Prof. Dr. Verena Majuntke - Prof. Dr. Ulrich Meissen

Grundlagen der Programmierung

Strings (Zeichenketten)

14.12.2020

Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Themen

- Die Klasse *String*
- Die Klasse *StringBuffer*
- String-Vergleich

Lernziele

- Wichtige Methoden der Klasse String?
- Unterschied zwischen String und StringBuffer?
- Zeichen/-ketten im StringBuffer ersetzen?
- Gleichheit von String-Inhalten?
- Gleichheit von String-Objekten?
- Was ist der Literal-Pool?

Die Klasse *String*

Ein *String*

Definition eines Strings ohne Initialwert

```
String meinString;
```

Definition eines Strings mit Initialwert

```
String meinString = “Hallo Welt“;
String meinString = new String(“Hallo Welt“);
```

Definition eines Null-Wertes

```
String meinString = null;
```

Methoden der Klasse *String*

Die Klasse *String* bietet einige Operationen:

- length()
- toUpperCase()
- charAt(int i)
- …
- vollständige Liste siehe Java Doc (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html)

Strings zusammenfügen

Zusammenfügen (engl. concatenating) von Strings mittels des Plus-Operators:

```
String vorname = new String(“Frodo“);
String nachname = “Beutlin“;
System.out.println( vorname + “   “  + nachname );
```

Strings mit Operationen

Verwendung des Plus-Operators bei Operationen und Strings:

```
System.out.println(1 + 2 + "0"); => 30
```

aber

```
System.out.println("0" + 1 + 2); => 012!!!
```

String = konstante Zeichenkette

Strings lassen sich inhaltlich **nicht** ändern! Sie repräsentieren konstante Zeichenketten.

Man kann jedoch dem Referenzattribut ein **neues** *String*-Objekt zuweisen:

```
String  vorname = null;
        vorname = “Frodo“;
        vorname = “Gandalf“;
```

Die Klasse *StringBuffer*

Die Klasse *StringBuffer*

Die Klasse StringBuffer stellt einen dynamischen Behälter für Zeichen zur Verfügung, die sich jederzeit im Nachhinein ändern lassen:

Deklaration und Initialisierung:

```
StringBuffer buffer = new StringBuffer( 20 );
```

Operationen von *StringBuffer*

Die Klasse *StringBuffer* bietet einige Operationen:

- length() // Anzahl der Zeichen
- toString() // Umwandlung in einen String
- append() // Anhängen der Zeichen
- …

StringBuffer - Länge

Ermitteln der String-Länge in einem StringBuffer:

```
int len = buffer.length() ;

for( int i = 0; i < buffer.length() ; i++ ){
    System.out.print(“das“ + i + “te Zeichen  =  “);
    System.out.println( buffer.charAt(i) );
}
```

StringBuffer = variable Zeichenkette

StringBuffer lassen sich inhaltlich ändern!

Sie repräsentieren variable Zeichenketten:

```
String ort = "Hallo Wien";
StringBuffer buffer = new StringBuffer( ort );
buffer.deleteCharAt(5);
buffer.setCharAt(6, 'e');
System.out.println ( buffer.toString() );
    // was wird ausgegeben?
```

Konvertierung: StringBuffer zu String

```
StringBuffer buffer = new StringBuffer( “Frodo“ );
```

Konvertierung eines StringBuffers in einen String:

```
String name = buffer.toString();
```

Beispiel: StringBuffer und String

```
String   zeichenkette  = new String(“eine alte Ganz holt Essen“);
StringBuffer   buffer = new StringBuffer(zeichenkette);
// hier hat sich ein Fehler eingeschlichen, deshalb muss korrigiert werden:
// Hier muss man die Position aber schon wissen….
buffer.replace(10, 14, "Gans");
System.out.println(buffer.toString());
// Alternative: Hier muss man die Position aber schon wissen….
buffer.setCharAt(13, 's');
System.out.println(buffer.toString());
```

Beispiel: StringBuffer und String

```
String   zeichenkette  = new String(“eine alte Ganz holt Essen“);
StringBuffer   buffer = new StringBuffer(zeichenkette);
 // Wenn man nicht selber abzählen will ….
int index = 0;
while( index < buffer.length() ){
  if( buffer.charAt(index) == 'z' ){
    System.out.println("z gefunden an Position " + index);
    buffer.deleteCharAt(index);
    buffer.insert(index, 's');
  }
  index++;
}
System.out.println(buffer.toString());
```

Stringvergleich

Stringvergleich mit *equals*

equals vergleicht die augenblickliche Inhalte der Strings.

```
String name1 = “Harry”; // String name1 = new String(“Harry”);
String name2 = “Hermine”; // String name1 = new String(“Hermine”);
```

Stringvergleich mit *equals*

equals vergleicht die augenblickliche Inhalte der Strings.

```
String name1 = “Harry”; // String name1 = new String(“Harry”);
String name2 = “Hermine”; // String name1 = new String(“Hermine”);
```

```
name1.equals(name2); //wertet zu false aus
```

Stringvergleich mit *equals*

equals vergleicht die augenblickliche Inhalte der Strings.

```
String name1 = “Harry”; // String name1 = new String(“Harry”);
String name2 = “Harry”; // String name1 = new String(“Harry”);
```

```
name1.equals(name2); //wertet zu true aus
```

Stringvergleich mit ==

== Objektvergleich, d.h. Speicherort

```
String name1 = new String(“Harry”);
String name2 = new String (“Harry”) ;
```

```
name1 == name2; //wertet zu false aus
```

Stringvergleich mit ==

ABER...

```
String name1 = “Harry”;
String name2 = “Harry”;

                    name1 == name2; //wertet zu true aus
```

→ Grund ist der *Literal-Pool*

Literal-Pool

- Ein Speicher, der zur Laufzeit String-Literale vorhält
- Jedes Literal wird genau nur einmal im Pool abgelegt
- Wird ein weiteres identisches Literal erzeugt, wird eine Referenz auf das Objekt im Pool erstellt

```
String name1 = “Harry”;
String name2 = “Harry”;
```

Beide Referenzen zeigen auf dasselbe Objekt im Pool

daher: **`name1 == name2;`** `//wertet zu true aus`

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

www.htw-berlin.de