



## **PM2 Java: Zeichenketten**



# Fahrplan

- Zeichenketten
  - Aufbau einer Zeichenkette
  - Literale
  - Immutables und Wertsemantik
  - Elementzugriff
  - Methoden für Strings
  - Besonderheiten der Klasse *String*
  - Methode toString()
  - Klasse *StringBuilder*
  - String Formatierung



# Einführung Strings

- Ein String ist eine Folge von Zeichen.
- Ein String ist ein vordefinierter Typ in Java wie *int*, *double* etc, aber ein Referenztyp
- Ein String ist ein „Containertyp“ für den Typ *char*.
- Containertypen
  - speichern Elemente anderer Typen
  - deren Elemente anonym sind
  - deren Elementanzahl nicht begrenzt ist
- Darstellung von *"Java"*

'J'	'a'	'v'	'a'
-----	-----	-----	-----



# String-Literale

- Zeichenfolgen zur direkten Darstellung der Werte des Typs String
- Beispiele:
  - "Java", "Oracle", " ", "",
  - beliebige Zeichenfolge:
    - "a",
    - "zwei-\nzeilig",
    - "M\u00FCnchen",
    - "Im Unicode ein \alpha\: \u03B1",
- Länge eines Strings ist die Anzahl der Zeichen eines Strings



# Immutable und Wertsemantik

- String ist eine unveränderlicher Typ mit Wertsemantik. Strings sind **immutable**.
- **Wertsemantik:** Objekte von Klassen verändern ihre Werte bei Methodenaufrufen nicht. Sie verhalten sich wie Werte der Basisdatentypen.
- Es können keine Zeichen eingefügt, ausgetauscht oder entfernt werden!
- Strings können mit dem "+" Operator konkateniert werden. Das Ergebnis ist eine neuer dritter String. Die Operanden werden nicht verändert!
- "+" kann Operanden aller Basistypen und aller Referenztypen mit String konkatenieren.



# *String* ist immutable

- Nach der Anwendung des **+** Operators auf *str* ist *str* unverändert.
- Modifiziert ist nur *strMod*.

**package** zeichenketten;

```
public class StringImmutable {  
    public static void main(String[] args) {  
        String str = "original";  
        String strMod;  
        strMod = str + " Modified";  
        System.out.println("str " + str);  
        System.out.println("strMod " + strMod);  
    }  
}
```



str original  
strMod original Modified



# Strings sind immutable aber nicht unique

- Strings sind Objekte (Referenztypen).
- Zwei *String* Objekte (*str1*, *str2*) gleichen Inhalts sind nicht identisch, aber inhaltsgleich.
- Inhaltsgleichheit von Objekten wird in Java mit der Methode *equals()*, Identität mittels *==* geprüft.
- *str1 == str2* liefert für inhaltsgleiche Strings *false*
- *str1.equals(str2)* liefert für inhaltsgleiche Strings *true*



# Strings sind immutable aber nicht unique

args ist die Parameterliste von main,  
enthält alle Parameter die beim  
Programmstart übergeben werden.

```
String str1, str2, str3;
str1 = "Hallo";
str2 = args[0];
System.out.println("str2 " + str2);
str3 = "Hallo";
System.out.println("str1 == str2 " + (str1 == str2));
System.out.println("str1 == str3 " + (str1 == str3));
System.out.println("str1.equals(str2) " + str1.equals(str2));
```

str2 Hallo

str1 == str2 false  
str1 == str3 true  
str1.equals(str2) true

str1 und str3 sind nicht identisch  
und dennoch liefert == true. WARUM?

→ Package immutable Klasse StringImmutableNotUnique



# Einige inhaltsgleiche Strings sind auch identisch?

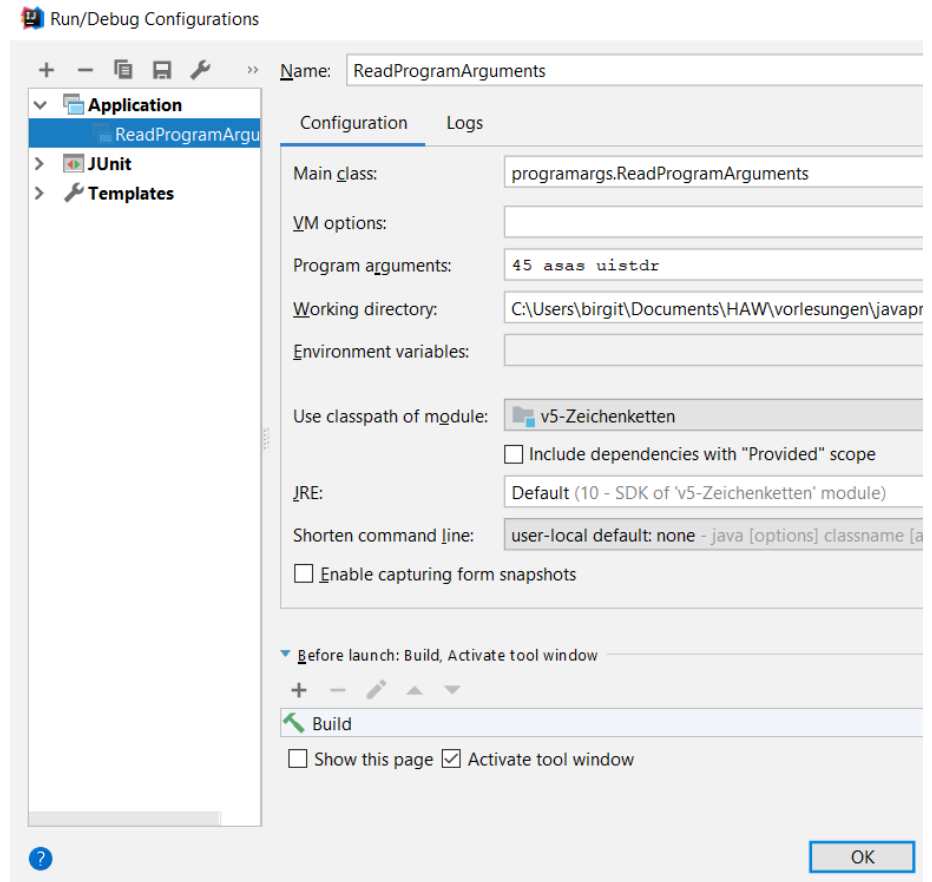


- Compiler machen aus Platzgründen inhaltsgleiche Strings zu identischen Strings. Sie verwenden dazu einen **Stringpool**.
- **Effekt:** inhaltsgleiche Strings sind plötzlich identisch ☹.
- **VORSICHT** 💣: nicht alle inhaltsgleichen Strings sind identisch
  - nur gleiche Literale und
  - Strings, die durch „+“ Konkatenation von Stringliteralen in einem Programm entstehen.
- **DAHER:** Prüfe **Gleichheit** von Strings immer mit *equals*.

# EXKURS: Parameterübergabe an Java-Programme in Eclipse



- Selektieren der ausführbaren Klasse rechte Maustaste.
- **Run→Edit Configuration...** öffnet den Dialog rechts (Auszug).
- Unter **Program arguments** Argumente mit Leerzeichen getrennt eingeben
- Die Argumente werden in den ***String[] args*** Parameter (ein String Array) der ***main*** Methode übertragen.
- An Position 0 steht das erste Argument, an Position n-1 das n'te Argument.





# EXKURS: Parameter *String[] args* in *main* auslesen

- Unten ein Beispiel, in dem alle dem Programm übergebenen Argumente ausgelesen und ausgegeben werden.

```
public class ReadProgramArguments {  
    public static void main(String[] args) {  
        for (int i =0; i < args.length; i++) {  
            System.out.println("args[" + i + "]= " + args[i]);  
        }  
    }  
}
```

→ *Package programargs*



# Elementzugriff

- Die Indizierung der Zeichen eines Strings
  - beginnt wie bei Arrays mit der Position 0
  - endet mit der Position (Länge-1).
  - Zugriff mit negativem Index oder einem Index  $\geq$  Länge führt zu einem Laufzeitfehler.
- Die Methode für den Elementzugriff *charAt(int index)* liefert das Zeichen an der Position *index* zurück.



# Elementzugriff

```
public class ElementzugriffDemo {  
    public static void main(String[] args) {  
        String s = "Java";  
        System.out.println(s.charAt(0));  
        System.out.println(s.charAt(-1)); // Laufzeitfehler  
        System.out.println(s.charAt(4)); // Laufzeitfehler  
  
        /*  
        * Ausgabe aller Zeichen eines Strings  
        */  
        for (int i= 0; i < s.length(); i++) {  
            System.out.println(s.charAt(i));  
        }  
    }  
}
```



# Methoden für Strings

*String*

erzeugen

Test auf  
Leerstring

Teile  
suchen

konkatenieren

zerlegen  
→ reguläre  
Ausdrücke

konvertieren

vergleichen

Teile  
extrahieren

Leerzeichen  
entfernen

Groß-  
Kleinschreibung

Teile  
ersetzen



# Strings erzeugen

- Literale wie z.B. *"abc"*
- *new* und einer der Konstruktoren
- Konvertierungsmethoden *valueOf(<Type> val)* von *String*:
  - definiert für die meisten Basisdatentypen, *char* Arrays und *Object*
  - wandeln bei der Konkatination von *String* mit nicht *String*-Objekten letztere in String-Objekte um.
  - rufen bei den Objekttypen die Methode *toString* auf.
- die *toString()* Methode der Objekttypen.

→ Package *stringmethoden* Klasse *StringCreate*



# Strings erzeugen

```
String s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13;  
char[] moinFolks = { 'M','o','i','n',' ','f','o','l','k','s' };  
Person person = new Person("Gundula", "Gause");  
  
p("LITERAL");  
p(s1 = "Moin Folks");           // Literal  
"Moin Folks".length();         // ein Literal ist ein String Objekt  
  
p("KONSTRUKTOR");  
p(s1 = new String("Moin Folks"));  
p(s4 = new String(moinFolks));  
p(s5 = new String(moinFolks, 0, 4));  
  
p("KONVERTIERUNG valueOf(...)");  
p(s6 = String.valueOf(true));           // boolean  
p(s7 = String.valueOf('M'));             // char  
p(s8 = String.valueOf(moinFolks));        // char[]  
p(s9 = String.valueOf(moinFolks, 0, 4));  
p(s10 = String.valueOf(17.982));          // double und float  
p(s11 = String.valueOf(17.789f));  
p(s12 = String.valueOf(1345689));        // int und long  
p(s13 = String.valueOf(person));          // Objects  
  
p("OBJEKTYPEN toString()");  
p(person.toString());
```





# Strings konvertieren

Alle Methoden sind public static	Erläuterung
<b><i>String valueOf(&lt;Type&gt;)</i></b> <Type> = boolean, char, char[], double, float, int, long, Object	Konvertierung von Basisdatentypen, char Arrays, Referenztypen in String
<b><i>String valueOf(char[] cAry,int offs,int cnt)</i></b>	Konvertierung eines Teils von <b><i>cAry</i></b> von <b><i>offs</i></b> bis <b><i>offs +cnt</i></b> in <b><i>String</i></b>
<b><i>String copyValueOf(char[] cAry)</i></b>	vgl. <b><i>valueOf</i></b>
<b><i>String copyValueOf(char[] cAry,int offs,int cnt)</i></b>	vgl. <b><i>valueOf</i></b>
<b><i>new String(&lt;Type&gt;)</i></b> <Type> = char[], String, StringBuilder, StringBuffer	Konvertierung von <b><i>char</i></b> Arrays, Kopie von <b><i>String</i></b> , Umwandlung von <b><i>StringBuffer</i></b> und <b><i>StringBuilder</i></b> in <b><i>String</i></b> über Konstruktoren
<b><i>String(char[] cAry,int offs,int cnt)</i></b>	vgl. <b><i>valueOf</i></b>
<b><i>char[] toCharArray()</i></b>	Umwandlung eines Strings in einen <b><i>char</i></b> Array.



# Groß- und Kleinschreibung

- Transformation in Zeichenketten die nur aus Klein-/ Großbuchstaben bestehen.  
*toLowerCase(), toUpperCase()*
- Die Methoden arbeiten mit der Default Spracheinstellung der Umgebung (German in unserem Fall) . Sie verwandelt z.B. "ß" in "SS".
- **VORSICHT:** Wurde zuvor ein Array auf Basis der Länge des Originalstrings allokiert, dann führt diese Umwandlung zu einem Indexfehler.



# Groß- und Kleinschreibung

```
String s1,s2;  
s1 ="Großschreibung";  
char[] cAry = new char[s1.length()];  
s2 = s1.toUpperCase();  
p(s2);  
p(s2.toLowerCase());  
// mit Locale  
Locale germ = Locale.GERMAN;  
p(s1.toUpperCase(germ));  
p(s1.toUpperCase(germ).toLowerCase(germ));  
// Laufzeitfehler ArrayIndexOutOfBoundsException  
s2.getChars(0, s2.length(), cAry, 0);
```



GROSSSCHREIBUNG  
grossschreibung

GROSSSCHREIBUNG  
Grossschreibung



Exception in thread "main"

java.lang.ArrayIndexOutOfBoundsException

at java.lang.System.arraycopy(Native Method)

at java.lang.String.getChars(String.java:854)

at stringmethoden.StringLowerUpper.main(StringLowerUpper.java:19)

→ Package stringmethoden Klasse StringLowerUpper



# Test auf Leerstring

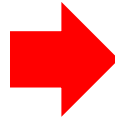
- Der Leerstring ist definiert als der String mit der Länge *0*.
- Statt Test auf Länge bietet die Klasse String die Methode *isEmpty()* an.
- **VORSICHT:** *null* ist kein Leerstring.
- Auch wenn jede Variable eines Objekttyps den Wert *null* annehmen kann, erzeugen die Zugriffe *sNull.length* als auch *sNull.isEmpty()* im nachfolgenden Beispiel einen Laufzeitfehler (*NullPointerException*).

→ Package stringmethoden Klasse EmptyString



# Test auf Leerstring

```
String s1,s2,sNull;  
s1 = "";  
s2 = " ";  
sNull = null;  
p(s1.length() == 0);  
p(s2.length() == 0);  
p(s1.isEmpty());  
p(s2.isEmpty());  
p(isEmptySafe(sNull));  
p(sNull.length() == 0);  
p(sNull.isEmpty());
```



```
true  
false  
true  
false
```

```
Exception in thread "main"  
java.lang.NullPointerException  
at  
zeichenketten.EmptyString.main(EmptyString.  
java:15)
```



# Sichere Methode für den Test auf einen Leerstring

- Eine sichere Methode einen *String* auf den Leerstring zu prüfen ist die Methode *isEmptySafe(String s)*.
- Diese nutzt *short circuit* Evaluierung und prüft zuerst, ob der *String s != null* ist.

```
private static boolean isEmptySafe(String s){  
    return s!=null && s.isEmpty();  
}
```



false



# Strings vergleichen

- **String** ist ein Referenztyp, d.h.:
  - **==** überprüft Identität
  - **equals** überprüft Gleichheit
- Strings sollten immer mit **equals** auf Gleichheit geprüft werden.
- Strings lassen sich mit **equalsIgnoreCase** auch Modulo Groß- und Kleinschreibung vergleichen.
- Lexikografischer Vergleich von Strings vergleicht zwei Strings zeichenweise und legt eine Ordnung auf Strings fest ( → Methoden **compareTo** und **compareToIgnoreCase**)
- Die Methode **s1.compareTo(s2)** liefert einen Wert
  - $<0$ : wenn  $s1 < s2$
  - $=0$  : wenn  $s1.equals(s2)$
  - $>0$ : wenn  $s1 > s2$
- Analog für **compareToIgnoreCase**, nur dass hier korrespondierende Klein und Großbuchstaben gleich sind.



# Strings vergleichen

```
String s1, s2;  
s1 = "Moin Folks";  
s2 = "m0in f0Lks";  
p(s1.equals(s2));  
p(s1.equalsIgnoreCase(s2));
```



false  
true

```
p("Moin".compareTo("Morgen") < 0);  
p("Morgen".compareTo("Morgen") == 0);  
p("Morgen".compareTo("Gestern") > 0);
```



true  
true  
true

→ Package stringmethoden Klassen StringEquals / StringCompare





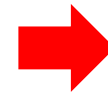
# Teile in einem String suchen

- String enthalten in? *contains*
- Position eines Teilstrings, eines Zeichen erstes oder letztes Vorkommen *indexOf*, *lastIndexOf*
- Position eines Teilstrings, eines Zeichen ab einer Position *str.indexOf*, *str.lastIndexOf*
- String enthalten am Anfang, am Ende? *startsWith*, *endsWith*
- Gleicher Teilstrings in zwei Strings enthalten? *regionMatches*



# Teile in einem String suchen

```
String str, s;  
char c = 'v';  
str = "Nachdem wir das Ziel aus unseren Augen verloren hatten, "  
      + "verdoppelten wir unsere Anstrengungen.";   
s = "ver";  
p(str.contains(s));  
p(str.indexOf(c));  
p(str.indexOf((int) c + 2)); // char = x  
p(str.indexOf(s, str.indexOf(s) + 1));  
p(str.lastIndexOf(s));  
p(str.lastIndexOf(s, str.lastIndexOf(s) - 1));  
  
p("http://informatik.haw-hamburg.de".startsWith("http"));  
p("http://informatik.haw-hamburg.de/scripts/v4.pdf".endsWith("pdf"));
```



true  
39  
-1  
56  
56  
39  
  
true  
true

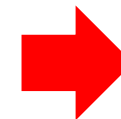
→ Package stringmethoden Klasse StringSearch



# Teilstring in zwei Strings enthalten?

- Verglichen werden die Teilstring in *str* von Position **33 bis 37** mit dem Teilstring in *s* von Position **20 bis 24**.
- Dies gibt in beiden Fällen den Teilstring "**Auge**".

```
String str, s,s2;  
str = "Nachdem wir das Ziel aus unseren Augen verloren hatten, "  
      + "verdoppelten wir unsere Anstrengungen.";   
s = "Er ging mit offenen Auges durch die Welt.";   
s2 = "Parlament: Offene Aussprache";   
p(str.regionMatches(33,s, 20, 4));   
p(str.substring(33,37));   
p(s.substring(20, 24));
```



true  
Auge  
Auge


→ Package stringmethoden Klasse StringRegionMatches



# Teilstring in zwei Strings enthalten?

- Verglichen wird der Teilstring **"offen"** in **s** mit **"Offen"** in **s2**.
- Durch das **true** im zweiten Match wird Groß-Kleinschreibung ignoriert. Das Ergebnis ist **true**.

```
String str, s,s2;  
str = "Nachdem wir das Ziel aus unseren Augen verloren hatten, "  
      + "verdoppelten wir unsere Anstrengungen.";   
s = "Er ging mit offenen Auges durch die Welt.";   
s2 = "Parlament: Offene Aussprache";  
p(s.indexOf("off"));  
p(s2.indexOf("Off"));  
p(s.regionMatches(12,s2, 11, 5));  
p(s.regionMatches(true,12,s2, 11, 5));  
p(s.substring(12, 17));  
p(s2.substring(11,16));
```



12
11
false
true
offen
Offen



# Teile extrahieren

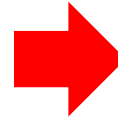
- Zeichen extrahieren: *charAt*
- Zeichenketten extrahieren: *substring*
- Zeichenfolgen in ein *char*-Array extrahieren: *getChars*
- Zeichen eines *Strings* in ein *char* Array umwandeln: *toCharArray*.
- Für alle diese Methoden gilt: Indizierte Zugriffe außerhalb des gültigen Bereichs erzeugen immer einen Laufzeitfehler  
*StringIndexOutOfBoundsException*
- Nicht gültige Bereiche sind:
  - Index < 0
  - Index >= *String.length()*
  - Ende Index < Start Index



# Zeichen und Teilstrings extrahieren

```
String str;  
str = "Nachdem wir das Ziel aus unseren Augen verloren hatten, "  
      + "verdoppelten wir unsere Anstrengungen."  
char[] cAry = new char[10];  
// Teilstrings extrahieren  
p(str.substring(33,43));  
p(str.subSequence(33, 43));  
p(str.substring(str.length() - 14));  
// char Arrays extrahieren  
str.getChars(33,38, cAry, 0);  
str.getChars(33, 38, cAry, 5);  
for (char c : cAry) {  
    println(c);  
}  
print("");  
// in char Array verwandeln  
cAry = str.toCharArray();  
for (char c : cAry) {  
    println(c);  
}
```

Augen verl  
Augen verl  
Anstrengungen.



AugenAugen

Nachdem wir das Ziel aus  
unseren Augen verloren  
hatten, verdoppelten wir  
unsere Anstrengungen.

→ Package stringmethoden Klasse StringExtract



# Teile ersetzen

- einzelne Zeichen oder Zeichenfolgen ersetzen: *replace*
  - alle Vorkommen einer Teilzeichenkette ersetzen (über reguläre Ausdrücke): *replaceAll*
  - erstes Vorkommen einer Teilzeichenkette ersetzen (über reguläre Ausdrücke): *replaceFirst*
- Finden die Methoden, die zu ersetzenden Zeichen oder Zeichenfolgen nicht in dem String, dann wird immer die Originalzeichenkette zurückgegeben.
- Da String immutable ist, arbeiten alle diese Methoden nicht destruktiv!



# Teile ersetzen

```
String str = "Auf    den Schultern    von Giganten.";
```

```
p(str.replace('n','x'));  
p(str.replace("Auf","Neben"));
```

```
// Mehr als zwei Leerzeichen durch eins ersetzen
```

```
p(str.replaceFirst(" +", " "));  
p(str.replaceAll(" +", " "));
```

```
// Punkt durch ! ersetzen  
// Liefert nicht das korrekte Ergebnis  
p(str.replaceFirst(".", "!"));  
p(str.replaceAll(".", "!"));
```

```
// Escapen von Sonderzeichen in regex  
// liefert das gewünschte Ergebnis
```

```
String regex1 = "\\.";  
String regex2 = Pattern.quote(".");  
p(str.replaceFirst(regex1, "!"));  
p(str.replaceAll(regex2, "!"));
```



```
Auf    dex Schulterx    vox  
Gigaxtex.  
Neben    den Schultern    von  
Giganten.
```

```
Auf den Schultern    von Giganten.  
Auf den Schultern von Giganten.
```

```
!uf    den Schultern    von Giganten.  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
Auf    den Schultern    von Giganten!  
Auf    den Schultern    von Giganten!
```

→ Package stringmethoden Klasse StringReplace





# Strings konkatenieren

- Für die *String*-Konkatenation steht entweder der Operator *+* oder die Methode *concat* zur Verfügung.
- *concat* ist schneller als *+*, da die Argumente *Strings* sein müssen und die Konvertierung der Argumente entfällt.
- *+* ist kürzer in der Schreibweise und bequemer in der Verwendung, da man sich um die Konvertierung der Operanden nicht kümmern muss.

```
p("x1".concat("+x2=").concat("x3"));  
p("x1" + "+x2=" + "x3");  
// p("x1=".concat(4)); // Fehler  
p("x1=" + 4);
```

→ Package stringmethoden Klasse StringConcat



# Leerzeichen entfernen

- Die Methode *trim* entfernt Leerzeichen (Whitespaces) am Anfang und Ende eines *String*.
- Zu den Leerzeichen zählen in Java: Leerzeichen, Tabulatoren, Zeilenumbrüche

```
String str = " \tAuf    den Schultern  von  Giganten. \n\n";  
p(str);  
p(str.trim());
```



Auf den Schultern von Giganten.

Auf den Schultern von Giganten.



# Besonderheiten der Klasse String

- **String** ist eine Bibliotheksklasse, die wir nicht selber definieren können, da **String** in einigen Aspekten besonders behandelt wird:
  - String-Literale erzeugen neue Objekte. Dies ist eine der Ausnahmen, in denen Objekte ohne explizites **new** erzeugt werden.
  - Überladene Operatoren (**+** für Strings), können wir in Java nicht selbst definieren.
  - **+** ist eine der wenigen Ausnahmen für Operatoren für einen Referenztyp.
  - Bei der Konkatenation von Objekten anderen Typs als String, wandelt der Java Compiler diese Objekte zunächst in **String** und führt dann die Konkatenation durch.
  - Bei primitiven Typen erfolgt das durch die Methode **String.valueOf**.



# Methode *toString*

- Bei der Verkettung von Objekten mit Zeichenketten ruft der Compiler die Methode *toString* auf den Objekten auf.
- Das Ergebnis von *toString* eines Objektes wird dann mit dem String konkateniert.
- Die Standardimplementierung von *toString* der Klasse *Object* liefert eine technische Sicht auf die Objekte und sollte daher immer überschrieben werden.

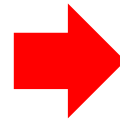


# Methode *toString*

```
public Rational(int zaehler, int nenner) {  
    this.zaehler=zaehler;  
    this.nenner=nenner;  
}
```

```
@Override  
public String toString() {  
    return zaehler + "/" + nenner;  
}
```

```
Rational r = new Rational(3, 4);  
p("Rational r is " + r);  
p("Rational r is " + r.toString());
```



```
Rational r is 3/4  
Rational r is 3/4
```

Ergebnis ohne toString  
in Rational



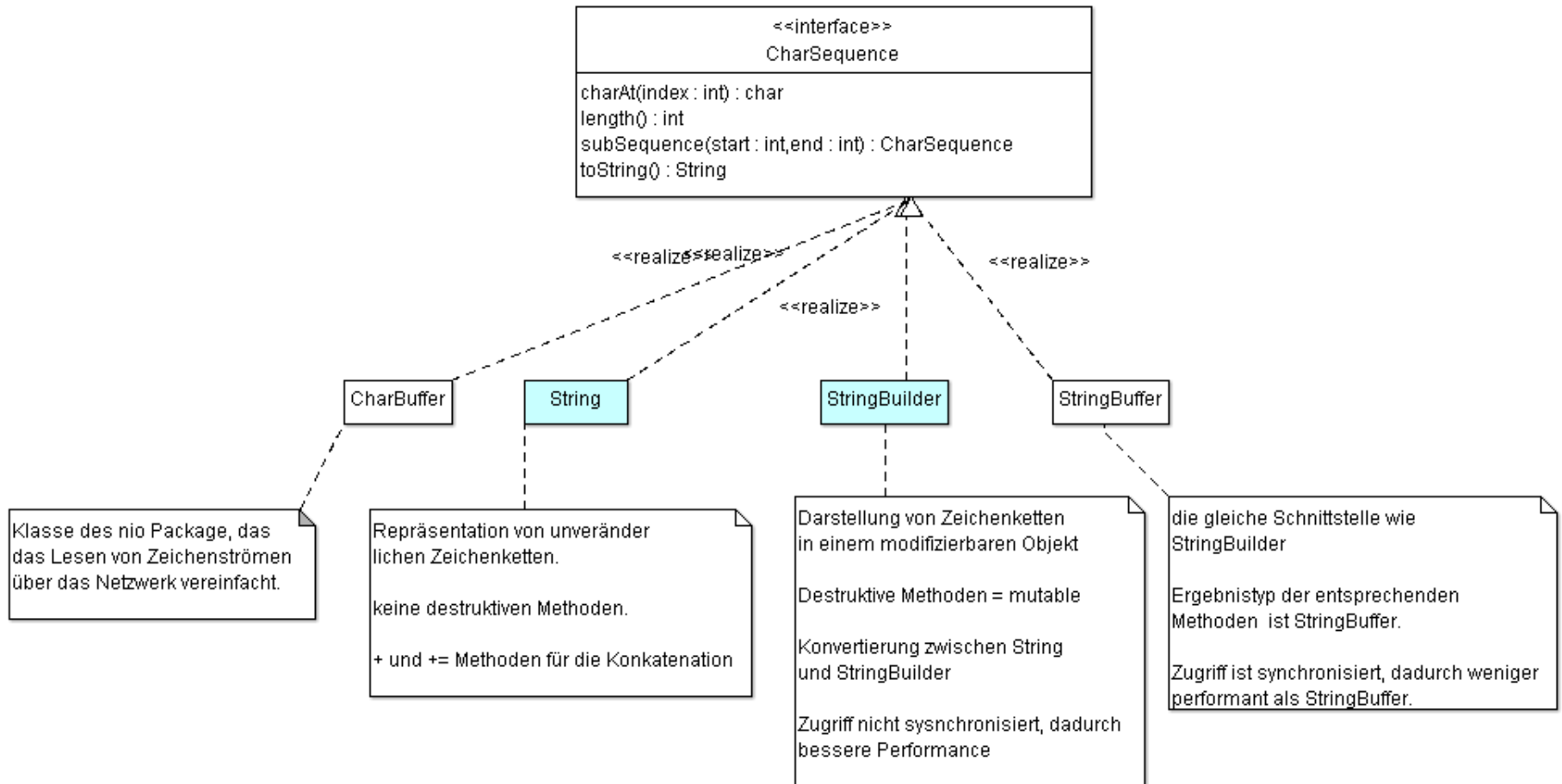
```
Rational r is toString.Rational@2a139a55  
Rational r is toString.Rational@2a139a55
```

→ Package *toString*



# Klassen und Interfaces für Zeichenketten

## Klasse *StringBuilder*





# Unterschiede *String* und *StringBuilder*

## *String*

- unveränderliche Repräsentationen von Zeichenfolgen (Immutable)
- ➔ Literale
- ➔ Jede Operation auf einem *String* erzeugt ein neues *String*-Objekt
- ➔ viele Manipulationen von Zeichenketten führen sehr schnell zu Performance-Problemen.

## *StringBuilder*

- veränderliche Repräsentationen von Zeichenfolgen (=Mutable)
- ➔ keine Literale
- ➔ destruktive Operationen verändern die Zeichenfolge
- ➔ effiziente Konkatenation durch Array-Operationen
- ➔ weniger Speicherbedarf und schnellere Operationen.



# Performance-Messung

## Konkatenation von Zeichenfolgen

- Geg. ein einfaches Programm, das für eine obere Grenze (*upperBound*) einen *String* wiederholt an einen *String* oder *StringBuilder* anhängt.
- Wir testen für verschiedene obere Grenzen die Zeit, die für die Konkatenation in der Klasse *String* im Vergleich zur Klasse *StringBuilder* benötigt wird.
- Dazu merken wir uns die Systemzeit vor Beginn der Konkatenation (*System.currentTimeMillis()*, *nanoTime()*) und bilden die Differenz mit der Zeit nach der Konkatenation.





# Konkatenation mit *String* +=

```
public class StringPerformance {  
    public static void main(String[] args) {  
        String harmless = "harmless";  
        System.out.println("started");  
        int upperBound = Integer.parseInt(args[0]);  
        stringConcat(harmless, upperBound);  
    }  
    private static void stringConcat(String harmless, int upperBound) {  
        long time = -System.currentTimeMillis();  
        String becomesHarmful = "";  
        for (int i = 0; i < upperBound; i++) {  
            becomesHarmful += harmless;  
        }  
        time += System.currentTimeMillis();  
        System.out.println("String Iterationen " + upperBound + " " + time + " ms");  
    }  
}
```

→ Package performance



# Konkatenation mit *StringBuilder append*

```
public class StringBuilderPerformance {  
    public static void main(String[] args) {  
        String harmless = "harmless";  
        int upperBound = Integer.parseInt(args[0]);  
        stringBuildConcat(harmless, upperBound);  
    }  
  
    private static void stringBuildConcat(String harmless, int upperBound) {  
        long time = -System.nanoTime();  
        StringBuilder sb = new StringBuilder();  
        for (int i = 0; i < upperBound; i++) {  
            sb.append(harmless);  
        }  
        time += System.nanoTime();  
        System.out.println("Builder Iterationen " + upperBound + " " + time + " ns");  
    }  
}
```

→ Package performance



# Messungen vom 19.10.2015

Anzahl Iterationen	String	StringBuilder
1.000	11 ms	0.28 ms
10.000	617 ms	3 ms
20.000	2148 ms	4 ms
50.000	12744 ms	8 ms
100.000	44963 ms	13 ms

Die Zeit für die Konkatenation von **String** Objekten wächst nichtlinear zur Anzahl der Konkatenations-Schritte.

**WARUM?**

Die Zeit für die Konkatenation mit **StringBuilder** Objekten wächst mit einer nahezu konstanten und sehr kleinen Größe.



# Performance der String-Konkatenation

## *String* Konkatenation

1. bei jeder **+=** Operation wird ein neues *String* Objekt erzeugt und der ursprünglichen Variablen zugewiesen. → Objekterzeugung ist teuer.
2. Das Objekt, das vor der Konkatenation in der Variable stand, wird nicht mehr referenziert, belegt aber nach wie vor Speicher. → Es entsteht Speichermüll (garbage).
3. Erreicht der Speichermüll eine kritische Grenze, dann räumt eine Laufzeitkomponente, der **Garbage Collector**, den Speichermüll auf: Nicht mehr referenzierte Objekte werden gelöscht und der Speicher wieder freigegeben.
4. Garbage Collection ist eine sehr teure Operation. Daher das inperformante Verhalten von **+=**.

# Performance der Konkatenation mit `StringBuilder`



## *StringBuilder* Konkatenation mit *append*

1. *append* hängt einen *String* an ein bestehendes *StringBuilder* Objekt an.
2. Es existiert also während der gesamten Iteration nur genau ein *StringBuilder* Objekt.
3. Es entsteht **kein** Speichermüll.
4. Der Garbage Collector wird nicht aktiv. Daher performantes Konkatenieren mit *StringBuilder*.



# *StringBuilder*-Methoden

*StringBuilder*

erzeugen

Elemente  
anhängen

Zeichenfolgen  
abfragen

Vergleich mit  
Strings

Länge  
bestimmen

Elemente  
ersetzen

Zeichenfolgen  
einfügen

Zeichenfolgen  
löschen

Zeichenfolgen  
Invertieren



# *StringBuilder* erzeugen

- ***StringBuilder()***:  
legt ein neues Objekt an, das Leerzeichen enthält und Platz für 16 Zeichen bietet.
- ***StringBuilder(int cap)***:  
legt ein neues Objekt an, das Leerzeichen mit Platz für *cap* Zeichen enthält.
- ***StringBuilder(String str)***: Kopier-Konstruktor  
legt ein neues Objekt an, das eine Kopie der Zeichen von *str* enthält.  
Konvertierung von *String* in *StringBuilder*.
- ***StringBuilder(CharSequence seq)***: Kopierkonstruktor  
legt ein neues Objekt an, das eine Kopie der Zeichen von *seq* enthält.  
Konvertierung von *CharSequence* Objekten in *StringBuilder*.



# *StringBuilder* erzeugen

```
StringBuilder sb;  
p("StringBuilder Objekte erzeugen");  
p(sb =new StringBuilder());  
p(sb.capacity());p(sb.length());  
p(sb=new StringBuilder(50));  
p(sb.capacity());p(sb.length());  
p(sb =new StringBuilder("www.haw-hamburg.de"));  
p(sb.capacity());p(sb.length());  
p(sb =new StringBuilder(sb));  
p(sb.capacity());p(sb.length());
```



StringBuilder Objekte erzeugen

```
16  
0  
50  
0  
www.haw-hamburg.de  
34  
18  
www.haw-hamburg.de  
34  
18
```





# Länge eines *StringBuilders*

- *StringBuilders* kennt zwei Methoden, um die Länge zu bestimmen:
  - *length*: die Anzahl der enthaltenen Elemente
  - *capacity*: der reservierte Platz für einzufügende Element (initial 16 Zeichen). Wird nach jedem Einfügen angepasst.
- *setLength(int l)* setzt die Länge auf eine angegebene Anzahl von Zeichen.
  - *l < length()*: wird der Rest der Zeichenkette abgeschnitten.
  - *l > length()*: vergrößert den Puffer, füllt die übrigen Zeichen mit Nullzeichen '\0000' auf.

```
// Länge und Kapazität
sb = new StringBuilder( "www.haw-hamburg.de" );
int length    = sb.length();           // 18
int capacity  = sb.capacity();         // 34
p(length);
p(capacity);
```



# Elemente anhängen

- Mit den **append** Methoden werden Elemente an den **StringBuilder** angehängt.
- sind mehrfach für unterschiedlichen Parametertypen überladen.
- Methode **StringBuilder append(CharSequence s, int start, int end)** erlaubt beliebige **CharSequence** Objekte anzufügen, was die Verknüpfung von **String** und **StringBuilder** vereinfacht.
- Jede **append** Methode verändert den Inhalt eines **StringBuilders** und liefert eine Referenz auf das Objekt zurück → **append** Kaskaden sind möglich.

```
StringBuilder append( boolean b )
StringBuilder append( char c )
StringBuilder append( char[] str )
StringBuilder append( char[] str, int
    offset, int len )
StringBuilder append( CharSequence s )
StringBuilder append( CharSequence s,
    int start, int end )
StringBuilder append( double d )
StringBuilder append( float f )
StringBuilder append( int i )
StringBuilder append( long lng )
StringBuilder append( Object obj )
StringBuilder append( String str )
StringBuilder append( StringBuffer sb )
```



# Elemente anhängen

```
p("Elemente anhängen");  
p(sb.append(false));  
p(sb.append('Z'));  
p(sb.append(new char[] {'v','e','r','z','o','c','k','t'}));  
p(sb.append(sb.substring(0, 3)));  
p(sb.append(6710.987)); // analog float, long, int  
p(sb.append(new Person("Crash", "Test Dummy")));  
p(sb.append(new char[] {'v','e','r','z','o','c','k','t'}, 3, 4));
```



Elemente anhängen

www.haw-hamburg.defalse

www.haw-hamburg.defalseZ

www.haw-hamburg.defalseZverzockt

www.haw-hamburg.defalseZverzocktwww

www.haw-hamburg.defalseZverzocktwww6710.987

www.haw-

hamburg.defalseZverzocktwww6710.987P(Crash,Test  
Dummy)

www.haw-

hamburg.defalseZverzocktwww6710.987P(Crash,Test  
Dummy) zock

# Zeichen(folgen) abfragen, (er)setzen und einfügen, löschen und invertieren



- Abfrage Methoden:
  - *getChars, indexOf, lastIndexOf, substring* von *String* finden sich auch in *StringBuilder* wieder.
  - *charAt* und *subSequence*  
Implementierungen der  
Interfacemethoden.
- Modifizierende Operationen:
  - *setCharAt(int indx, char c)*
  - *insert(int offs, Type val) : Type ist hier* eines der Typen, die auch für *append* definiert sind.
  - *insert(int index, char[] cAry, int offs, int len) :* es wird nur ein Teil des *cAry* (*offs-offs+len*) übernommen
  - *replace(int strt, int end, String str)*  
ersetzt den Bereich *[strt, end)* durch *str*
  - *delete(int start, int end):* löscht Elemente im Intervall *[start, end]*
  - *deleteCharAt(int indx):* löscht das Zeichen an Position *indx*.
  - *reverse():* invertiert eine Zeichenkette destruktiv



# Zeichen(folgen) abfragen

```
sb= new StringBuilder( "www.haw-hamburg.de" );
p("Zeichenketten und Zeichen abfragen");
//-----
String subStr = "ha";
p(sb.charAt(0));
//-----
char[] cAry = new char[10];
sb.getChars(4,6,cAry,2);
pAry(cAry);           // Ausgabe siehe Klasse Printer
p("");
// sb.getChars(0,sb.length(),cAry,2); // ArrayIndexOutOfBoundsException
// cAry hat nicht genügend Speicher
//-----
// p(sb.charAt(sb.length())); // StringIndexOutOfBoundsException
p(sb.indexOf(subStr));
p(sb.indexOf(subStr,sb.indexOf(subStr)+1));
p(sb.lastIndexOf(subStr));
p(sb.lastIndexOf(subStr,sb.lastIndexOf(subStr)-1));
//-----
p(sb.subSequence(0, 5)); // www.h
p(sb.substring(5));      // aw-hamburg.de
p(sb.substring(0, 5));   // www.h
```



```
Zeichenketten und Zeichen abfragen
w
[,,h,a,,,,]
4
8
8
4
www.h
aw-hamburg.de
www.h
```



# Zeichen(folgen) (er)setzen und einfügen

```
p("Zeichenfolgen (er)setzen und einfügen");
sb.setCharAt(7, 'Y');
p(sb);
p(sb.insert(7, true));
p(sb.insert(7, 'X'));
p(sb.insert(7, new char[] {'c', 'o', 'o', 'l'}));
p(sb.insert(7, new StringBuffer("not so ")));
p(sb.insert(7, 10.7)); // analog für float int long
p(sb.insert(7, new char[] {'c', 'o', 'o', 'l'}, 1, 3));
p(sb.insert(7, new Person("Crash", "Test Dummy")));
p(sb.replace(7, 49, ""));
p(sb.replace(7, 8, "-"));
```



Zeichenfolgen (er)setzen und einfügen  
www.hawYhamburg.de  
www.hawtrueYhamburg.de  
www.hawXtrueYhamburg.de  
www.hawcoolXtrueYhamburg.de  
www.hawnot so coolXtrueYhamburg.de  
www.haw10.7not so coolXtrueYhamburg.de  
www.hawool10.7not so coolXtrueYhamburg.de  
www.hawP(Crash,Test Dummy)ool10.7not so  
coolXtrueYhamburg.de  
www.hawYhamburg.de  
www.haw-hamburg.de



# Zeichen(folgen) löschen und invertieren

```
p("Zeichen(folgen) löschen und invertieren");  
p(sb.deleteCharAt(0));  
p(sb.delete(0, 2));      // Ende Index exklusiv  
p(sb.reverse());  
p(sb);
```



Zeichen(folgen) löschen und invertieren  
ww.haw-hamburg.de  
.haw-hamburg.de  
ed.grubmah-wah.  
ed.grubmah-wah.



# Vergleich von *String* und *StringBuilder*

- *equals* in *String* prüft zunächst auf Klassengleichheit. Daher würde ein Vergleich zwischen *StringBuilder* und *String* immer *false* ergeben, auch wenn beide den gleichen Inhalt haben.
- Zum Vergleich mit einem *String* müssen daher *StringBuilder* Objekte zunächst in einen *String* gewandelt und dann mit *equals* von *String* verglichen werden.
- Alternativ lässt sich mit der Methode *contentEquals* von *String* ein direkter Vergleich durchführen. Die Methode prüft auf gleichen Inhalt und ignoriert die Kapazität des *Builders* im Vergleich.
- Da es keine *equals* Methode in *StringBuilder* gibt, können zwei *StringBuilder* nur verglichen werden, wenn Sie zuvor in einen *String* gewandelt werden.
- Ohne überschriebene *equals* Methode gibt es konsequenterweise auch keine *hashCode* Methode für *StringBuilder*.





# Vergleich von *String* und *StringBuilder*

```
p("Vergleich StingBuilder String");
String s = "Entspannter Morgen";
StringBuilder sb1 = new StringBuilder( "Entspannter Morgen" );
p( s.equals(sb) );           // false
p( s.equals(sb1.toString()) ); // true
p( s.contentEquals(sb1) );   // true

sb1 = new StringBuilder( "www.haw-hamburg.de" );
StringBuilder sb2 = new StringBuilder( "www.haw-hamburg.de" );
p( sb1.equals( sb2 ) );      // false
p( sb1.toString().equals( sb2.toString() ) ); // true
p( sb1.toString().contentEquals( sb2 ) );   // true
```



false
true
true
false
true
true



# String Formatierung

- Der Erzeugen größerer Texte über Konkatenieren von Teilstrings kann sehr schreibintensiv werden.
- Die Methode *String.format* ermöglicht es einen Text durch Ersetzen von Platzhaltern in einer Schablone flexibel zu konstruieren.
- Die Methode erwartet mindestens ein Argument, den **Formatstring**.
- Unveränderliche Bestandteile stehen wörtlich in dem Formatstring, variable Bestandteile werden durch Ersetzen von Formatangaben eingefügt. Formatangaben beginnen mit dem %-Zeichen.
- Werte, die für die Formatangaben sind die weiteren Parameter der Methode *String.format*.
- Um die Anzahl der variablen Anteile im Formatstring nicht einzuschränken, verfügt die Methode *String.format* über eine variable Parameterliste.

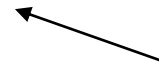


# String Formatierung

Formatstring: %f sind Platzhalter für zu ersetzende Werte.  
f besagt, dass der Wert eine Gleitkommazahl sein muss.



```
String formatString = "Der Abstand des Punktes (%f,%f) vom Ursprung ist %f";  
double x = 1;  
double y = 2.5;  
String s = String.format(formatString, x,y,Math.hypot(x, y));  
System.out.println(s);
```



x,y, Math.hypot(x,y) sind die variablen Argumente.  
Die Werte werden in der Reihenfolge der Nennung eingesetzt



Der Abstand des Punktes (1,000000,2,500000) vom Ursprung ist 2,692582

→ Package format



# String Formatierung

- Es gibt eine Reihe von Formatangaben für verschiedene Typen von Java.
  - %d    Ganze Zahl
  - %f    Gleitkomma Zahl
  - %s    Zeichenkette
  - %c    Zeichen
  - ...    ...
- Für jede Formatangabe muss hinter dem Formatstring ein Wert passenden Typs folgen.
- Daneben gibt es Formatangaben, für die kein Argumente angegeben wird.
  - %n    Zeilenwechsel
  - %%    % Zeichen



# String Formatierung

- Zwischen den %-Zeichen und den Typ-Buchstaben können zusätzliche Steuerzeichen stehen, die die Art der Darstellung beeinflussen.
  - m      Wert mit Leerzeichen auf mindestens m Spalten auffüllen
  - .<p>    Gleitkommazahl mit mindestens p Stellen nach dem Komma ausgeben
  - 0      Zahlenwert mit führenden Nullen statt Leerzeichen auffüllen
  - Linksbündig anordnen
  - +      Zahlenwert mit Vorzeichen ausgeben
- Eine Kurzform, um einen formatierten String auf der Konsole auszugeben:  
***System.out.printf(formatString,...)***
- anstelle von  
***System.out.print(String.format(formatString,...))***



# String Formatierung

```
System.out.println(String.format("|%4d|",23));  
System.out.printf(String.format("|%-4d|",23));  
System.out.printf(String.format("|%12f|",Math.sqrt(2)));  
System.out.printf(String.format("|%12.2f|",Math.sqrt(2)));  
System.out.printf(String.format("|%12f|",1e20));
```



```
|  23|  
|23  |  
|      1,414214|  
|              1,41|  
|10000000000000000000,000000|
```

→ Package format



# Zusammenfassung Zeichenketten

- Zeichenketten sind Abfolgen von Zeichen
- Zeichenketten lassen sich durch Literale einfach erzeugen
- Zeichenketten sind immutable und haben daher Wertsemantik
- Die Klasse String überlädt den Operator **+**.
- **+** ruft auf allen Objekte in der Konkatenation mit einem String die Methode `toString()` auf
- **StringBuilder** ist eine performante Alternative für das Konkatenieren von Zeichenfolgen.
- **StringBuilder** ist mutable und verfügt über eine Reihe von destruktiven Methoden.
- Die Methode **format** interpretiert Formatanweisungen im Zeichenketten und erlaubt das Formatieren unterschiedlicher Datentypen in einer Zeichenkette.



# Aufgabe

1. Eine Vampirzahl hat eine gerade Anzahl von Ziffern und wird durch Multiplikation zweier Zahlen, die beide die Hälfte der Länge der Ziffern der Vampirzahl haben und nur aus den Ziffern der Vampirzahl bestehen, erzeugt. Schreiben Sie ein Programm, das alle 4-ziffrigen Vampirzahlen berechnet.

Beispiele:

$$1260 = 21 * 60$$

$$1827 = 21 * 87$$

$$2187 = 27 * 81$$





# Quelle

