

Programmieren II: Java

Zeichen und Zeichenketten

Prof. Dr. Christopher Auer

Sommersemester 2024



Zeichen: `char`

Zeichenketten: `String`

Inhalt

Zeichen: `char`

Primitiver Typ `char`

Die Klasse `Character`

Inhalt

Zeichen: char

Primitiver Typ char

Primitiver Typ char: Wiederholung

► Zur Erinnerung

Primitiver Typ char: Wiederholung

- ▶ Zur Erinnerung
 - ▶ **char**: Primitiver Typ für Zeichen

Primitiver Typ char: Wiederholung

- ▶ Zur Erinnerung
 - ▶ **char**: Primitiver Typ für Zeichen
 - ▶ Literale

Primitiver Typ char: Wiederholung

- ▶ Zur Erinnerung
 - ▶ **char**: Primitiver Typ für Zeichen
 - ▶ Literale
 - ▶ 'a', 'b', '?', 'X', 'Y', ''

Primitiver Typ char: Wiederholung

- ▶ Zur Erinnerung
 - ▶ **char**: Primitiver Typ für Zeichen
 - ▶ Literale
 - ▶ 'a', 'b', '?', 'X', 'Y', ''
 - ▶ Escape-Sequenzen: '\n', '\t', '\r', ...

Primitiver Typ char: Wiederholung

- ▶ Zur Erinnerung
 - ▶ **char**: Primitiver Typ für Zeichen
 - ▶ **Literale**
 - ▶ 'a', 'b', '?', 'X', 'Y', ''
 - ▶ **Escape-Sequenzen**: '\n', '\t', '\r', ...
 - ▶ **Unicode** (zwei Bytes): '\u0000', '\u03B1', (α)

Primitiver Typ char: Wiederholung

► Zur Erinnerung

- **char**: Primitiver Typ für Zeichen

- Literale

 - 'a', 'b', '?', 'X', 'Y', ''

 - Escape-Sequenzen: '\n', '\t', '\r', ...

 - Unicode (zwei Bytes): '\u0000', '\u03B1', (α)

- Impliziter Widening-Cast nach **int** (und größer)

```
char alpha = '\u03B1';  
int i = alpha; // == 945 == 0x03B1
```

Primitiver Typ char: Wiederholung

► Zur Erinnerung

- **char**: Primitiver Typ für Zeichen

- **Literale**

 - 'a', 'b', '?', 'X', 'Y', ''

 - **Escape-Sequenzen**: '\n', '\t', '\r', ...

 - **Unicode** (zwei Bytes): '\u0000', '\u03B1', (α)

- **Impliziter Widening-Cast** nach **int** (und größer)

```
char alpha = '\u03B1';  
int i = alpha; // == 945 == 0x03B1
```

- **int**-Wert entspricht Index in **Encoding-Tabelle** (s. unten)

Primitiver Typ char: Wiederholung

► Zur Erinnerung

- **char**: Primitiver Typ für Zeichen

- **Literale**

 - 'a', 'b', '?', 'X', 'Y', ''

 - **Escape-Sequenzen**: '\n', '\t', '\r', ...

 - **Unicode** (zwei Bytes): '\u0000', '\u03B1', (α)

- **Impliziter Widening-Cast** nach **int** (und größer)

```
char alpha = '\u03B1';  
int i = alpha; // == 945 == 0x03B1
```


- **int**-Wert entspricht Index in **Encoding-Tabelle** (s. unten)

- **Expliziter Narrowing-Cast** nach **char** (und größer)

```
int i = 0x03B1;  
char alpha = i; // FEHLER  
char alpha = (char) i; // OK
```

Weitere Operationen mit char


- **Ausdrücke:** `char` wird automatisch zu `int` promotet

```
9   runCharPromotionExample  
10 out.println('a' + 1); // 98  
11 out.println('b' / 2); // 49  
12 out.println('b' > 'a'); // true
```

 CharExpressions.java

Weitere Operationen mit char

- ▶ **Ausdrücke:** `char` wird automatisch zu `int` **promotet**


```
9   runCharPromotionExample  
10 out.println('a' + 1); // 98  
11 out.println('b' / 2); // 49  
12 out.println('b' > 'a'); // true
```

 CharExpressions.java

- ▶ Was heißt `'a' + 1`?

Weitere Operationen mit char

- ▶ **Ausdrücke:** `char` wird automatisch zu `int` **promotet**


```
9   runCharPromotionExample  
10 out.println('a' + 1); // 98  
11 out.println('b' / 2); // 49  
12 out.println('b' > 'a'); // true
```

 CharExpressions.java

- ▶ Was heißt `'a' + 1`?
 - ▶ `'a'` wird in `int` konvertiert \Rightarrow 97

Weitere Operationen mit char

- ▶ **Ausdrücke:** `char` wird automatisch zu `int` promotet


```
9   runCharPromotionExample  
10 out.println('a' + 1); // 98  
11 out.println('b' / 2); // 49  
12 out.println('b' > 'a'); // true
```

 CharExpressions.java

- ▶ Was heißt `'a' + 1`?
 - ▶ `'a'` wird in `int` konvertiert \Rightarrow 97
 - ▶ `'a'+1 = 98` (`== (int)'b'`)

Weitere Operationen mit char

- ▶ **Ausdrücke:** `char` wird automatisch zu `int` **promotet**


```
9   runCharPromotionExample  
10 out.println('a' + 1); // 98  
11 out.println('b' / 2); // 49  
12 out.println('b' > 'a'); // true
```

 CharExpressions.java

- ▶ Was heißt `'a' + 1`?
 - ▶ `'a'` wird in `int` konvertiert \Rightarrow 97
 - ▶ `'a'+1 = 98` (`== (int)'b'`)
 - ▶ `'b'` ist nächstes Zeichen nach `'a'` in der **Unicode-Tabelle**

Weitere Operationen mit char

- ▶ **Ausdrücke:** `char` wird automatisch zu `int` **promotet**


```
9   runCharPromotionExample  
10 out.println('a' + 1); // 98  
11 out.println('b' / 2); // 49  
12 out.println('b' > 'a'); // true
```

 CharExpressions.java

- ▶ Was heißt `'a' + 1`?
 - ▶ `'a'` wird in `int` konvertiert \Rightarrow 97
 - ▶ `'a'+1 = 98` (`== (int)'b'`)
 - ▶ `'b'` ist nächstes Zeichen nach `'a'` in der **Unicode-Tabelle**
- ▶ Was heißt `'b' > 'a'`?

Weitere Operationen mit char

- ▶ **Ausdrücke:** `char` wird automatisch zu `int` **promotet**


```
9   runCharPromotionExample  
10 out.println('a' + 1); // 98  
11 out.println('b' / 2); // 49  
12 out.println('b' > 'a'); // true
```

 CharExpressions.java

- ▶ Was heißt `'a' + 1`?
 - ▶ `'a'` wird in `int` konvertiert \Rightarrow 97
 - ▶ `'a'+1 = 98` (`== (int)'b'`)
 - ▶ `'b'` ist nächstes Zeichen nach `'a'` in der **Unicode-Tabelle**
- ▶ Was heißt `'b' > 'a'`?
 - ▶ `(int)'a' == 97`, `(int)'b' == 98` und damit `'b' > 'a' == true`

Weitere Operationen mit char

- ▶ **Ausdrücke:** `char` wird automatisch zu `int` **promotet**


```
9   runCharPromotionExample  
10 out.println('a' + 1); // 98  
11 out.println('b' / 2); // 49  
12 out.println('b' > 'a'); // true
```

 CharExpressions.java

- ▶ Was heißt `'a' + 1`?
 - ▶ `'a'` wird in `int` konvertiert \Rightarrow 97
 - ▶ `'a'+1 = 98` (`== (int)'b'`)
 - ▶ `'b'` ist nächstes Zeichen nach `'a'` in der **Unicode-Tabelle**
- ▶ Was heißt `'b' > 'a'`?
 - ▶ `(int)'a' == 97`, `(int)'b' == 98` und damit `'b' > 'a' == true`
 - ▶ **Vergleich von Positionen** in Unicode-Tabelle

Weitere Operationen mit char

- ▶ **Ausdrücke:** `char` wird automatisch zu `int` **promotet**


```
9   runCharPromotionExample  
10 out.println('a' + 1); // 98  
11 out.println('b' / 2); // 49  
12 out.println('b' > 'a'); // true
```

 CharExpressions.java

- ▶ Was heißt `'a' + 1`?
 - ▶ `'a'` wird in `int` konvertiert \Rightarrow 97
 - ▶ `'a'+1 = 98` (`== (int)'b'`)
 - ▶ `'b'` ist nächstes Zeichen nach `'a'` in der **Unicode-Tabelle**
- ▶ Was heißt `'b' > 'a'`?
 - ▶ `(int)'a' == 97`, `(int)'b' == 98` und damit `'b' > 'a' == true`
 - ▶ Vergleich von Positionen in Unicode-Tabelle
- ▶ Siehe auch <https://unicode-table.com/>

Vergleich von chars


- **Praktisch:** Positionen entsprechen alphabetischer Ordnung

```
18  runCharComparisonExample  
19 out.println('b' < 'd'); // true  
20 out.println('K' < 'M'); // true  
21 out.println('0' < '5'); // true  
22 out.println('Ä' < 'B'); // false  
23 out.println('a' < 'A'); // false
```

 CharExpressions.java

Vergleich von chars

- ▶ **Praktisch:** Positionen entsprechen alphabetischer Ordnung


```
18  runCharComparisonExample  
19 out.println('b' < 'd'); // true  
20 out.println('K' < 'M'); // true  
21 out.println('0' < '5'); // true  
22 out.println('Ä' < 'B'); // false  
23 out.println('a' < 'A'); // false
```

 CharExpressions.java

- ▶ **Vorsicht**

Vergleich von chars

- ▶ **Praktisch:** Positionen entsprechen alphabetischer Ordnung


```
18  runCharComparisonExample  
19 out.println('b' < 'd'); // true  
20 out.println('K' < 'M'); // true  
21 out.println('0' < '5'); // true  
22 out.println('Ä' < 'B'); // false  
23 out.println('a' < 'A'); // false
```

 CharExpressions.java

- ▶ **Vorsicht**
 - ▶ Ordnung nur innerhalb von **Groß- und Kleinbuchstaben** oder **Ziffern**

Vergleich von chars

- ▶ **Praktisch:** Positionen entsprechen alphabetischer Ordnung

```
18  runCharComparisonExample  
19 out.println('b' < 'd'); // true  
20 out.println('K' < 'M'); // true  
21 out.println('0' < '5'); // true  
22 out.println('Ä' < 'B'); // false  
23 out.println('a' < 'A'); // false
```

 CharExpressions.java

- ▶ **Vorsicht**
 - ▶ Ordnung nur innerhalb von **Groß- und Kleinbuchstaben** oder **Ziffern**
 - ▶ Nicht für **Umlaute**

Inhalt

Zeichen: `char`

Die Klasse `Character`

Die Klasse Character

► Klasse [↗](#) Character


Die Klasse `Character`


- ▶ Klasse `Character`
 - ▶ Hilfsmethoden für den Typ `char`

Die Klasse `Character`

- ▶ Klasse `Character`
 - ▶ Hilfsmethoden für den Typ `char`
 - ▶ Beispiele


Die Klasse Character


- ▶ Klasse  Character
 - ▶ Hilfsmethoden für den Typ **char**
 - ▶ Beispiele
 - ▶ isUpperCase/isLowerCase

```
29  runCharUpperLowerCaseExample  
30 out.println(Character.isUpperCase('A')); //true  
31 out.println(Character.isLowerCase('A')); //false  
32 out.println(Character.isUpperCase('Ä')); //true  
33 out.println(Character.isUpperCase('Ø')); //false  
34 out.println(Character.isLowerCase('Ø')); //false
```

 CharExpressions.java


Die Klasse Character

- ▶ Klasse  Character
 - ▶ Hilfsmethoden für den Typ char
 - ▶ Beispiele
 - ▶ isUpperCase/isLowerCase

```
29  runCharUpperLowerCaseExample  
30 out.println(Character.isUpperCase('A')); //true  
31 out.println(Character.isLowerCase('A')); //false  
32 out.println(Character.isUpperCase('Ä')); //true  
33 out.println(Character.isUpperCase('0')); //false  
34 out.println(Character.isLowerCase('0')); //false
```

 CharExpressions.java

- ▶ isDigit

```
40  runCharIsDigitExample  
41 out.println(Character.isDigit('5')); //true  
42 out.println(Character.isDigit('F')); //false
```

 CharExpressions.java

Funktioniert auch für Ziffern anderer Sprachen


Die Klasse Character

► Beispiele

Die Klasse Character

► Beispiele

► isWhitespace


```
48  runCharIsWhitespaceExample  
49 out.println(Character.isWhitespace(' '));// true  
50 out.println(Character.isWhitespace('\t'));// true  
51 out.println(Character.isWhitespace('\n'));// true  
52 out.println(Character.isWhitespace('_'));// false
```

 CharExpressions.java

Die Klasse Character


► Beispiele

► isWhitespace

```
48  runCharIsWhitespaceExample  
49 out.println(Character.isWhitespace(' '));// true  
50 out.println(Character.isWhitespace('\t'));// true  
51 out.println(Character.isWhitespace('\n'));// true  
52 out.println(Character.isWhitespace('_'));// false
```

 CharExpressions.java

► toUpperCase/toLowerCase


```
58  runCharToUpperCaseLowerCaseExample  
59 out.println(Character.toUpperCase('a'));// 'A'  
60 out.println(Character.toUpperCase('A'));// 'A'  
61 out.println(Character.toLowerCase('A'));// 'a'  
62 out.println(Character.toUpperCase('1'));// '1'
```

 CharExpressions.java

Die Klasse Character


► Beispiele

► isWhitespace

```
48  runCharIsWhitespaceExample  
49 out.println(Character.isWhitespace(' '));// true  
50 out.println(Character.isWhitespace('\t'));// true  
51 out.println(Character.isWhitespace('\n'));// true  
52 out.println(Character.isWhitespace('_'));// false
```

 CharExpressions.java

► toUpperCase/toLowerCase

```
58  runCharToUpperCaseLowerCaseExample  
59 out.println(Character.toUpperCase('a'));// 'A'  
60 out.println(Character.toUpperCase('A'));// 'A'  
61 out.println(Character.toLowerCase('A'));// 'a'  
62 out.println(Character.toUpperCase('1'));// '1'
```


 CharExpressions.java

► Und noch viele mehr

Die Klasse Character


► Beispiele

► isWhitespace

```
48  runCharIsWhitespaceExample  
49 out.println(Character.isWhitespace(' '));// true  
50 out.println(Character.isWhitespace('\t'));// true  
51 out.println(Character.isWhitespace('\n'));// true  
52 out.println(Character.isWhitespace('_'));// false
```

 CharExpressions.java

► toUpperCase/toLowerCase

```
58  runCharToUpperCaseLowerCaseExample  
59 out.println(Character.toUpperCase('a'));// 'A'  
60 out.println(Character.toUpperCase('A'));// 'A'  
61 out.println(Character.toLowerCase('A'));// 'a'  
62 out.println(Character.toUpperCase('1'));// '1'
```

 CharExpressions.java

► Und noch viele mehr

► Diese Methoden statt direkte Vergleiche von chars nutzen!

Inhalt

Zeichenketten: **String**

String != char[]

Strings in Java

Konkatenation

Konversion in String

Konversion von String

Die Klasse String

Die Klasse StringBuilder

Inhalt

Zeichenketten: **String**

String != char[]

Vergleich mit C-Strings

- ▶ Zur Erinnerung: Zeichenketten in C

```
char[] s = "Hello World!";
```


Vergleich mit C-Strings

- ▶ Zur Erinnerung: Zeichenketten in C

```
char[] s = "Hello World!";
```

- ▶ **char**-Array

Vergleich mit C-Strings

- ▶ Zur Erinnerung: Zeichenketten in C

```
char[] s = "Hello World!";
```

- ▶ **char**-Array
- ▶ Letztes Zeichen: `'\0'`

Vergleich mit C-Strings

- ▶ Zur Erinnerung: Zeichenketten in C

```
char[] s = "Hello World!";
```

- ▶ **char**-Array
- ▶ Letztes Zeichen: `'\0'`
- ▶ Länge ermitteln: Array durchlaufen bis `'\0'` erreicht

Vergleich mit C-Strings

- ▶ Zur Erinnerung: Zeichenketten in C

```
char[] s = "Hello World!";
```

- ▶ **char**-Array
- ▶ Letztes Zeichen: `'\0'`
- ▶ Länge ermitteln: Array durchlaufen bis `'\0'` erreicht
- ▶ Veränderlich: `s[11] = '?'`

Vergleich mit C-Strings

- ▶ Zur Erinnerung: Zeichenketten in C

```
char[] s = "Hello World!";
```

- ▶ **char**-Array
 - ▶ Letztes Zeichen: `'\0'`
 - ▶ Länge ermitteln: Array durchlaufen bis `'\0'` erreicht
 - ▶ Veränderlich: `s[11] = '?'`
- ▶ In Java

```
String s = "Hello World!";
```

Vergleich mit C-Strings

- ▶ Zur Erinnerung: Zeichenketten in C

```
char[] s = "Hello World!";
```

- ▶ **char**-Array
 - ▶ Letztes Zeichen: `'\0'`
 - ▶ Länge ermitteln: Array durchlaufen bis `'\0'` erreicht
 - ▶ Veränderlich: `s[11] = '?'`
- ▶ In Java

```
String s = "Hello World!";
```

- ▶ Objekt der (besonderen) Klasse `String`

Vergleich mit C-Strings

- ▶ Zur Erinnerung: Zeichenketten in C

```
char[] s = "Hello World!";
```

- ▶ **char**-Array
 - ▶ Letztes Zeichen: `'\0'`
 - ▶ Länge ermitteln: Array durchlaufen bis `'\0'` erreicht
 - ▶ Veränderlich: `s[11] = '?'`
- ▶ In Java

```
String s = "Hello World!";
```

- ▶ Objekt der (besonderen) Klasse `String`
- ▶ Letztes Zeichen: `!` (hier)

Vergleich mit C-Strings

- ▶ Zur Erinnerung: Zeichenketten in C

```
char[] s = "Hello World!";
```

- ▶ **char**-Array
- ▶ Letztes Zeichen: `'\0'`
- ▶ Länge ermitteln: Array durchlaufen bis `'\0'` erreicht
- ▶ Veränderlich: `s[11] = '?'`

- ▶ In Java

```
String s = "Hello World!";
```

- ▶ Objekt der (besonderen) Klasse `String`
- ▶ Letztes Zeichen: `!` (hier)
- ▶ Länge ermitteln: `s.length()`

Vergleich mit C-Strings

- ▶ Zur Erinnerung: Zeichenketten in C

```
char[] s = "Hello World!";
```

- ▶ **char**-Array
 - ▶ Letztes Zeichen: `'\0'`
 - ▶ Länge ermitteln: Array durchlaufen bis `'\0'` erreicht
 - ▶ Veränderlich: `s[11] = '?'`
- ▶ In Java

```
String s = "Hello World!";
```

- ▶ Objekt der (besonderen) Klasse `String`
- ▶ Letztes Zeichen: `!` (hier)
- ▶ Länge ermitteln: `s.length()`
- ▶ Unveränderlich: `String` hat **keine** modifizierenden Methoden

Vergleich mit C-Strings

- ▶ Zur Erinnerung: Zeichenketten in C

```
char[] s = "Hello World!";
```

- ▶ **char**-Array
- ▶ Letztes Zeichen: `'\0'`
- ▶ Länge ermitteln: Array durchlaufen bis `'\0'` erreicht
- ▶ Veränderlich: `s[11] = '?'`

- ▶ In Java

```
String s = "Hello World!";
```

- ▶ Objekt der (besonderen) Klasse `String`
- ▶ Letztes Zeichen: `!` (hier)
- ▶ Länge ermitteln: `s.length()`
- ▶ Unveränderlich: `String` hat **keine** modifizierenden Methoden

- ▶ In Java sind `char[]` und `String` unterschiedliche Dinge

```
char charArray = new char[] { 'H', 'e', 'l', 'l', 'o' }:  
String string = charArray; // FEHLER  
charArray = "Hello"; // FEHLER
```

Zeichenketten: **String** Strings in Java

Die Klasse String

- ▶  String-Objekte beinhalten chars (Container)



Die Klasse String


- ▶ ↗ String-Objekte **beinhalten chars** (**Container**)
- ▶ ↗ String-Objekte können über **Literale** definiert werden

Die Klasse String

- ▶ ↗ String-Objekte **beinhalten chars** (**Container**)
- ▶ ↗ String-Objekte können über **Literale** definiert werden
 - ▶ **Format:** "<chars>"

Die Klasse String

- ▶  String-Objekte **beinhalten chars** (Container)
- ▶  String-Objekte können über **Literale** definiert werden
 - ▶ Format: "<chars>"
 - ▶ Beispiel

```
13  runStringLiteralsExample  
14 out.print("Hello World");  
15 out.print(""); // leerer String  
16 out.print("\nOne\nTwo\nThree\n");  
17 out.print("S\u00FC\u00DF\u00F6lgef\u00E4\u00DF\n");
```

 StringExamples.java

```
Hello World  
One  
Two  
Three  
Süßölgefäß
```

Inhalt

Zeichenketten: **String** Konkatenation




Konkatenation von Strings

- ▶ Operator `+` ist für `String` `s` überladen

Konkatenation von Strings

- ▶ Operator + ist für `String` s überladen
- ▶ Auswertung von links nach rechts

```
23  runStringConcatExample  
24 String truth = "Half the Truth: ";  
25 String all = truth + 20 + 1;  
26 out.println(all);
```

 StringExamples.java

Half the Truth: 201

Konkatenation von Strings

- ▶ Operator + ist für `String` s überladen
- ▶ Auswertung von links nach rechts

23 runStringConcatExample

```
24 String truth = "Half the Truth: ";  
25 String all = truth + 20 + 1;  
26 out.println(all);
```

 StringExamples.java

Half the Truth: 201

- ▶ Besser

32 runStringConcatExample2


```
33 String truth = "Half the Truth: ";  
34 String all = truth + (20 + 1);  
35 out.println(all);
```

 StringExamples.java

Half the Truth: 21

Konkatenation von Strings

- Konkatenation erzeugt **neue** String-Objekte


```
41  runStringConcatExample3  
42 String s1 = "Hello";  
43 String s2 = "World";  
44 String s3 = s1 + " " + s2;  
45 out.printf("s1 == s3 = %b%n", s1 == s3);
```

 StringExamples.java

```
s1 == s3 = false
```

Konkatenation von Strings

- Konkatenation erzeugt **neue** String-Objekte

```
41  runStringConcatExample3  
42 String s1 = "Hello";  
43 String s2 = "World";  
44 String s3 = s1 + " " + s2;  
45 out.printf("s1 == s3 = %b%n", s1 == s3);
```


 StringExamples.java

```
s1 == s3 = false
```

- Jede Konkatenation verbraucht **Speicherplatz**

Konkatenation von Strings

- ▶ Konkatenation erzeugt **neue** String-Objekte

```
41  runStringConcatExample3  
42 String s1 = "Hello";  
43 String s2 = "World";  
44 String s3 = s1 + " " + s2;  
45 out.printf("s1 == s3 = %b%n", s1 == s3);
```


 StringExamples.java

```
s1 == s3 = false
```

- ▶ Jede Konkatenation verbraucht **Speicherplatz**
- ▶ Bei vielen Konkatenationen brauchen wir eine **Alternative**


Konkatenation von Strings

- ▶ Konkatenation erzeugt **neue** String-Objekte

```
41  runStringConcatExample3  
42 String s1 = "Hello";  
43 String s2 = "World";  
44 String s3 = s1 + " " + s2;  
45 out.printf("s1 == s3 = %b%n", s1 == s3);
```

 StringExamples.java


```
s1 == s3 = false
```

- ▶ Jede Konkatenation verbraucht **Speicherplatz**
- ▶ Bei vielen Konkatenationen brauchen wir eine **Alternative**
- ▶ Später:  **StringBuilder**

Zeichenketten: **String** Konversion in String

Referenztypen

► Konkatination: Noch ein Beispiel


```
51  runStringConversionExample  
52 Point2D p = new Point2D(1,2);  
53 String s = "p = " + p;  
54 out.println(s);
```

 StringExamples.java

```
p = de.hawlandshut.java1.oop.shapes.Point2D@11f1109b
```

Referenztypen

► Konkatination: Noch ein Beispiel

```
51  runStringConversionExample  
52 Point2D p = new Point2D(1,2);  
53 String s = "p = " + p;  
54 out.println(s);
```


 StringExamples.java


```
p = de.hawlandshut.java1.oop.shapes.Point2D@11f1109b
```

► Was passiert hier?

Referenztypen

- ▶ **Konkatenation:** Noch ein Beispiel

```
51  runStringConversionExample  
52 Point2D p = new Point2D(1,2);  
53 String s = "p = " + p;  
54 out.println(s);
```


 StringExamples.java

```
p = de.hawlandshut.java1.oop.shapes.Point2D@11f1109b
```

- ▶ Was passiert hier?
 - ▶ Impliziter Aufruf von  **String** toString() der Klasse Point2D

Referenztypen

► Konkatination: Noch ein Beispiel

```
51  runStringConversionExample  
52 Point2D p = new Point2D(1,2);  
53 String s = "p = " + p;  
54 out.println(s);
```

 StringExamples.java


```
p = de.hawlandshut.java1.oop.shapes.Point2D@11f1109b
```

► Was passiert hier?

- Impliziter Aufruf von  `String toString()` der Klasse `Point2D`
- Jede Klasse erbt Default-Implementierung in  `Object.toString` (s. oben)

Referenztypen



► Konkatination: Noch ein Beispiel

```
51  runStringConversionExample
52 Point2D p = new Point2D(1,2);
53 String s = "p = " + p;
54 out.println(s);
```

 StringExamples.java

```
p = de.hawlandshut.java1.oop.shapes.Point2D@11f1109b
```

► Was passiert hier?

- Impliziter Aufruf von  `String toString()` der Klasse `Point2D`
- Jede Klasse erbt **Default-Implementierung** in  `Object.toString` (s. oben)
- D.h. obige Konkatination ist **äquivalent** zu

```
String s = "p = " + p.toString();
```

► Überschreiben von toString in Point2D

```
@Override  
public String toString(){  
    return String.format("Point2D: { x = %d, y = %d }", x, y);  
}
```

Referenztypen

- **Überschreiben** von toString in Point2D

```
@Override  
public String toString(){  
    return String.format("Point2D: { x = %d, y = %d }", x, y);  
}
```

- **Ergebnis**

```
p = Point2D: { x = 1, y = 2 }
```

Referenztypen

- **Überschreiben** von toString in Point2D

```
@Override  
public String toString(){  
    return String.format("Point2D: { x = %d, y = %d }", x, y);  
}
```

- **Ergebnis**

```
p = Point2D: { x = 1, y = 2 }
```

- ([↗ String.format\(\)](#) — s. unten)

Referenztypen

- **Überschreiben** von toString in Point2D

```
@Override  
public String toString(){  
    return String.format("Point2D: { x = %d, y = %d }", x, y);  
}
```

- **Ergebnis**

```
p = Point2D: { x = 1, y = 2 }
```

- ([↗ String.format\(\)](#) — s. unten)
- **Allgemeine Konvention** in Java

Referenztypen

- ▶ **Überschreiben** von `toString` in `Point2D`

```
@Override
public String toString(){
    return String.format("Point2D: { x = %d, y = %d }", x, y);
}
```

- ▶ **Ergebnis**

```
p = Point2D: { x = 1, y = 2 }
```

- ▶ ([↗](#) `String.format()` — s. unten)
- ▶ **Allgemeine Konvention** in Java
 - ▶ **String-Repräsentation** eines Objekts über `toString`

Referenztypen

- **Überschreiben** von `toString` in `Point2D`

```
@Override
public String toString(){
    return String.format("Point2D: { x = %d, y = %d }", x, y);
}
```

- **Ergebnis**


```
p = Point2D: { x = 1, y = 2 }
```


- ([↗](#) `String.format()` — s. unten)
- **Allgemeine Konvention in Java**
 - **String-Repräsentation** eines Objekts über `toString`
 - **Beispiel:** [↗](#) `PrintStream.println(Object obj)` entspricht

```
println(obj.toString());
```

Primitive Typen

- ▶ Implizite Konversion auch bei primitiven Typen


```
60  runStringConversionExample2
61 out.println("PI = " + Math.PI);
62 out.println("Truth/2 = " + 21);
63 out.println(true + " that!");
```

 StringExamples.java

```
PI = 3.141592653589793
Truth/2 = 21
true that!
```

Primitive Typen

- ▶ Implizite Konversion auch bei primitiven Typen

```
60  runStringConversionExample2
61 out.println("PI = " + Math.PI);
62 out.println("Truth/2 = " + 21);
63 out.println(true + " that!");
```


 StringExamples.java

```
PI = 3.141592653589793
Truth/2 = 21
true that!
```

- ▶ Aber hier kein toString() möglich!

Primitive Typen

- ▶ Implizite Konversion auch bei primitiven Typen

```
60  runStringConversionExample2
61 out.println("PI = " + Math.PI);
62 out.println("Truth/2 = " + 21);
63 out.println(true + " that!");
```


 StringExamples.java

```
PI = 3.141592653589793
Truth/2 = 21
true that!
```

- ▶ Aber hier kein toString() möglich!
- ▶ Stattdessen Konversion über Utility-Klassen

Primitive Typen

- ▶ Implizite Konversion auch bei **primitiven Typen**

```
60  runStringConversionExample2  
61 out.println("PI = " + Math.PI);  
62 out.println("Truth/2 = " + 21);  
63 out.println(true + " that!");
```


 StringExamples.java

```
PI = 3.141592653589793  
Truth/2 = 21  
true that!
```

- ▶ **Aber** hier kein toString() möglich!
- ▶ Stattdessen Konversion über **Utility-Klassen**
 - ▶ **static** String Double.toString(**double**)

Primitive Typen

- ▶ Implizite Konversion auch bei primitiven Typen

```
60  runStringConversionExample2
61 out.println("PI = " + Math.PI);
62 out.println("Truth/2 = " + 21);
63 out.println(true + " that!");
```


 StringExamples.java

```
PI = 3.141592653589793
Truth/2 = 21
true that!
```

- ▶ Aber hier kein toString() möglich!
- ▶ Stattdessen Konversion über Utility-Klassen
 - ▶ **static** String Double.toString(double)
 - ▶ **static** String Integer.toString(int)

Primitive Typen

- ▶ Implizite Konversion auch bei **primitiven Typen**

```
60  runStringConversionExample2  
61 out.println("PI = " + Math.PI);  
62 out.println("Truth/2 = " + 21);  
63 out.println(true + " that!");
```


 StringExamples.java

```
PI = 3.141592653589793  
Truth/2 = 21  
true that!
```

- ▶ **Aber** hier kein toString() möglich!
- ▶ Stattdessen Konversion über **Utility-Klassen**
 - ▶ **static** String Double.toString(**double**)
 - ▶ **static** String Integer.toString(**int**)
 - ▶ **static** String Boolean.toString(**boolean**)

Primitive Typen

- ▶ Implizite Konversion auch bei **primitiven Typen**

```
60  runStringConversionExample2  
61 out.println("PI = " + Math.PI);  
62 out.println("Truth/2 = " + 21);  
63 out.println(true + " that!");
```

 StringExamples.java

```
PI = 3.141592653589793  
Truth/2 = 21  
true that!
```

- ▶ **Aber** hier kein toString() möglich!
- ▶ Stattdessen Konversion über **Utility-Klassen**
 - ▶ **static** String Double.toString(**double**)
 - ▶ **static** String Integer.toString(**int**)
 - ▶ **static** String Boolean.toString(**boolean**)
 - ▶ ...

String.format

- ▶ Oft mehr **Steuerungsmöglichkeiten** bei Konversion in `String` gewünscht

String.format

- ▶ Oft mehr **Steuerungsmöglichkeiten** bei Konversion in [↗ String](#) gewünscht
- ▶ [↗ String](#) `String.format(String format, Object... args)`

String.format

- ▶ Oft mehr **Steuerungsmöglichkeiten** bei Konversion in [↗ String](#) gewünscht
- ▶ [↗ String](#) `String.format(String format, Object... args)`
 - ▶ `format`: Format-String

String.format

- ▶ Oft mehr **Steuerungsmöglichkeiten** bei Konversion in `String` gewünscht
- ▶ `String` `String.format(String format, Object... args)`
 - ▶ `format`: Format-String
 - ▶ `args`: Argumente

String.format


- ▶ Oft mehr **Steuerungsmöglichkeiten** bei Konversion in [↗ String](#) gewünscht
- ▶ [↗ String](#) `String.format(String format, Object... args)`
 - ▶ **format**: Format-String
 - ▶ **args**: Argumente
 - ▶ Setzt Argumente in Format-String gemäß **Formatanweisungen** ein

String.format

- ▶ Oft mehr **Steuerungsmöglichkeiten** bei Konversion in [↗ String](#) gewünscht
- ▶ [↗ String](#) `String.format(String format, Object... args)`
 - ▶ **format**: Format-String
 - ▶ **args**: Argumente
 - ▶ Setzt Argumente in Format-String gemäß **Formatanweisungen** ein
- ▶ Ähnlich zu [↗ PrintStream.printf](#) (vgl. Folien zu `printf` in Grundlagen-Kapitel)

String.format

- ▶ Oft mehr **Steuerungsmöglichkeiten** bei Konversion in [↗ String](#) gewünscht
- ▶ [↗ String](#) `String.format(String format, Object... args)`
 - ▶ **format**: Format-String
 - ▶ **args**: Argumente
 - ▶ Setzt Argumente in Format-String gemäß **Formatanweisungen** ein
- ▶ Ähnlich zu [↗ PrintStream.printf](#) (vgl. Folien zu `printf` in Grundlagen-Kapitel)
- ▶ Beispiel

```
78  runStringFormatExample  
79 String s = String.format("%.2f, %b, %d, %x",  
80     Math.PI, true, 42, 42);  
81 out.println(s);
```

 StringExamples.java

3,14, true, 42, 2a

Zeichenketten: **String** Konversion von String

Konversion primitiver Typen in String

- ▶ Oft: Benutzereingabe als `String`

Konversion primitiver Typen in String

- ▶ Oft: Benutzereingabe als `String`
- ▶ Konversion in `int`, `double`, (seltener `boolean`)

Konversion primitiver Typen in String

- ▶ Oft: Benutzereingabe als `String`
- ▶ Konversion in `int`, `double`, (seltener `boolean`)
- ▶ Utility-Klassen:

Konversion primitiver Typen in String

- ▶ Oft: Benutzereingabe als `String`
- ▶ Konversion in `int`, `double`, (seltener `boolean`)
- ▶ Utility-Klassen:
 - ▶ `Integer.parseInt(String s)`

Konversion primitiver Typen in String

- ▶ Oft: Benutzereingabe als `String`
- ▶ Konversion in `int`, `double`, (seltener `boolean`)
- ▶ Utility-Klassen:
 - ▶ `Integer.parseInt(String s)`
 - ▶ `Double.parseDouble(String s)`

Konversion primitiver Typen in String


- ▶ Oft: Benutzereingabe als `String`
- ▶ Konversion in `int`, `double`, (seltener `boolean`)
- ▶ Utility-Klassen:
 - ▶ `Integer.parseInt(String s)`
 - ▶ `Double.parseDouble(String s)`
 - ▶ `Boolean.parseBoolean(String s)`

Konversion primitiver Typen in String

- ▶ Oft: Benutzereingabe als `String`
- ▶ Konversion in `int`, `double`, (seltener `boolean`)
- ▶ Utility-Klassen:
 - ▶ `Integer.parseInt(String s)`
 - ▶ `Double.parseDouble(String s)`
 - ▶ `Boolean.parseBoolean(String s)`
 - ▶ ...

Konversion primitiver Typen in String

- ▶ Oft: Benutzereingabe als `String`
- ▶ Konversion in `int`, `double`, (seltener `boolean`)
- ▶ Utility-Klassen:
 - ▶ `Integer.parseInt(String s)`
 - ▶ `Double.parseDouble(String s)`
 - ▶ `Boolean.parseBoolean(String s)`
 - ▶ ...
- ▶ Beispiel

```
88  runStringParseIntDoubleExample  
89 String stringX = scanner.next();  
90 int x = Integer.parseInt(stringX);  
92 String stringY = scanner.next();  
93 double y = Double.parseDouble(stringY);  
95 out.printf("%d * %f = %f", x, y, x * y);
```

 StringExamples.java

Konversion primitiver Typen in String

► Aufruf

```
4
```

```
3.1
```

```
4 * 3,100000 = 12,400000
```

Konversion primitiver Typen in String

► Aufruf

```
4  
3.1  
4 * 3,100000 = 12,400000
```

► Sogar Exponentendarstellung

```
2  
2.1e2  
2 * 210,000000 = 420,000000
```

Konversion primitiver Typen in String

► Aufruf

```
4  
3.1  
4 * 3,100000 = 12,400000
```

► Sogar Exponentendarstellung

```
2  
2.1e2  
2 * 210,000000 = 420,000000
```

► `NumberFormatException` bei ungültiger Eingabe

```
3.13  
NumberFormatException
```

```
Hello  
NumberFormatException
```

Inhalt

Zeichenketten: **String**

Die Klasse String

Die Klasse `String`

- ▶ Objekte der Klasse `String` sind **unveränderlich**

Die Klasse String


- ▶ Objekte der Klasse `String` sind **unveränderlich**
- ▶ Vermeintlich „modifizierende“ Methoden erstellen neue `String`s

Die Klasse String

- ▶ Objekte der Klasse `String` sind **unveränderlich**
- ▶ Vermeintlich „modifizierende“ Methoden erstellen neue `String`s
- ▶ Hilfreiche Methoden (Auswahl)

Die Klasse String


- ▶ Objekte der Klasse `String` sind **unveränderlich**
- ▶ Vermeintlich „modifizierende“ Methoden erstellen neue `String` s
- ▶ Hilfreiche Methoden (Auswahl)
 - ▶ `char charAt(int i)` — Zeichen an Stelle `i` (bei 0 beginnend)

```
101  runStringCharAtExample  
102 String s = "YMCA";  
103 for (int i = 0; i < s.length(); i++)  
104     out.println(s.charAt(i));
```

 StringExamples.java


Die Klasse String

- ▶ Objekte der Klasse `String` sind **unveränderlich**
- ▶ Vermeintlich „modifizierende“ Methoden erstellen neue `String`s
- ▶ Hilfreiche Methoden (Auswahl)
 - ▶ `char charAt(int i)` — Zeichen an Stelle `i` (bei 0 beginnend)

```
101  runStringCharAtExample  
102 String s = "YMCA";  
103 for (int i = 0; i < s.length(); i++)  
104     out.println(s.charAt(i));
```

 StringExamples.java

- ▶ `String trim()` — entfernt Whitespaces am Anfang und Ende

```
110  runStringTrimExample  
111 String s = " \t\n\r I need my hair cut!\n \t";  
112 out.println(s.trim());
```

 StringExamples.java

I need my hairs cut!


Die Klasse `String`

► Hilfreiche Methoden

Die Klasse String

► Hilfreiche Methoden

- **int** indexOf(String/**char** x)/**int** lastIndexOf(String/**char** x) — liefert ersten/letzten Index an dem x **beginnt** ist (-1 wenn nicht gefunden)


```
118  runStringIndexOfExample  
119 String s = "Yeah Yeah Yeah!";  
120 out.println(s.indexOf("Yeah")); // 0  
121 out.println(s.lastIndexOf("Yeah")); // 10
```

 StringExamples.java

Die Klasse String

► Hilfreiche Methoden

- **int** `indexOf(String/char x)/int` `lastIndexOf(String/char x)` — liefert ersten/letzten Index an dem x **beginnt** ist (-1 wenn nicht gefunden)

```
118  runStringIndexOfExample  
119 String s = "Yeah Yeah Yeah!";  
120 out.println(s.indexOf("Yeah")); // 0  
121 out.println(s.lastIndexOf("Yeah")); // 10
```


 StringExamples.java

- **boolean** `isEmpty()` — liefert **true** wenn String leer ist (`length()==0`)

Die Klasse String


► Hilfreiche Methoden

- **int** `indexOf(String/char x)/int` `lastIndexOf(String/char x)` — liefert **ersten/letzten** Index an dem x **beginnt** ist (-1 wenn nicht gefunden)

```
118  runStringIndexOfExample  
119 String s = "Yeah Yeah Yeah!";  
120 out.println(s.indexOf("Yeah")); // 0  
121 out.println(s.lastIndexOf("Yeah")); // 10
```

 StringExamples.java

- **boolean** `isEmpty()` — liefert **true** wenn String leer ist (`length()==0`)
- **boolean** `isBlank()` — liefert **true** wenn String nur aus **Whitespaces** besteht

```
127  runStringIsBlankExample  
128 String s = " \t\n";  
129 out.println("Empty: " + s.isEmpty());  
130 out.println("Blank: " + s.isBlank());
```

 StringExamples.java

```
Empty: false  
Blank: true
```


Die Klasse String

► Hilfreiche Methoden

Die Klasse String

► Hilfreiche Methoden

- `substring(int from, int to)/substring(int from)` — liefert Substring von Index from bis Index to (ausschließlich)/Ende

```
136  runStringSubstringExample  
137 String s = "All glory to the Hypnotoad!";  
138 out.println(s.substring(17)); // Hypnotoad!  
139 out.println(s.substring(4, 9)); // glory
```

 StringExamples.java

Die Klasse String

► Hilfreiche Methoden

- `substring(int from, int to)/substring(int from)` — liefert Substring von Index from bis Index to (ausschließlich)/Ende

136 **runStringSubstringExample**

```
137 String s = "All glory to the Hypnotoad!";  
138 out.println(s.substring(17)); // Hypnotoad!  
139 out.println(s.substring(4, 9)); // glory
```

 StringExamples.java

- `toUpperCase()/toLowerCase()` — konvertiert alle Zeichen in Groß-/Kleinbuchstaben

145 **runStringToUpperCaseExample**

```
146 String input = "Yes";  
147 if (input.toUpperCase().equals("YES"))  
148     out.println("Agreed!");
```

 StringExamples.java


Die Klasse String


► Hilfreiche Methoden

Die Klasse String

► Hilfreiche Methoden

- `replace(char x, char y)` — ersetzt Vorkommen von x durch y


```
154  runStringReplaceExample  
155 String s = "The CAKE is a lie!";  
156 String s2 = s.replace('A', 'O');  
157 out.println(s2); // The COKE is a lie!
```

 StringExamples.java

Die Klasse String


► Hilfreiche Methoden

- `replace(char x, char y)` — ersetzt Vorkommen von x durch y

```
154  runStringReplaceExample  
155 String s = "The CAKE is a lie!";  
156 String s2 = s.replace('A', 'O');  
157 out.println(s2); // The COKE is a lie!
```

 StringExamples.java

- `startsWith(String x)/endsWith(String x)` — liefert **true** wenn String mit x beginnt/aufhört


```
163  runStringStartsEndsWithExample  
164 String story = "Once upon a time...";  
165 out.println(story.startsWith("Once")); // true  
166 out.println(story.endsWith("...")); // true
```

 StringExamples.java

Die Klasse String


► Hilfreiche Methoden

- `replace(char x, char y)` — ersetzt Vorkommen von x durch y

```
154  runStringReplaceExample  
155 String s = "The CAKE is a lie!";  
156 String s2 = s.replace('A', 'O');  
157 out.println(s2); // The COKE is a lie!
```

 StringExamples.java

- `startsWith(String x)/endsWith(String x)` — liefert **true** wenn String mit x beginnt/aufhört


```
163  runStringStartsEndsWithExample  
164 String story = "Once upon a time...";  
165 out.println(story.startsWith("Once")); // true  
166 out.println(story.endsWith("...")); // true
```

 StringExamples.java

- Viele mehr: `repeat`, `join`, `split`, `lines` ...

Vergleich von Strings

- Schon gesehen: Gleichheit **nicht** mit == prüfen


```
172  runStringIdentityExample  
173 String x = "Cake";  
174 String y = "Coke".replace('o', 'a');  
175 out.printf("%s == %s : %b%n", x, y, x == y);
```

 StringExamples.java

```
Cake == Cake : false
```

Vergleich von Strings


- ▶ Schon gesehen: Gleichheit **nicht** mit == prüfen

```
172  runStringIdentityExample  
173 String x = "Cake";  
174 String y = "Coke".replace('o', 'a');  
175 out.printf("%s == %s : %b%n", x, y, x == y);
```

 StringExamples.java

```
Cake == Cake : false
```

- ▶ Stattdessen mit equals/equalsIgnoreCase auf **Gleichheit** prüfen

```
181  runStringEqualsExample  
182 String x = "Cake";  
183 String y = "Coke".replace('o', 'a');  
184 out.printf("%s equals %s : %b%n", x, y, x.equals(y));
```


 StringExamples.java

```
Cake equals Cake : true
```


Vergleich von Strings

- ▶ Wann kommt ein `String` alphabetisch vor einem anderen?

Vergleich von Strings

- ▶ Wann kommt ein  String alphabetisch vor einem anderen?
- ▶ Lexikographische Ordnung: z.B. $a < aa < ab < b < ba$

Vergleich von Strings

- ▶ Wann kommt ein `String` alphabetisch vor einem anderen?
- ▶ Lexikographische Ordnung: z.B. $a < aa < ab < b < ba$
 - ▶ „Zeichen für Zeichen“-Vergleich bis kürzerer `String`

Vergleich von Strings

- ▶ Wann kommt ein `String` alphabetisch vor einem anderen?
- ▶ Lexikographische Ordnung: z.B. $a < aa < ab < b < ba$
 - ▶ „Zeichen für Zeichen“-Vergleich bis kürzerer `String`
 - ▶ Unterschiedliche **chars**: Rückgabe der Differenz

Vergleich von Strings

- ▶ Wann kommt ein `String` alphabetisch vor einem anderen?
- ▶ Lexikographische Ordnung: z.B. $a < aa < ab < b < ba$
 - ▶ „Zeichen für Zeichen“-Vergleich bis kürzerer `String`
 - ▶ Unterschiedliche **chars**: Rückgabe der Differenz
 - ▶ Gleiche **chars**: kürzerer `String` ist kleiner

Vergleich von Strings

- ▶ Wann kommt ein `String` alphabetisch vor einem anderen?
- ▶ Lexikographische Ordnung: z.B. $a < aa < ab < b < ba$
 - ▶ „Zeichen für Zeichen“-Vergleich bis kürzerer `String`
 - ▶ Unterschiedliche **chars**: Rückgabe der Differenz
 - ▶ Gleiche **chars**: kürzerer `String` ist kleiner
- ▶ `int compareTo(String y)` und `int compareToIgnoreCase(String y)`

Vergleich von Strings

- ▶ Wann kommt ein `String` alphabetisch vor einem anderen?
- ▶ Lexikographische Ordnung: z.B. $a < aa < ab < b < ba$
 - ▶ „Zeichen für Zeichen“-Vergleich bis kürzerer `String`
 - ▶ Unterschiedliche **chars**: Rückgabe der Differenz
 - ▶ Gleiche **chars**: kürzerer `String` ist kleiner
- ▶ `int compareTo(String y)` und `int compareToIgnoreCase(String y)`
- ▶ Rückgabewert: `x.compareTo(y)`

Vergleich von Strings

- ▶ Wann kommt ein `String` alphabetisch vor einem anderen?
- ▶ Lexikographische Ordnung: z.B. $a < aa < ab < b < ba$
 - ▶ „Zeichen für Zeichen“-Vergleich bis kürzerer `String`
 - ▶ Unterschiedliche **chars**: Rückgabe der Differenz
 - ▶ Gleiche **chars**: kürzerer `String` ist kleiner
- ▶ `int compareTo(String y)` und `int compareToIgnoreCase(String y)`
- ▶ Rückgabewert: `x.compareTo(y)`
 - ▶ $< 0 \rightarrow x$ kommt vor y




Vergleich von Strings


- ▶ Wann kommt ein `String` alphabetisch vor einem anderen?
- ▶ Lexikographische Ordnung: z.B. $a < aa < ab < b < ba$
 - ▶ „Zeichen für Zeichen“-Vergleich bis kürzerer `String`
 - ▶ Unterschiedliche **chars**: Rückgabe der Differenz
 - ▶ Gleiche **chars**: kürzerer `String` ist kleiner
- ▶ `int compareTo(String y)` und `int compareToIgnoreCase(String y)`
- ▶ Rückgabewert: `x.compareTo(y)`
 - ▶ $< 0 \rightarrow x$ kommt vor y
 - ▶ $= 0 \rightarrow x$ und y sind wertgleich

Vergleich von Strings

- ▶ Wann kommt ein `String` alphabetisch vor einem anderen?
- ▶ Lexikographische Ordnung: z.B. $a < aa < ab < b < ba$
 - ▶ „Zeichen für Zeichen“-Vergleich bis kürzerer `String`
 - ▶ Unterschiedliche **chars**: Rückgabe der Differenz
 - ▶ Gleiche **chars**: kürzerer `String` ist kleiner
- ▶ `int compareTo(String y)` und `int compareToIgnoreCase(String y)`
- ▶ Rückgabewert: `x.compareTo(y)`
 - ▶ $< 0 \rightarrow x$ kommt vor y
 - ▶ $= 0 \rightarrow x$ und y sind wertgleich
 - ▶ $> 0 \rightarrow x$ kommt nach y

Vergleich von Strings

- ▶ Wann kommt ein  String alphabetisch vor einem anderen?
- ▶ Lexikographische Ordnung: z.B. $a < aa < ab < b < ba$
 - ▶ „Zeichen für Zeichen“-Vergleich bis kürzerer  String
 - ▶ Unterschiedliche chars: Rückgabe der Differenz
 - ▶ Gleiche chars: kürzerer  String ist kleiner
- ▶ `int compareTo(String y)` und `int compareToIgnoreCase(String y)`
- ▶ Rückgabewert: `x.compareTo(y)`
 - ▶ $< 0 \rightarrow x$ kommt vor y
 - ▶ $= 0 \rightarrow x$ und y sind wertgleich
 - ▶ $> 0 \rightarrow x$ kommt nach y
- ▶ Beispiel

```
190  runStringCompareToExample
191 String s = "Auer";
192 out.println(s.compareTo("Aaronson")); // 20
193 out.println(s.compareTo("Zukowski")); // -25
```

 StringExamples.java

Inhalt

Zeichenketten: **String**

Die Klasse `StringBuilder`

Die Klasse `StringBuilder`

- ▶ `String` is unveränderlich

Die Klasse `StringBuilder`

- ▶ `String` is unveränderlich
- ▶ Operationen erzeugen neue `String` s

Die Klasse `StringBuilder`

- ▶ `String` is unveränderlich
- ▶ Operationen erzeugen neue `String`s
- ▶ Problem bei Programmen mit vielen `String`-Operationen

Die Klasse `StringBuilder`

- ▶ `String` is unveränderlich
- ▶ Operationen erzeugen neue `String` s
- ▶ **Problem** bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String` s

Die Klasse `StringBuilder`

- ▶ `String` is unveränderlich
- ▶ Operationen erzeugen neue `String` s
- ▶ **Problem** bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String` s
 - ▶ Laufzeit: `String` s werden oft kopiert (auch bei kleinen Änderungen)

Die Klasse `StringBuilder`

- ▶ `String` is unveränderlich
- ▶ Operationen erzeugen neue `String` s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String` s
 - ▶ Laufzeit: `String` s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`

Die Klasse `StringBuilder`

- ▶ `String` is unveränderlich
- ▶ Operationen erzeugen neue `String` s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String` s
 - ▶ Laufzeit: `String` s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`
 - ▶ „Veränderliche Version“ von `String`

Die Klasse `StringBuilder`

- ▶ `String` is unveränderlich
- ▶ Operationen erzeugen neue `String` s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String` s
 - ▶ Laufzeit: `String` s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`
 - ▶ „Veränderliche Version“ von `String`
 - ▶ Operationen

Die Klasse `StringBuilder`

- ▶ `String` is unveränderlich
- ▶ Operationen erzeugen neue `String` s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String` s
 - ▶ Laufzeit: `String` s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`
 - ▶ „Veränderliche Version“ von `String`
 - ▶ Operationen
 - ▶ Anhängen: `append(String/int/double/...)`

Die Klasse `StringBuilder`

- ▶ `String` ist unveränderlich
- ▶ Operationen erzeugen neue `String`s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String`s
 - ▶ Laufzeit: `String`s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`
 - ▶ „Veränderliche Version“ von `String`
 - ▶ Operationen
 - ▶ Anhängen: `append(String/int/double/...)`
 - ▶ Einfügen: `insert(int offset, String/int/double/...)`

Die Klasse `StringBuilder`

- ▶ `String` ist unveränderlich
- ▶ Operationen erzeugen neue `String`s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String`s
 - ▶ Laufzeit: `String`s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`
 - ▶ „Veränderliche Version“ von `String`
 - ▶ Operationen
 - ▶ Anhängen: `append(String/int/double/...)`
 - ▶ Einfügen: `insert(int offset, String/int/double/...)`
 - ▶ Löschen: `delete(int from, int to)`, `deleteCharAt(int index)`

Die Klasse `StringBuilder`

- ▶ `String` ist unveränderlich
- ▶ Operationen erzeugen neue `String`s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String`s
 - ▶ Laufzeit: `String`s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`
 - ▶ „Veränderliche Version“ von `String`
 - ▶ Operationen
 - ▶ Anhängen: `append(String/int/double/...)`
 - ▶ Einfügen: `insert(int offset, String/int/double/...)`
 - ▶ Löschen: `delete(int from, int to)`, `deleteCharAt(int index)`
 - ▶ Einzelne Zeichen ändern: `setCharAt(int index, char x)`

Die Klasse `StringBuilder`

- ▶ `String` ist unveränderlich
- ▶ Operationen erzeugen neue `String`s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String`s
 - ▶ Laufzeit: `String`s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`
 - ▶ „Veränderliche Version“ von `String`
 - ▶ Operationen
 - ▶ Anhängen: `append(String/int/double/...)`
 - ▶ Einfügen: `insert(int offset, String/int/double/...)`
 - ▶ Löschen: `delete(int from, int to)`, `deleteCharAt(int index)`
 - ▶ Einzelne Zeichen ändern: `setCharAt(int index, char x)`
 - ▶ ...

Die Klasse `StringBuilder`

- ▶ `String` ist unveränderlich
- ▶ Operationen erzeugen neue `String`s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String`s
 - ▶ Laufzeit: `String`s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`
 - ▶ „Veränderliche Version“ von `String`
 - ▶ Operationen
 - ▶ Anhängen: `append(String/int/double/...)`
 - ▶ Einfügen: `insert(int offset, String/int/double/...)`
 - ▶ Löschen: `delete(int from, int to)`, `deleteCharAt(int index)`
 - ▶ Einzelne Zeichen ändern: `setCharAt(int index, char x)`
 - ▶ ...
 - ▶ `StringBuilder` verwaltet intern `char`-Array

Die Klasse `StringBuilder`

- ▶ `String` ist unveränderlich
- ▶ Operationen erzeugen neue `String`s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String`s
 - ▶ Laufzeit: `String`s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`
 - ▶ „Veränderliche Version“ von `String`
 - ▶ Operationen
 - ▶ Anhängen: `append(String/int/double/...)`
 - ▶ Einfügen: `insert(int offset, String/int/double/...)`
 - ▶ Löschen: `delete(int from, int to)`, `deleteCharAt(int index)`
 - ▶ Einzelne Zeichen ändern: `setCharAt(int index, char x)`
 - ▶ ...
 - ▶ `StringBuilder` verwaltet intern `char`-Array
 - ▶ Kapazität: Länge des Arrays (`StringBuilder.capacity()`)

Die Klasse `StringBuilder`

- ▶ `String` ist unveränderlich
- ▶ Operationen erzeugen neue `String`s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String`s
 - ▶ Laufzeit: `String`s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`
 - ▶ „Veränderliche Version“ von `String`
 - ▶ Operationen
 - ▶ Anhängen: `append(String/int/double/...)`
 - ▶ Einfügen: `insert(int offset, String/int/double/...)`
 - ▶ Löschen: `delete(int from, int to)`, `deleteCharAt(int index)`
 - ▶ Einzelne Zeichen ändern: `setCharAt(int index, char x)`
 - ▶ ...
 - ▶ `StringBuilder` verwaltet intern `char`-Array
 - ▶ Kapazität: Länge des Arrays (`StringBuilder.capacity()`)
 - ▶ Array wird bei Bedarf vergrößert (Achtung: Kopiervorgang!)

Die Klasse `StringBuilder`


- ▶ `String` ist unveränderlich
- ▶ Operationen erzeugen neue `String`s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String`s
 - ▶ Laufzeit: `String`s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`
 - ▶ „Veränderliche Version“ von `String`
 - ▶ Operationen
 - ▶ Anhängen: `append(String/int/double/...)`
 - ▶ Einfügen: `insert(int offset, String/int/double/...)`
 - ▶ Löschen: `delete(int from, int to)`, `deleteCharAt(int index)`
 - ▶ Einzelne Zeichen ändern: `setCharAt(int index, char x)`
 - ▶ ...
 - ▶ `StringBuilder` verwaltet intern `char`-Array
 - ▶ Kapazität: Länge des Arrays (`StringBuilder.capacity()`)
 - ▶ Array wird bei Bedarf vergrößert (Achtung: Kopiervorgang!)
 - ▶ Initiale Kapazität kann im Konstruktor angegeben werden

Die Klasse `StringBuilder`

- ▶ `String` ist unveränderlich
- ▶ Operationen erzeugen neue `String`s
- ▶ Problem bei Programmen mit vielen `String`-Operationen
 - ▶ Speicher: `String`-Operationen erzeugen neue `String`s
 - ▶ Laufzeit: `String`s werden oft kopiert (auch bei kleinen Änderungen)
- ▶ Abhilfe: `StringBuilder`
 - ▶ „Veränderliche Version“ von `String`
 - ▶ Operationen
 - ▶ Anhängen: `append(String/int/double/...)`
 - ▶ Einfügen: `insert(int offset, String/int/double/...)`
 - ▶ Löschen: `delete(int from, int to)`, `deleteCharAt(int index)`
 - ▶ Einzelne Zeichen ändern: `setCharAt(int index, char x)`
 - ▶ ...
 - ▶ `StringBuilder` verwaltet intern `char`-Array
 - ▶ Kapazität: Länge des Arrays (`StringBuilder.capacity()`)
 - ▶ Array wird bei Bedarf vergrößert (Achtung: Kopiervorgang!)
 - ▶ Initiale Kapazität kann im Konstruktor angegeben werden
 - ▶ `length` liefert tatsächliche Länge des gebauten `String`s

Beispiel: StringBuilder I

Hinweis: `printInfo` gibt Inhalt, Länge und Kapazität aus

```
12  runStringBuilderExample  
13 StringBuilder builder = new StringBuilder(10);  
15 builder.append("Die Fläche");  
16 printInfo(builder);  
18 builder.append(' ');  
19 printInfo(builder);  
21 builder.append("Kreises");  
22 printInfo(builder);  
24 builder.insert(10, " eines");  
25 printInfo(builder);  
27 builder.append(" mit Radius ");  
28 printInfo(builder);  
30 builder.append(2.3);  
31 printInfo(builder);  
33 builder.append(" ist ");  
34 printInfo(builder);
```

Beispiel: StringBuilder II

```
36 builder.append(2.3 * 2.3 * Math.PI);  
37 printInfo(builder);  
39 builder.delete(25, 40);  
40 printInfo(builder);
```

StringBulderExample.java

```
"Die Fläche" (L:10, C:10)  
"Die Fläche " (L:11, C:22)  
"Die Fläche Kreises" (L:18, C:22)  
"Die Fläche eines Kreises" (L:24, C:46)  
"Die Fläche eines Kreises mit Radius " (L:36, C:46)  
"Die Fläche eines Kreises mit Radius 2.3" (L:39, C:46)  
"Die Fläche eines Kreises mit Radius 2.3 ist " (L:44, C:46)  
"Die Fläche eines Kreises mit Radius 2.3 ist 16.619025137490002"  
  (L:62, C:94)  
"Die Fläche eines Kreises ist 16.619025137490002" (L:47, C:94)
```