.

String-Objekte sind nicht veränderbar

- Ein einmal erzeugter String kann nicht mehr verändert werden.
- Das bedeutet, String-Objekte sind nicht dynamisch, auch wenn das manche Methoden suggerieren.

Länge einer Zeichenkette

- Die Länge einer Zeichenkette entspricht der Länge des zugrundeliegenden **`char`**-Arrays.
- Ein leerer String hat die Länge 0. (Und ein String der Länge 0 ist immer leer.)
- Bei einer Länge $n > 0$ enthält ein String n Zeichen, die an den Indexpositionen 0 bis $n - 1$ liegen.
- Ein String-Objekt gibt über seine Länge Auskunft durch die Methode **`int`** `length()`.

```
String s = new String("Hello, World!");
int l = s.length();
System.out.println("Laenge von \""+s+"\": "+l);
// Ausgabe:
// Laenge von "Hello, World!": 13
```

Zeichenextraktion

- Die Methode **`char`** `charAt(`**`int`** `index)` liefert das Zeichen an der gegebenen Stelle des Strings.
- Das erste Element hat den Index 0.
- Das letzte Element hat den Index `length()` - 1.
- Beispiel:

```
"Hello, World!".charAt(12); //Wert: '!'
```

Teilstring

- Die Methode `String substring(int beginIndex)` liefert den Teilstring von Stelle `beginIndex` an bis zum Ende des Strings.
- Die Methode
  `String substring(int beginIndex, int endIndex)`
  gibt zusätzlich, das Ende des zu extrahierenden Teilstrings an.

Achtung:
Ungewöhnlich ist bei dieser Methode, dass der Parameter `endIndex` die erste Stelle *nach* dem zu extrahierenden Teilstring angibt.

```
String s = "Hello, World!";
String s1 = s.substring(1,9);   // Wert: ello, Wo
String s2 = s.substring(12,13); // Wert: !
```

- Die Methode `String trim()` gibt eine Kopie des Strings zurück, bei der führende und schließende Leerzeichen (= Code ≤ 32) entfernt wurden.

Ersetzen von Teilen einer Zeichenkette

- Die Methoden `String toLowerCase()` bzw. `String toUpperCase()` geben den String zurück, der entsteht, wenn man alle Großbuchstaben im aufrufenden Objekt durch Kleinbuchstaben ersetzt bzw. umgekehrt.
- Die Methode `String replace(char oldChar, char newChar)` gibt den String zurück, der durch Ersetzen aller Vorkommen von `oldChar` durch `newChar` entsteht.

Nochmal: String-Objekte sind nicht veränderbar

All diese Methoden (`substring`, `trim`, `toLowerCase`, `toUpperCase`, `replace` u.a.) verändern niemals das aufrufende Objekt (das wäre auch gar nicht möglich, weil ein String-Objekt unveränderlich ist), sondern erzeugen ein neues Objekt mit den gewünschten Eigenschaften.

```
String o1 = "Kroeger";
String o2 = o1.replace('o', 'i').substring(0, 6)+"l";
System.out.println(o2+" & "+o1); // Ausgabe:
                                 // Kroeger & Kriegel
```

Nochmal: String-Objekte sind nicht veränderbar

Soll nur ein String mit den neuen Eigenschaften übrigbleiben?

```
String o1 = "Kroeger";
o1 = o1.replace('o', 'i').substring(0, 6)+"l";
System.out.println(o1);     // Ausgabe:
                            // Kriegel
```

Was passiert hier im Speicher?

Nochmal: String-Objekte sind nicht veränderbar

```
1   String o1 = "Kroeger";
2   o1 = o1.replace('o', 'i');
3   o1 = o1.substring(0, 6);
4   o1 = o1 +"l";
```

Stack nach Zeile 1:

```
o1 = <adr1>
```

Heap nach Zeile 1:

<adr1> : "Kroeger"

Nochmal: String-Objekte sind nicht veränderbar

```
1   String o1 = "Kroeger";
2   o1 = o1.replace('o', 'i');
3   o1 = o1.substring(0, 6);
4   o1 = o1 +"l";
```

Stack nach Zeile 2:

```
o1 = <adr2>
```

Heap nach Zeile 2:

<adr1> : "Kroeger"
<adr2> : "Krieger"

Nochmal: String-Objekte sind nicht veränderbar

```
1   String o1 = "Kroeger";
2   o1 = o1.replace('o', 'i');
3   o1 = o1.substring(0, 6);
4   o1 = o1 +"l";
```

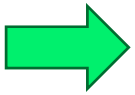
Stack nach Zeile 3:

```
o1 = <adr3>
```

Heap nach Zeile 3:

<adr1> : "Kroeger"
<adr2> : "Krieger"
<adr3> : "Kriege"

Nochmal: String-Objekte sind nicht veränderbar



Es wird also immer mehr Speicher im Heap belegt. Auf die alten Zustände von `o1` kann allerdings nicht mehr zugegriffen werden (es gibt keine Referenz mehr), der entsprechende Speicherplatz kann also vom Garbage Collector bereinigt werden.

Stack nach Zeile 4:

```
o1 = <adr4>
```

Heap nach Zeile 4:

<adr1> : "Kroeger"
<adr2> : "Krieger"
<adr3> : "Kriege"
<adr4> : "Kriegel"

Warum sind Strings nicht veränderbar?

- String-Objekte spielen in nahezu jedem Programm eine wichtige Rolle.
- Gefahr bei veränderlichen Objekten: Das Objekt wird an einer anderen (dem Entwickler vielleicht gar nicht bekannten) Stelle im Programm verändert.
- Als nicht veränderbares Objekt kann ein String gefahrlos von mehreren Stellen des Programmes referenziert werden.
- Außerdem erfordern nicht-veränderbare Objekte keinen Synchronisationsaufwand in nebenläufigen Programmen (*Multithreading*).
- Strings implementieren deshalb das Design-Pattern *Immutable*, das genau auf die genannten Vorteile abzielt.

Designmerkmale einer *Immutable*-Klasse

- Alle Attribute sind **`private`**.
- Schreib-Zugriffe auf Attribute erfolgen ausschließlich im Konstruktor oder in Initialisierungsmethoden.
- Lesende Zugriffe auf Attribute sind nur erlaubt, wenn das Attribut selbst immutable oder ein primitiver Typ ist. Auf veränderliche Objekte oder Arrays als Attribute darf keine Zugriffsmöglichkeit bestehen.
- Wenn veränderliche Objekte oder Arrays an einen Konstruktor übergeben werden, dann müssen sie kopiert (geklont) werden, bevor sie Attributen zugewiesen werden dürfen.

Dynamisches Arbeiten mit Strings

- Es ist also durchaus erwünscht, dass ein einmal im Speicher liegendes String-Objekt seinen Wert nicht verändert.
- Nicht wünschenswert ist hingegen, dass sich der Speicher immer mehr mit "toten" String-Objekten füllt, auf die gar nicht mehr zugegriffen werden kann, denn eine Zeichenkette im Laufe eines Programmes dynamisch verändern zu können, ist durchaus auch eine häufige Anforderung.
- Hierfür gibt es eigene Klassen, deren Verwendung viele Programme deutlich beschleunigen kann, den `StringBuilder` und den `StringBuffer`.
  - `StringBuffer`: seit langem gebräuchlich, Thread-Safe.
  - `StringBuilder`: wurde in Version 1.5 eingeführt, betreibt keinen Synchronisationsaufwand.

Beide Klassen haben ansonsten ähnliche Methoden und zeigen ähnliches Verhalten.

Methoden von StringBuilder und StringBuffer

- Die gebräuchlichste Methode ist die `append`-Methode. Sie hängt den übergebenen Parameter an die momentane Zeichenkette an. Sie ist überladen und kann mit jedem primitiven Typ, Strings, einem StringBuilder bzw. StringBuffer und sogar jedem anderen Objekt aufgerufen werden.
- Ebenso überladen ist die `insert`-Methode, die den übergebenen Wert an einer ebenfalls als Parameter angegebenen Stelle einfügt.
- Weitere interessante Methoden sind `replace`, `reverse` und `substring`.
- Wenn die Zeichenkette schließlich feststeht, erhält man durch Aufruf der Methode `toString` den entsprechenden String.

Machen Sie sich mit diesen Klassen durch die Java-Dokumentation vertraut!