**In diesem Kapitel**:



# *Zeichenketten*

*What's the use of a good quotation if you can't change it?*
- Dr. Who, The Two Doctors

Zeichenketten sind in Java Objekte. Sie werden durch eigene Sprachkonstrukte unterstützt. Wir haben bereits in vielen Beispielen Anführungszeichen zur Erzeugung von Zeichenkettenobjekten benutzt. Ebenfalls haben wir die Operatoren + und += verwendet, um Zeichenketten zu neuen Objekten zu verknüpfen. Die Klasse `String` bietet jedoch noch viel mehr Funktionalität. Während Objekte der Klasse `String` nur-lesbar sind, bietet die Klasse `StringBuffer` veränderbare Zeichenketten. Dieses Kapitel beschreibt diese beiden Klassen sowie die Konvertierungsmöglichkeiten von Zeichenketten zu anderen Typen wie `integer` und `boolean.`

Alle Methoden von `String` und `StringBuffer,` die von der Klein- und Grossschreibung von Buchstaben betroffen sind, arbeiten gemäss den entsprechenden Methoden der Klasse `Character.` Diese sind in einem anderen Abschnitt beschrieben.

### *Grundlegende Operationen der Klasse* `string`

Die Klasse `String` ist für nicht veränderbare Zeichenketten geeignet und unterstützt Operationen auf diesen. Neue Objekte der Klasse `String` können implizit entweder durch in Anführungszeichen eingeschlossene Zeichenketten (wie etwa `"Grösse")` oder durch Anwendung der Operatoren + oder += auf zwei Objekte der Klasse `String` erzeugt werden. Man kann Objekte der Klasse `String` auch ausdrücklich durch Anwendung von `new` erzeugen. Die Klasse `String` bietet die folgenden Konstruktoren:

`public` **`String`**
: erzeugt ein neues, leeres Objekt der Klasse `String` mit dem Wert

`public` **`String(String`** `value)`
: erzeugt ein neues Objekt der Klasse `String,` das eine Kopie des angegebenen Objekts `value` ist.

Die zwei grundlegendsten Methoden der Klasse `String` sind `length` und `charAt.` Die

Methode `length` gibt die Anzahl der Zeichen in einer Zeichenkette zurück, während `charAt` das Zeichen (vom Typ `char`) an der angegebenen Position liefert. Die folgende Schleife zählt die Häufigkeit eines jeden Zeichens in einer Zeichenkette:

```
for  (int i = 0; i < str.length i++
     counts[str.charAt(i)]++;
```

Ein Zugriff auf eine Zeichenkettenposition kleiner Null oder grösser als `1ength( )-1` erzeugt eine Ausnahme vom Typ `Index0utOfBoundsException.` Dies gilt nicht nur für `charAt,` sondern auch für jede andere Methode von `String.` Solche ungültigen Zugriffe weisen üblicherweise auf fehlerhafte Programmierung hin.

Ebenfalls gibt es einfache Methoden, um das erste oder letzte Auftreten eines Zeichens oder einer (Teil-)Zeichenkette (engl. *substring)* in einer Zeichenkette zu finden. Folgende Methode liefert die Anzahl der Zeichen zwischen dem ersten und letzten Auftreten des angegebenen Zeichens in einer Zeichenkette:

```
static int countBetween(String str, char ch)
     int begPos = str.index0f(ch);
     if (begPos < 0) // nicht vorhanden
          return -1;
     int endPos = str.lastIndex0f(ch);
     return endPos - begpos - 1;
```

Die benutzten Methoden finden die erste bzw. letzte Position des Zeichens in der Zeichenkette. Ist das Zeichen nicht enthalten, liefern sie -1. Die Differenz der beiden Positionen ist um 1 grösser als die Anzahl der Zeichen dazwischen. Im Beispiel: Sind die Positionen 2 und 3, so ist die Zahl der Zeichen dazwischen 0.

Die Methoden `index0f` und `lastIndexOf` sind mehrfach überladen, wobei `indexOf` stets eine Suche vorwärts im Text durchführt, `lastIndexOf` dagegen rückwärts. Jede liefert den Positionsindex bei erfolgreicher oder -1 bei erfolgloser Suche (siehe nebenstehende Tabelle).

*Übung 8-1:* Schreiben Sie eine Methode, die die Häufigkeit eines angegebenen Zeichens in einer Zeichenkette zählt.

*Übung 8-2:* Schreiben Sie eine Methode, die die Häufigkeit einer bestimmten Zeichenkette in einer anderen Zeichenkette bestimmt.

| **Methode** | **liefert Index von** |
|---|---|
| `index0f(char ch)` | der ersten Position von `ch` |
| `indexOf(char ch, int start)` | der ersten Position von `ch > start` |
| `index0f(String str)` | der ersten Position von `str` |
| `index0f(String str, int Start)` | der ersten Position von `str > start` |
| `lastIndex0f(char ch)` | der letzten Position von `ch` |
| `lastIndex0f(char ch, int Start)` | der letzten Position von `ch < start` |
| `lastIndex0f(String str)` | der letzten Position von `str` |
| `lastIndex0f(String str, int start)` | der letzten Position von `str < start` |

### *Vergleiche von Zeichenketten*

Die Klasse `String` bietet verschiedene Methoden zum Vergleich von Zeichenketten und von Teilen von Zeichenketten. Bevor wir die Methoden beschreiben, müssen wir zunächst erwähnen, dass internationale bzw. lokale Aspekte von Zeichenketten in Unicode nicht mit diesen Methoden behandelt werden. Wenn zum Beispiel zwei Zeichenketten bei einer Sortierung verglichen werden müssen, um zu bestimmen, welcher «grösser» ist, werden die Zeichen in den Ketten numerisch über ihre Werte im Unicode verglichen, und nicht nach lokal üblichen Anordnungen. Für einen Franzosen sind c und ç derselbe Buchstabe mit dem Unterschied einer kleinen diakritischen Kennung. Beim Sortieren von Zeichenketten im Französischen sollte man diesen Unterschied ignorieren, so dass zum Beispiel `"açb"` vor `"acz"` eingeordnet wird. Aber die Anordnung der Zeichen im Unicode ist anders: c (\ u063) kommt vor ç (\u0e7), so dass diese Zeichenketten in umgekehrter Reihenfolge angeordnet werden.

Die erste Vergleichsoperation ist `equals`. `equals` liefert `true`, wenn die ihr übergebene Objektreferenz auf ein Objekt der Klasse `String` mit demselben Inhalt zeigt, d.h. wenn beide Zeichenketten dieselbe Länge und genau dieselben Unicode-Zeichen in gleicher Reihenfolge enthalten. Ist das andere Objekt nicht vom Typ `String` oder sind die Inhalte unterschiedlich, gibt `String.equals fa1se` zurück.

Zum Vergleich von **Zeichenketten ohne Unterscheidung** von Gross- und Kleinschreibung kann die Methode `equalsIgnoreCase` benutzt werden. Mit Verzicht auf die Unterscheidung von Gross- und Kleinschreibung meinen wir, dass E und e als gleich angesehen werden sollen, aber zu unterscheiden sind von Ë und ë. Zeichen ohne solche Unterschiede, wie zum Beispiel Satzzeichen, sind nur mit sich selbst gleich.

Zur Sortierung von Zeichenketten ist deren Vergleich nötig. Die Methode `compareTo` liefert einen `int` kleiner, gleich oder grösser Null, je nachdem, ob der angegebene String kleiner, gleich oder grösser ist. Die Ordnung ergibt sich aus der Ordnung des UnicodeZeichensatzes.
Die Methode `compareTo` kann zur Erzeugung einer internen, kanonischen Anordnung von Zeichenketten verwendet werden. Eine binäre Suche zum Beispiel erfordert eine sortierte Liste der Elemente, wobei die konkret benutzte Ordnung unwichtig ist, so dass auch eine an lokalen Sprachgegebenheiten orientierte Ordnung ausreichend ist.

Nachfolgend eine Methode zur binären Suche für eine Klasse mit einem sortierten Feld von Zeichenketten:

```
private String table[ ]

public int position(String key) {
      int lo = 0;
      int hi = table.length - 1;
      while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            int cmp = key.compareTo(table[mid]);
            if    (cmp == 0)        // gefunden!
                  return mid;,
            else if (cmp < 0)       // im unteren Bereich suchen
                  hi = mid - 1;
            else                    // im oberen Bereich suche
                  lo = mid + 1;
      }
      return - 1;             //nicht gefunden
}
```

Dies ist der übliche binäre Suchalgorithmus. Erst wird in der Mitte des Suchbereichs geprüft, ob die gesuchte Zeichenkette kleiner, gleich oder grösser dem dortigen Element ist. Wenn sie gleich sind, so ist das Element gefunden und die Suche kann enden. Wenn die gesuchte Zeichenkette kleiner ist als das Element an der Position, so wird in der unteren Hälfte des Bereichs weitergesucht, anderenfalls in der oberen. Entweder wird das Element gefunden, oder die untere Grenze des Suchbereichs überschreitet die obere, was bedeutet, dass die gesuchte Zeichenkette nicht in dem Feld enthalten ist.

Zusätzlich zu ganzen Zeichenketten können auch Teilbereiche von Zeichenketten auf Gleichheit geprüft werden. Die Methode hierfür heisst `regionMatches` und hat zwei Formen: eine prüft auf exakte Übereinstimmung (wie bei `equals`), die andere ermittelt Übereinstimmung ohne Unterscheidung von Gross- und Kleinschreibung (wie bei `equalsIgnoreCase`):

`public boolean` **`regionMatches(int start,String other,int ostart, int len)`**

liefert `true`, wenn der gegebene Bereich von `String` mit dem gegebenen Bereich der Zeichenkette `other` übereinstimmt. Der Vergleich beginnt für die aktuelle Zeichenkette bei `start`, für die andere Zeichenkette bei `ostart`. Nur die ersten `len` Zeichen werden verglichen.

`public boolean` **`regionMatches(boolean ignoreCase, int Start, String other, int ostart, int len)`**

Diese Version von `regionmatches` verhält sich wie die zuvor beschriebene. Zusätzlich ist die Vernachlässigung von Gross- und Kleinschreibung wählbar.

Zum Beispiel:

```
class RegionMatch {
      public static void main(String[] args) {
            String str = "Look, look!";
            boolean b1, b2, b3;

            b1 = str.regionMatches(6, "Look", 0, 4);
            b2 = str.regionMatches(true, 6, "Look", 0, 4);
            b3 = str.regionMatches(true, 6, "Look", 0, 5);

            System.out.println("b1 = " + b1);
            System.out.println("b2 = " + b2);
            System.out.println("b3 = " + b3);
      }
}
```

Und dazu die Ausgabe:

```
      b1 = false
      b2 = true
      b3 = false
```

Der erste Vergleich liefert `fa1se,` da das Zeichen an Position 6 der langen Zeichenkette ein `'1'` ist, das Zeichen an Position 0 der anderen Zeichenkette dagegen ein `'L'` Der zweite Vergleich liefert `true,` da hierbei die Grossschreibung nicht von Bedeutung ist. Der dritte Vergleich ergibt `fa1se,` weil die Vergleichslänge nun 5 ist und die beiden Zeichenketten über diese fünf Zeichen selbst bei Ignorierung der Grossschreibung nicht gleich sind.

Falls bei einer der Methoden wie beispielsweise `regionMatches` oder jenen, die wir gleich kennen lernen, liefert ein ungültiger Index einfach das Ergebnis `false`, ohne eine `Exception` zu werfen. Falls Sie ein null Argument an Stelle eines Objekts verwenden, wird eine `NullPointerException` geworfen.

Einfachere Vergleichsformen für Start- und Endstücke von Zeichenketten sind mit den Methoden `startsWith` und `endsWith` möglich:

`public boolean` **`startsWith(String prefix, int toffset)`**
: liefert `true,` wenn dieses Objekt vom Typ `String` (bei `toffset)` mit dem angegebenen `prefix` beginnt.

`public boolean` **`startsWith(String prefix)`**
: ist eine Abkürzung für `startsWith(prefix, 0).`

`public boolean` **`endsWith(String suffix)`**
: liefert `true` wenn dieses Objekt vom Typ `String` mit dem angegebenen Suffix endet.

## String Literal Äquivalenz

Im allgemeinen können Zeichenketten nicht mit == verglichen werden:

```
if (str == "¿Peñia?") //Das macht keinen Sinn!
     answer(str);
```

Dies vergleicht nicht die *Inhalte* zweier Objekte vom Typ String. Es vergleicht lediglich eine *Objektreferenz* (str) mit einer anderen (auf ein vorübergehend angelegtes Objekt mit konstantem Inhalt `"¿Peñia?"` zeigenden). Die beiden Referenzen verweisen sicherlich auf verschiedene Objekte, auch wenn beide denselben Inhalt haben.

Haben allerdings zwei Zeichenketten in Anführungszeichen - also zwei Zeichenkettenliterale - den gleichen Inhalt, so verweisen sie auf dasselbe String-Objekt. Deshalb realisiert == im folgenden Kontext wahrscheinlich auch einen Vergleich der Inhalte:

```
String str = "¿Peñia?";
if   (str == "¿Peñia?"
     answer(str);
```

Weil `str` initial auf ein Zeichenkettenliteral gesetzt wird, bedeutet der Vergleich mit einem anderen Zeichenkettenliteral gleichzeitig einen Vergleich auf dieselben Inhalte. Aber Vorsicht ist geboten: Dieser Trick funktioniert nur, wenn man sicher sein kann, dass alle Zeichenkettenreferenzen auf Zeichenkettenliterale verweisen. Wenn `str` auf ein andersartig erzeugtes String-Objekt umgesetzt wurde, etwa als Folge einer Benutzereingabe, wird der Operator == stets `fa1se` liefern, auch wenn der Benutzer `"¿Peñia?"` eingegeben haben sollte.

### *Hilfsfunktionen*

Die Klasse String bietet zwei in speziellen Anwendungen hilfreiche Funktionen. Die eine heisst `hashCode` und liefert einen Hashwert basierend auf dem Inhalt des Objekts vom Typ `String`. Zeichenkettenobjekte mit demselben Inhalt erhalten auch denselben Hashwert, allerdings können auch Objekte verschiedenen Inhalts denselben Hashwert erhalten. Hashwerte sind nützlich für Hashtabellen, wie sie zum Beispiel die Klasse `Hashtable` oder `HashMap` in `java.util` zur Verfügung stellt.

Die andere Hilfsmethode heisst `intern` und liefert ein `String`-Objekt mit demselben Inhalt wie das Objekt, von dem die Methode aufgerufen worden ist. Insbesondere erhält man für zwei beliebige Objekte vom Typ `String` mit demselben Inhalt über `intern` ein und dasselbe `String`-Objekt. Dies ermöglicht, den Gleichheitstest indirekt über einen Vergleich von *Referenzen* auf Zeichenketten zu realisieren, anstatt einen langsameren Vergleich der *Inhalte* der Zeichenkettenobjekte vorzunehmen.

**Zum Beispiel:**

```
int putIn(String key) {
      String unique = key.intern
      int i;
      // Nachschauen, ob key nicht schon in table
      for (i = 0; i < tableSize; i++)
            if (table[i] == unique)
                  return i;
      // es ist noch nicht da, also hinzu
      table[i] = unique; tableSize++;
      return i;
}
```

Alle Objekte vom Typ String in dem Feld `tab1e` sind das Ergebnis von `intern` Aufrufen. Wir suchen in der Tabelle nach einem Zeichenkettenobjekt, das wir als Ergebnis von `intern` auf `key` erhalten haben und das eine eindeutige Referenz für den Inhalt darstellt. Wenn wir diese Referenz gefunden haben, ist nichts mehr zu tun. Wenn nicht, fügen wir diesen eindeutigen Repräsentanten für den Inhalt von `key` am Ende hinzu. Da immer die Ergebnisse von `intern` benutzt werden, ist der Vergleich der Objektreferenzen gleichbedeutend mit dem Vergleich der Inhalte, aber bedeutend schneller.

### *Abgeleitete Zeichenketten*

Mehrere Methoden von `String` ergeben neue Objekte von Typ `String`. Oft sind diese gegenüber dem ursprünglichen nur leicht modifiziert. Es werden neue Objekte zurückgegeben, da Objekte vom Typ `String` nur lesbar sind. Um bestimmte Teilzeichenketten aus einer Zeichenkette herauszugreifen, kann man eine Methode wie die folgende anwenden:

```
public static String delimitedString( String from, char start, char
end) {

      int startPos = from.index0f(start);
      int endPos = from.lastIndexOf(end);
      if (startPos >  endPos)    // start nach end
            return null;
      else if (startPos == -1)   // kein Anfang gefunden
            return null;
      else if (endPos == -1)          // kein Ende gefunden
            return from.substring(startPos);
      else                  // Anfang und Ende gefunden
            return from.substring(startPos, endPos + 1);
}
```

`delimitedString` liefert ein neues `String`-Objekt, das die mit den Zeichen `start` und `end` eingefasste Teilzeichenkette enthält. Wird das Zeichen `start` gefunden, aber `end` nicht, wird die Teilzeichenkette von der Startposition bis zum Ende der ursprünglichen Zeichenkette zurückgegeben. `delimitedString` nutzt dazu zwei durch Überladung entstandene Formen von `substring`. Die erste Form erhält lediglich eine Startposition und liefert ein neues `String`-Objekt mit allen Zeichen aus der Ursprungszeichenkette von der Startposition an. In der zweiten Form wird sowohl eine Start- als auch eine Endposition übergeben und ein neues `String`-Objekt mit allen Zeichen der ursprünglichen Zeichenkette von der Startposition bis zur Endposition. Dabei ist das Zeichen an der Startposition eingeschlossen, das an der Endposition aber nicht. Deshalb mussten wir, um beide begrenzenden Zeichen im Rückgabeobjekt zu erhalten, zu der Endposition `endPos` eine 1 hinzuaddieren.

Zum Beispiel liefert der Aufruf

```
delimitedString("Il a dit -«Bonjour!»", '«', '»');
```

als Rückgabe

```
«Bonjour!»
```

Nachfolgend nun die noch ausstehenden Methoden, die «verwandte» bzw. «abgeleitete» Zeichenketten erzeugen:

`public String` **`replace(char oldChar, char newChar)`**
liefert ein neues `String`-Objekt, in dem jedes Auftreten von `oldChar` durch das Zeichen `newChar` ersetzt sind.

`public String` **`toLowerCase()`**
liefert ein neues `String`-Objekt, in dem Grossbuchstaben durch die entsprechenden Kleinbuchstaben ersetzt wurden.

`public String` **`toUpperCase`**`()`
liefert ein neues `String`-Objekt, in dem Kleinbuchstaben durch die entsprechenden Grossbuchstaben ersetzt wurden.

`public String` **`trim`**`()`
liefert ein neues `String`-Objekt, in dem alle führenden und abschliessenden Wort-Zwischenräume wie Leer- und Tabulatorzeichen entfernt sind.

Die Methode `concat` bildet eine neue Zeichenkette durch Anhängen einer zweiten Zeichenkette und arbeitet damit genauso wie der Konkatenationsoperator +. Die folgenden zwei Anweisungen sind gleichbedeutend:

```
newStr = oldStr.concat(" not ");
newStr = oldStr + "  not";
```

*Übung 8-3:* Wie oben gezeigt wurde, geht `delimitedString` nur von einer (mit den angegebenen Zeichen) eingeklammerten Zeichenkette pro Eingabezeichenkette aus. Schreiben Sie eine Version, die alle enthaltenen Klammerungen extrahiert und in einem Feld zurückgibt.

### *Zeichenketten und Typumwandlung*

Oft ist es nötig, Zeichenketten in etwas anderes umzuwandeln, zum Beispiel in ganze Zahlen oder Wahrheitswerte oder umgekehrt. Per Java-Konvention soll der Typ, *zu dem gewandelt* wird, die Methode zur Umwandlung besitzen.

Zum Beispiel erfordert die Umwandlung eines String-Objektes in eine ganze Zahl eine klassenbezogene Methode in der Klasse Integer.

Nachfolgend eine Tabelle mit allen Typen, für die Java eine Umwandlung zu oder von Zeichenketten ermöglicht, und jeweils der Art, wie die Umwandlung realisiert wird:

| **Typ** | **nach String** | **von String new** |
|---|---|---|
| `boolean` | `String.valueOf(boolean)` | `new Boolean(String).booleanValue()` |
| `byte` | `String.valueOf(int)` | `Byte.parseByte(String, int base)` |
| `short` | `String.valueOf(int)` | `Short.parseShort(String, int base)` |
| `int` | `String.valueOf(int)` | `Integer.parseInt(String, int base)` |
| `long` | `String.valueOf(long)` | `Long.parseLong(String, int base)` |
| `float` | `String.valueOf(float)` | `Float.parseFloat(String)` |
| `double` | `String.valueOf(double)` | `Double.parseDouble(String)` |

Es gibt keine Möglichkeit zur Erzeugung oder Umwandlung von in Java verwendbaren Zahlenbeschreibungen - weder für die Schreibweise mit führendem `0` für die oktale Notation, noch für die Schreibweise mit führendem `0x` für die hexadezimale Notation.

Für `Boolean` und Gleitkommazahlen besteht die Technik aus der Erzeugung eines entsprechenden Objekts, dessen Wert abgefragt wird. Es gibt kein Äquivalent zu `parseInt` bei Gleitkommazahlen, die direkt den Wert herauslesen. Die Integer Pasing Methode besitzt zwei überladene Varianten: die eine akzeptiert eine numerische Basis zwischen 2 und 32, zusätzlich zur Zeichenkette, welche geparsed werden soll. Die andere Variante akzeptiert lediglich einen Parameter und setzt die Basis 10 voraus. Falls die Zeichenkette keine Zahl darstellt, ausser `Boolean`, wird eine `NumberFormatException` geworfen. Im Falle der `Boolean` Klasse gilt die Regel, dass alles, was nicht `true` (gross oder klein geschrieben) ist, als `false` interpretiert wird.

Es gibt keine Methoden zur Umwandlung von Zeichen in die Form, wie sie in Java erkannt werden (`\b, \udddd`, etc.), oder umgekehrt. Die Methode String. `valueOf ( )` ist auch auf ein einzelnes Zeichen (Typ `char`) anwendbar, und man erhält eine Zeichenkette aus genau diesem Zeichen.
Umwandlung von und nach `byte` und `short` werden über `int` durchgeführt: Sie sind im Zahlenbereich von `int` enthalten und werden bei der Auswertung von Ausdrücken sowieso nach `int` umgewandelt.

Eigene Klassen können Kodierung und Dekodierung aus Zeichenketten unterstützen, indem sie eine Methode `toString` und einen Konstruktor erhalten, der ein neues Objekt aus der Zeichenkettenbeschreibung erzeugt. Klassen mit einer Methode `toString` können auch mit `va1ueOf` benutzt werden. `va1ueOf (Object obj )` `ist` so definiert, dass es entweder "`nu11`" oder `obj.toString` zurückliefert. Besitzen alle Klassen eine Methode `toString`, kann auch jedes Objekt durch Aufruf von `va1ueOf` zu einem `String`-Objekt umgewandelt werden.

### *Zeichenketten und* `char`*-Felder*

Ein `String`-Objekt korrespondiert mit einem *Feld* mit Elementen vom Typ `char` und umgekehrt. Oft will man eine Zeichenkette in einem Feld aufbauen und anschliessend ein `String`- Objekt mit entsprechendem Inhalt erzeugen. Wenn dafür die später beschriebene Klasse `StringBuffer` (Zeichenketten mit änderbarem Inhalt) nicht angemessen ist, helfen einige Methoden und Konstruktoren von `String` bei der Umwandlung eines `String`-Objektes in ein `char`-Feld und umgekehrt.

Es gibt zwei Konstruktoren, mit deren Hilfe `String` Objekte aus `char` Arrays / Feldern gebildet werden können:

`public` **`String(char[ ] chars, int start, int count)`**
konstruiert eine neue Zeichenkette, deren Inhalt der selbe ist, wie im Zeichenfeld `chars`, ab Index `start` bis maximal `count` Zeichen.

`public` **`String(char[ ] chars)`**
ist äquivalent zu `String(chars, 0, chars.length)`

Beide Konstruktoren kopieren den Inhalt des Zeichenfeldes. Daher hat jede Änderung an den Elementen des Zeichenfeldes nach dem Konstruieren der Zeichenkette keinen Einfluss auf die Zeichenkette.

Zum Beispiel lässt sich damit ein einfacher Algorithmus formulieren, der jedes Auftreten eines bestimmten Zeichens aus einem `String`-Objekt entfernt (*squeeze out*):

```
public static String squeeze0ut(String from, char toss)
     char[ ]chars = from.toCharArray();
     int len = chars.length;
     int put = 0;
     for (int i = 0; i < len; i++)
          if (chars[il] != toss)
              chars[put++] = chars[i];
     return new String(chars, 0, put);
}
```

Die Methode `squeezeOut` wandelt zuerst die Eingabezeichenkette `from` mit der Methode `toCharArray` in ein Feld von Zeichen um. Dann wird `put` definiert, in der die nächste Position abgespeichert wird. Danach wird eine Schleife durchlaufen, in der alle Zeichen kopiert werden, welche nicht mit toss übereinstimmen. Wenn die Schleife über das Feld abgeschlossen ist, geben wir ein neues - die reduzierte Zeichenkette enthaltendes `String` Objekt zurück. Der benutzte Konstruktor `String` erhält als Argumente das Ursprungsfeld, die Startposition und die Zahl der Zeichen.

Ein weiterer Konstruktor von `String` erhält nur ein Feld von Zeichen als Parameter und übernimmt dann alle Zeichen. Beide Konstruktoren kopieren aus dem Feld. Dadurch kann man anschliessend das Feld verändern, ohne dass die Inhalte der neuen `String` Objekte davon betroffen wären.

Man kann fallweise die beiden klassenbezogenen Methoden `String.copyValue0f` anstelle des Konstruktors bevorzugen. So hätte `squeezeOut` auch enden können mit

```
return String.copyValue0f(chars, 0, len)
```

Es gibt ebenfalls eine ein-argumentige Form von `copyValueOf`, die das gesamte Feld kopiert. Ausserdem gibt es zwei zu den beiden Konstruktoren in String äquivalente Methoden `value0f`, wobei auch für diese die Bemerkung zur Veränderbarkeit der Felder nach deren Nutzung als Parameter gilt.

Die Methode `toCharArray` ist einfach und zumeist ausreichend für die Umwandlung in ein Feld. Falls eine gezieltere Kontrolle zum Kopieren von Teilen einer Zeichenkette in ein `char`-Feld nötig ist, kann die Methode `getChars` eingesetzt werden:

`public void` **`getChars(int srcBegin, int srcEnd, char [] dst, int dstBegin)`**

> kopiert Zeichen vom `String` -Objekt in das angegebene Feld. Die Zeichen der angegebenen Teilzeichenkette werden in das `char`-Feld beginnend bei `dst[dstBegin]` kopiert. Die Teilzeichenkette beginnt bei Position `srcBegin` und geht bis ausschliesslich `srcEnd`. Ein Zugriff ausserhalb der Grenzen des ursprünglichen `String` -Objekts oder des Zielfeldes löst eine Ausnahme vom Typ `Index0utOfBoundsException` aus.

### *Zeichenketten und* `byte`*-Felder*

Methoden zur Umwandlung zwischen einem Feld von in 8 Bit kodierten Zeichen und `String` -Objekten in 16-Bit-Unicode sind verfügbar. Die Methoden erlauben insbesondere die Erzeugung von Zeichenketten in Unicode aus ASCII- und ISO-Latin-1-Zeichen, die die ersten 256 Zeichen im Unicode-Zeichensatz ausmachen. Diese Methoden sind analog denen aus dem vorherigen Abschnitt:

`public` **`String(byte[] bytes, int start, int count)`**

Der Konstruktor legt ein neues `String` -Objekt an, dessen Wert dem gewählten Abschnitt entspricht. Dabei wird die Standardkodierung für Standard Locale verwendet.

`public` **`String(byte[] bytes)`**

ist eine Kurzform für `String(bytes, 0 , bytes.length)`

`public` **`String(byte[] bytes, int start, int count, String enc)`**
**`throws UnsupportedEncodingException`**

Konstruiert eine neue Zeichenkette durch Konversion der Bytes von `start` bis zu maximal `count` Bytes in Zeichen (`char`), mit Hilfe der Kodierung `enc`.

`public` **`String(byte[] bytes, String enc)`**
**`throws UnsupportedEncodingException`**

ist eine Kurzform für `String(bytes, 0, bytes.length, enc)`.

`public byte[ ]` **`getBytes()`**

liefert ein Byte-Feld, welches den Inhalt der Zeichenkette (auf die die Methode angewandt wird) mit Hilfe der Standardkodierung für die Standard Locale Einstellung.

`public byte[ ]` **`getBytes(String enc)`**
**`throws UnsupportedEncodingException`**

liefert ein Byte-Feld, welches den Inhalt der Zeichenkette (auf die die Methode angewandt wird) mit Hilfe der Standardkodierung für die Standard Locale Einstellung.

Die String Konstruktoren, welche Zeichenketten aus Bytes zusammenbauen, kopieren die Daten. Alle Änderungen an den Bytes und Bytearrays haben somit keinerlei Auswirkungen auf die gebildete Zeichenketten.

## Character Encodings

Ein Charakter Encoding spezifiziert wie rohe 8-Bit Zeichen und deren 16-Bit Unicode Äquivalente ineinander konvertiert werden sollen. Encodings werden gemäss ihrem Standardnamen benannt. Die lokale Plattform definiert, welche Zeichen Kodierungen verstanden werden. Jede Implementierung muss mindestens folgende Kodierungen unterstützen:

`US-ASCII` Sieben-Bit ASCII, auch bekannt unter ISO646-US, Basic Latin Block des Unicode Zeichensatzes.

`ISO-8859-1` ISO Latin Alphabet No 1, auch bekannt unter ISO-LATIN-1

`UTF-8` Acht-Bit Unicode Transformation Format

`UTF-16BE` Sechzehn-Bit Unicode Transformation Format, big-endian Byte Reihenfolge

`UTF-16LE` Sechzehn-Bit Unicode Transformation Format, little-endian Byte Reihenfolge

`UTF-16` Sechzehn-Bit Unicode Transformation Format, Byte Reihenfolge gemäss einer Byte-Ordnungsmaske

Je nach Release können Unterschiede auftreten. Sie müssten also vor einem Einsatz der Kodierung die jeweilige Releasedokumentation konsultieren.

Falls Sie eine `UnsupportedEncodingExceptionn` erhalten, wissen Sie, dass diese Kodierung nicht unterstützt wird.

Jede Instanz der JavaVirtual Machine besitzt eine Standardeinstellung für die Zeichenkodierung. Diese hängt typischerweise von den lokalen Einstellungen ab, welche auch mit `locale` in Java kontrolliert werden können.

### *Die Klasse* `StringBuffer`

Gäbe es eine Beschränkung auf nur-lesbare Zeichenketten, wäre man gezwungen, für jedes Zwischenresultat in einer Folge von Zeichenkettenbearbeitungen ein neues String-Objekt anzulegen. Betrachten wir zum Beispiel die Auswertung des folgenden Ausdrucks:

```
public static String guillemete(String quote) {
     return '«' + quote + ' »';
}
```

Bei einer (compilerbedingten) Einschränkung auf Ausdrücke über `String` wäre dies gleichbedeutend mit:

```
quoted =
String.valueOf('«').concat(quote).concat(String.valueOf('»'));
```

Jeder Aufruf von `valueOf` und `concat` legt ein neues `String`-Objekt an, also würde die Operation die Erzeugung von vier `String`-Objekten bewirken, von denen nur eines später benötigt würde. Damit sind die anderen Zusatzaufwand. Für ihre Freigabe muss erst die Speicherbereinigung aktiv werden.

Der Compiler benutzt eine effizientere Technik als diese. Er benutzt ein Objekt der Klasse `StringBuffer` für die Zeichenketten in Ausdrücken. Ein am Ende resultierendes `String`-Objekt wird, wenn nötig, angelegt. `StringBuffer`- Objekte können verändert werden, so dass nicht immer neue Objekte für Zwischenergebnisse erzeugt werden.

Bei Benutzung der Klasse `StringBuffer` wird der `String`- Ausdruck im Beispiel folgendermassen dargestellt:

```
quoted = new
StringBuffer().append('«').append(quote).append('»').toString();
```

Hier wird genau ein Objekt vom Typ `StringBuffer` angelegt, die Stringteile angehängt und abschliessend mit `toString` ein neues `String`-Objekt für das Ergebnis erzeugt.

Für Ihre Zeichenkettenbehandlungen werden Sie möglicherweise auch die Klasse `StringBuffer` benutzen wollen. Sie verfügt über zwei Konstruktoren:

`public` **`StringBuffer()`**
: erzeugt ein `StringBuffer` -Objekt mit initialem Wert

`public` **`StringBuffer(String str)`**
: erzeugt ein `StringBuffer` -Objekt mit dem Inhalt von `str` als initialem Wert.

Die Klasse `StringBuffer` ist mit der Klasse `String` vergleichbar und unterstützt Methoden mit denselben Namen und Vereinbarungen wie einige Methoden von `String`. Allerdings `StringBuffer` keine Erweiterung von `String` oder umgekehrt. Sie sind voneinander unabhängige Klassen und beide Erweiterungen von `Object`.

## Ändern des Puffers

Verschiedene Wege zur Änderung des Puffers eines `StringBuffer`-Objekts sind möglich, darunter das Anhängen und das Einfügen. Die einfachste Methode ist `setCharAt`, wodurch das Zeichen an einer bestimmten Position geändert wird. Es folgt eine Methode namens `replace`, die dieselbe Funktion wie `String.replace` haben soll, jedoch mit `StringBuffer`-Objekten arbeitet. Das Anlegen eines neuen Objekts ist in der Methode `replace` nicht nötig, so dass aufeinanderfolgende Aufrufe mit demselben Puffer arbeiten.

```
public static void replace(StringBuffer str,
                                    char oldChar, char newChar){
     for (int i = 0; i < str.length(); i++)
          if (str.charAt(i) == oldChar)
          str.setCharAt(i , newChar);
}
```

Die Methode `setLength` verkürzt oder verlängert die Zeichenkette im Puffer. Wenn Sie `setLength` mit einer kürzeren Länge als die der aktuellen Zeichenkette verwenden, wird die Zeichenkette auf die angegebene Länge am Ende abgeschnitten. Ist sie länger, wird die Zeichenkette verlängert und mit Nullzeichen (\u0000) aufgefüllt.

Es gibt weiterhin Methoden `append` und `insert`, die ein Objekt eines anderen Typs in ein `String`-Objekt umwandeln und dieses anschliessend am Ende anhängen beziehungsweise an der angegebenen Position einfügen. Die Methode zum Einfügen verschiebt Zeichen nach Bedarf, um Platz für die einzufügenden Zeichen zu schaffen. Die folgenden Typen werden von den Methoden `append` und `insert` umgewandelt:

```
Object          String          char[]
boolean         char            int
long            float           double
```

Zudem gibt es Methoden `append` und `insert`, die einen Teil eines `char`-Felds als Argument verwenden. Um beispielsweise ein `StringBuffer`- Objekt zu erhalten, das die Quadratwurzel einer ganzen Zahl enthält, kann man folgendes schreiben:

```
String sqrtInt(int i) {
     StringBuffer buf = new StringBuffer();
     buf.append("sqrt(").append(i).append(')');
     buf.append(" = ").append(Math.sqrt(i));
     return buf.toString();
}
```

Die Methoden `append` und `insert` geben das `StringBuffer`-Objekt selbst wieder zurück, so dass es uns möglich ist, darauf sofort wieder `append` anzuwenden.

Die Methode `insert` verwendet zwei Parameter. Der erste ist die Position, an der Zeichen in das `StringBuffer` -Objekt einzufügen sind. Der zweite ist der einzufügende Wert, der gegebenenfalls zunächst in ein `String`-Objekt umzuwandeln ist.

Nachfolgend eine Methode, um das aktuelle Datum am Anfang des Puffers zu ergänzen:

```
public static StringBuffer addDate(StringBuffer buf) {
      String now = new java.util.Date().toString();
      buf.insert(0, now).insert(now.length(), ":");
      return buf;
}
```

Zuerst erzeugen wir eine Zeichenkette mit der aktuellen Zeit und aktuellem Datum mittels `java.util.Date`. Deren Konstruktor legt ein Objekt mit einer Beschreibung des Erzeugungszeitpunkts als Inhalt an. Wir stellen sicher, dass der Puffer gross genug ist, um alle neuen Zeichen aufzunehmen. Dadurch wird der Puffer höchstens einmal im Rahmen von `addDate` vergrössert und nicht bei jedem `insert`-Aufruf. Dann fügen wir die Zeichenkette mit dem aktuellen Datum und zwei Trennzeichen ein. Wir geben den als Argument erhaltenen Puffer zurück, so dass nach Aufruf weitere Aufrufe zum Anhängen angewendet werden können, wie es sich bei den Methoden von `StringBuffer` schon als angenehm erwiesen hat.

Die `reverse` Methode kehrt die Reihenfolge der Zeichen eines `StringBuffer` um. Falls der Inhalt des Puffers "doog" ist, ist das Ergebnis von `reverse` "good".

Sie können Teile einer Zeichenkette mit Hilfe der `delete` Methode löschen. Diese Methode benötigt einen Start- und einen End- Punkt. Der Endpunkt selbst wird nicht gelöscht! Der Puffer wird also verkürzt.

Mit `deleteCharAt` wird ein einzelnes Zeichen an einer bestimmten Stelle gelöscht.

`public StringBuffer` **`replace(int start, int end, String str)`**

Ersetzt die Zeichen ab der Position `start` bis zu, aber ohne `end` durch die Zeichenkette `str`. Der Puffer wächst oder schrumpft je nach Länge der Zeichenkette `str` und dem zu ersetzenden Bereich.

`public StringBuffer` **`insert(int pos, char[ ] chars, int start, int count)`**

Fügt Zeichen aus dem Feld `chars` in die Zeichenkette ein. Das erste Zeichen ist an Position `pos` und die bestehenden Zeichen werden verschoben. Insgesamt werden `count` Zeichen hinein kopiert, ab `chars[start]`.

## Auslesen von Daten

Um ein `String`-Objekt von einem `StringBuffer`-Objekt zu erhalten, kann `toString` verwendet werden.

In `StringBuffer` gibt es keine einen Teil des Puffers löschende Methode - man muss ein Zeichen-Array aus dem Puffer erstellen und daraus einen neuen Puffer mit dem verbleibenden Inhalt konstruieren. Dies ist wohl die häufigste Anwendung der Methode `getChars`, die analog zu String. `getChars` arbeitet.

`public void` **`getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`**

> kopiert die Zeichen aus dem angegebenen Puffer-Teil (festgelegt durch `srcBegin` und `srcEnd`) in das Zeichenfeld `dst`, beginnend bei `dst [ dstBegin ]` .Das Kopieren beginnt bei Position `srcBegin` und geht bis *ausschliesslich* srcEnd. srcBegin muss ein gültiger Index des Puffers sein, und `srcEnd` darf nicht grösser als die aktuelle Länge sein, die um 1 über dem letzten gültigen Index liegt**.** Wenn einer der Indizes ungültig ist, wird eine Ausnahme vom Typ `Index0utOfBoundsException` ausgelöst.

Es folgt eine Methode, die `getChars` benutzt und einen Teil aus dem Puffer entfernt:

```
public static StringBuffer
      remove(StringBuffer buf, int pos, int cnt)

      if (pos < 0 || cntJ< 0 || pos + cnt > buf.length throw new
                                    Index0utOfBoundsException();
      int leftover = buf.length() - (pos + cnt);
      if (leftover == 0) {  // einfaches Abschneiden
            buf.setLength(pos);
            return buf;
      }

      char[] chrs = new char[leftover];
      buf.getChars(pos + cnt, buf.length , chrs, 0);
      buf.setLength(pos);
      buf.append(chrs);
      return buf
}
```

Zuerst stellen wir sicher, dass alle Feldzugriffe innerhalb gültiger Grenzen bleiben. Man könnte eine auftretende Ausnahme zwar später behandeln, aber eine frühzeitige Prüfung ermöglicht eine bessere Steuerung. Als nächstes berechnen wir, wie viele Zeichen nach dem gelöschten Teil übrigbleiben. Falls keine, schneiden wir einfach ab und geben das Ergebnis zurück. Anderenfalls holen wir den verbleibenden Teil mit `getChars` in ein Feld. Dann schneiden wir den Puffer ab und hängen die verbleibenden Zeichen wieder an, bevor wir das Ergebnis zurückgeben.

## Verwalten der Kapazität

Der Puffer eines `StringBuffer`- Objekts hat eine Kapazität. Dies ist die Länge der Zeichenkette, die es noch speichern kann, bevor es mehr Platz anfordern muss. Der Puffer wächst zwar automatisch mit jedem zusätzlichen Zeichen, aber eine nur einmalige Puffergrössenangabe ist effizienter.

Die initiale Grösse eines `StringBuffer`-Objekts kann beim Konstruktor durch ein Argument vom Typ `int` angegeben werden:

`public` **`StringBuffer(int capacity)`**
: erzeugt ein `StringBuffer`-Objekt mit der angegebenen initialen Kapazität und einem initialen Wert "" .

`public syncronized void` **`ensureCapacity(int minimum)`**
: stellt sicher, dass die Kapazität des Puffers mindestens die angegebene ist.

`public int` **`capacity()`**
: liefert die aktuelle Kapazität des Puffers.

Diese Methoden sind einsetzbar, um ein ineffektives mehrmaliges Wachsen des Puffers zu vermeiden. Als Beispiel folgt eine überarbeitete Version der Methode `sqrtInt`, bei der sichergestellt ist, dass zusätzlicher Pufferspeicher nur einmal angefordert wird:

```
String sqrtIntFaster(int i) {
      StringBuffer buf = new StringBuffer(50);
      huf.append("sqrt(").append(i).append(')') ;
      huf.append(" = ").append(Math.sqrt(i));
      return buf.toString();
}
```

*Übung 8-4*: Schreiben Sie eine Methode, die Zeichenketten mit Ziffern aus dem Dezimalsystem in eine Folge von Ziffern umwandelt, bei der nach jeder dritten Ziffer von rechts ein Komma eingefügt wird. So soll die Methode etwa für "1543729" als Parameter als Ergebnis die Zeichenkette "1,543,729" liefern.

*Übung 8-5*: Verändern Sie die Methode so, dass sie zwei Parameter erhält, einen für das Trennzeichen und einen für die Anzahl der Ziffern zwischen den Trennzeichen.

# Zeichenketten in Java