

# Programmieren I

## Strings und Wrapperklassen



Heusch 7.3  
Ratz 6.5.2

Institut für Automation und angewandte Informatik

```
final List<String> allResults = new ArrayList<String>();  
final Map<String, Integer> typeWordResultCount = new HashMap<String, Integer>();  
final Map<String, Integer> typePoints = new HashMap<String, Integer>();  
evaluation.put(type, typePoints);  
  
for (final Sheet sheet : this.sheets) {  
    final String sheetResult = sheet.getPlayerInput(type);  
    if (sheetResult.startsWith(start) && this.isValidWord(sheetResult, type)) {  
        validWordCountForType++;  
        allResults.add(sheetResult);  
    }  
}
```

# Die Klasse `String`

- Für Zeichenketten gibt es in Java die Klasse `String`
  - `String` ist kein primitiver Datentyp sondern eine Klasse.
  - Die Klasse `String` besitzt Methoden zum Arbeiten mit Zeichenketten
  - Sonderform, Objekte müssen nicht mit `new` erzeugt werden.
  - Die Klasse `String` ist `final`, keine weiteren Klassen ableitbar.
  - Keine Setter-Methoden zum Ändern der gespeicherten Zeichenkette. Daher werden `String`-Objekte auch *immutable* (unveränderbar) bezeichnet.
  - Inhalt und Länge eines Strings sind immer konstant. Die Zuweisung einer neuen Zeichenkette erzeugt ein neues `String`-Objekt und gibt das alte frei.
  - Zeichenketten besteht aus Unicode-Zeichen.
  - Zeichenketten im Quellprogramm (Literele) werden in doppelte Anführungszeichen eingeschlossen.
  - Die leere Zeichenkette `" "` hat die Länge 0.

# String-Objekte

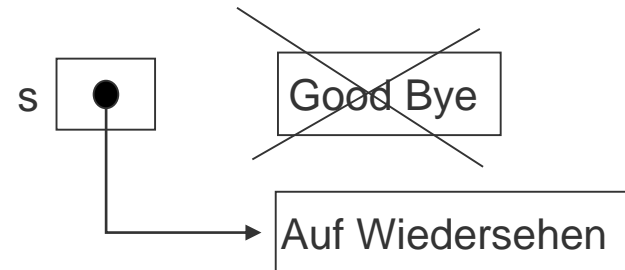
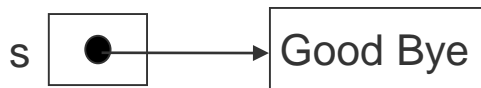
## ■ Deklaration von String-Variablen und Erzeugung von Strings:

```
String identifier;  
identifier = "Text";
```

```
String identifier = "Text";
```

## ■ Zuweisung einer neuen Zeichenkette:

```
String s = "Good bye";  
// ...  
s = new String("Auf Wiedersehen");
```



# Zeilenumbrüche und Stringausgabe

- Mit der Escape-Sequenz "`\n`", die man als Text in einen `String` eingeben kann, wird bei der Ausgabe des Strings ein Zeilenumbruch erzeugt.
- Strings mit der Methode `System.out.printf` ausgeben:  
Neben der Methoden `System.out.print` und `System.out.println` können Strings auch mit der Methode `System.out.printf` ausgegeben werden:

```
String text = "rectangle";  
int i = 4;  
System.out.printf("A %s has %d corners.\n", text, i);  
                // auch %n möglich
```

 **Ausgabe**  
A rectangle has 4 corners.

# Eingabe von Strings

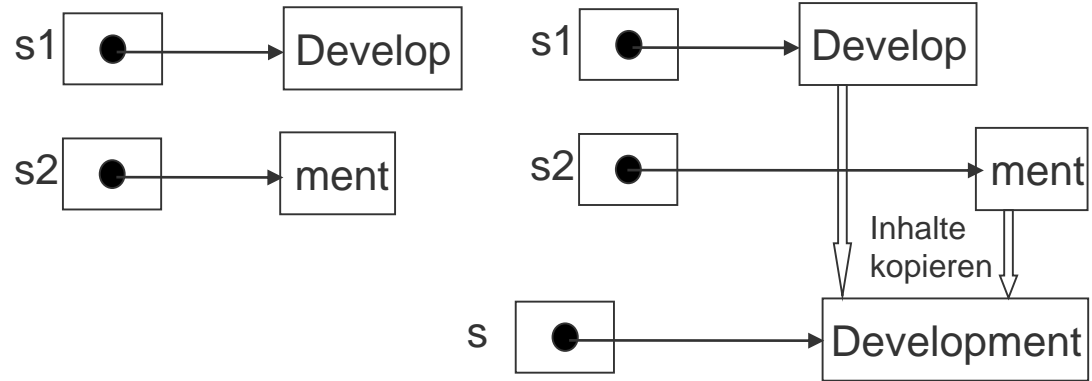
- Für die Eingabe von Strings mit der Klasse `java.util.Scanner` (vgl. Kapitel „Primitive Datentypen“) hat diese Klasse u.a. die Methoden
  - `String next()`  
Es werden die Zeichen bis zum nächsten Trennzeichen (Leerzeichen, Tabulator, Zeilentrenner) eingelesen (ein Token).
  - `String nextLine()`  
Es werden die Zeichen bis zum nächsten Zeilentrenner eingelesen.
- Beispiel für das Einlesen eines Token von der Standardeingabe:

```
java.util.Scanner scan = new java.util.Scanner(System.in);  
String s = scan.next(); // Einlesen eines String Tokens
```

# Strings verketten und vergleichen (1)

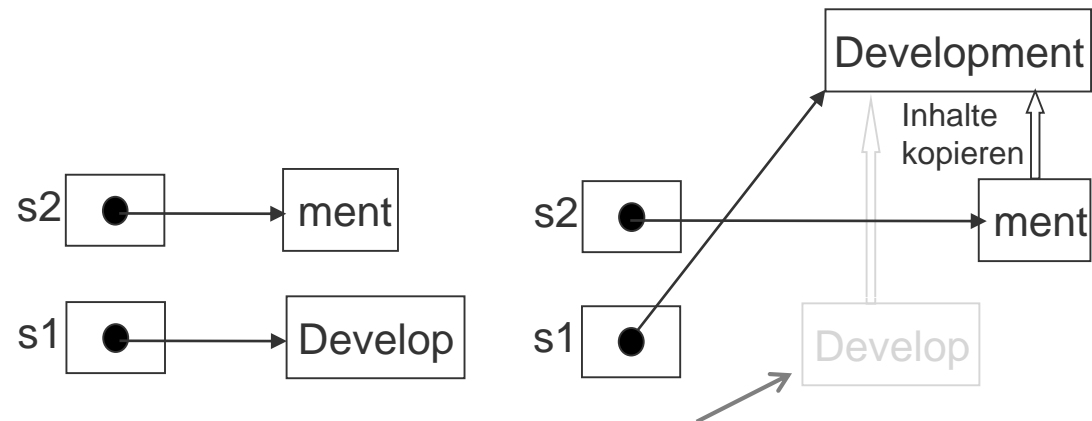
## ■ Strings verketten

```
String s1 = "Develop";
String s2 = "ment";
String s = s1 + s2;
```



```
String s1 = "Develop";
String s2 = "ment";

s1 = s1 + s2;
// oder: s1 += s2;
```



*Wird vom GarbageCollector eingesammelt sofern keine andere Referenz besteht*

# Strings verketten und vergleichen (2)

## ■ Strings vergleichen

### Interne Verwaltung von Strings:

- Strings deren Inhalt beim Compilieren bereits feststeht, z.B. `String s="Hallo"`, werden nur einmal angelegt (ein „String-Pool“ wird erzeugt). Existieren während des Kompilierens weitere Strings mit dem gleichen Inhalt, verweisen alle auf die gleiche interne Zeichenkette "Hallo", d.h. auf dasselbe `String`-Objekt.
- Für Strings, die dynamisch, d.h. erst zur Laufzeit eines Programms, gebildet werden, erzeugt Java immer ein neues separates `String`-Objekt.

# Strings inhaltlich vergleichen

- Für den inhaltlichen Vergleich von Strings gibt es in der Klasse `String` die Instanzmethoden:
  - `boolean equals(Object o)`  
vergleicht die Inhalte von `String`-Objekten, d.h. die Zeichenketten selbst, und liefert als Ergebnis entsprechend `true` oder `false`.
  - `boolean equalsIgnoreCase(String s)`  
überprüft den String auf inhaltliche Gleichheit mit einem anderen String, ignoriert aber die Groß-/Kleinschreibung.

```
String s1 = "text";  
String s2 = new String("Text");
```

```
System.out.println(s1.equals(s2));  
System.out.println(s1.equalsIgnoreCase(s2));
```

Ausgabe	
→	false
→	true



# Strings lexikografisch oder Teile vergleichen

## ■ `int compareTo (String s)`

Mit dieser Methode können Strings lexikografisch verglichen werden. Rückgabewerte sind 0 (bei Gleichheit) oder entsprechend der lexikografischen Ordnung kleiner oder größer 0.

## ■ `boolean contains (CharSequence s)`

Diese Methode überprüft ob eine Zeichenkette in einer anderen enthalten ist.

## ■ Beispiel:

```
String s1 = "text";  
String s2 = new String("Text");  
String s3 = new String("Context");
```

```
System.out.println(s1.compareTo(s2));  
System.out.println(s2.compareTo(s1));  
System.out.println(s3.contains(s1));  
System.out.println(s3.contains(s2));
```

Ausgabe

32  
-32  
true  
false

# Vorsicht beim Vergleich von Strings mit ==

```
String s1 = "Text";  
String s2 = "Text";  
String s3 = new String("Text");  
String s4 = new String("Text");
```

```
System.out.println(s1==s2);  
System.out.println(s2==s3);  
System.out.println(s3==s4);
```

Ausgabe

```
> true  
false  
false
```

- Beim Vergleich von Referenzen auf String-Objekte mit dem Operator == kann es unterschiedliche Ergebnisse geben:
  - Beim Vergleich von zwei identischen String-Literalen liefert == das Ergebnis `true`, da es sich um Referenzen auf dasselbe `String`-Objekt im String-Pool handelt
  - Werden Strings im Programm dynamisch mit identischem Inhalt erzeugt, liefert der Vergleich der Referenzvariablen den Wert `false`, da es sich um verschiedene `String`-Objekte handelt

# Methoden der Klasse String (1)

- `char charAt(int index)`  
Liefert das Zeichen an der Position index

```
char c = "abc".charAt(1);
```

- `boolean endsWith(String value)`  
Überprüft, ob der String mit value endet

```
String s1 = "Text";  
if ( s1.endsWith("xt") ) { /*...*/ }
```

- `int indexOf(String value)`  
Liefert den Index innerhalb eines Strings, an dem value beginnt. Falls nicht vorhanden, wird -1 geliefert.

```
String s = "Application";  
int i = s.indexOf("atio"); // liefert 6
```

## Methoden der Klasse String (2)

### ■ `int length()`

Liefert die Länge der Zeichenkette

```
int len = s.length(); // 11
```

```
String s = "Application";
```

### ■ `String replace(char oldC, char newC)`

Ersetzt alle vorkommenden Zeichen oldC durch newC

```
String s2 = s.replace('c', 'k');  
// nun sind s="Application" bzw. s2="Applikation"
```

### ■ `String substring(int s)`

`String substring(int s, int s1)`

Liefert einen Teil-String beginnend beim Index s bis zum Ende bzw. bis zu s1-1

```
String s3 = s.substring(2, 4);  
// nun ist s3="pl"
```

## Methoden der Klasse String (3)

- `static String valueOf(type var)`  
Liefert die String-Darstellung des Datentyps bzw. Objekts
- `boolean matches(String regex)`  
ermittelt, ob ein String dem regulären Ausdruck `regex` entspricht
- `String replaceAll(String regex, String replacement)`  
Ersetzt alle Vorkommen von `regex` durch `replacement`. `regex` ist ein *regulärer Ausdruck*

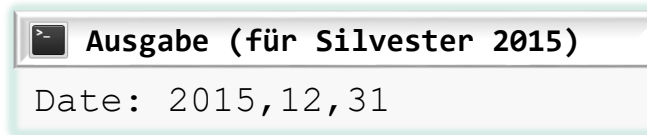
 Ab Java 1.4

 Ab Java 1.4

# Strings formatieren mit `format`

## ■ Beispiel: Datum formatieren und ausgeben

```
GregorianCalendar date = new GregorianCalendar();  
String s = String.format("Date: %04d,%02d,%02d",  
    date.get(Calendar.YEAR),  
    date.get(Calendar.MONTH) + 1, // month: 0 based  
    date.get(Calendar.DAY_OF_MONTH));  
System.out.println(s);
```



Ausgabe (für Silvester 2015)  
Date: 2015,12,31

## ■ Die Parameter für die Formatierung sind dabei dieselben wie bei `printf` (siehe Folien zu Operatoren).

# Strings formatieren mit Formatter-Klassen

- Datumsformatierungen noch einfacher mit der Klasse `java.text.SimpleDateFormat`

- Angabe eines Musters („Pattern“) für die Ausgabe des Datums

```
SimpleDateFormat sdf = new SimpleDateFormat("YYYY,MM,dd");
Date date = new Date();
System.out.println("Date: " + sdf.format(date));
```



```
> Ausgabe (für Silvester 2015)
Date: 2015,12,31
```

YYYY	Jahr, 4-stellig
MM	Monat, 2-stellig
dd	Tag des Monats, 2-stellig

Dokumentation der möglichen Pattern-Bestandteile:

<http://docs.oracle.com/javase/8/docs/api/index.html?java/text/SimpleDateFormat.html>

- Weitere Formatter-Klasse: `java.text.DecimalFormat`
  - Formatierte Ausgabe von Zahlen, bspw. maximale Anzahl von Nachkommastellen

# Die Klasse `StringBuffer`

## ■ Veränderliche Zeichenketten

- Die Klasse `String` bietet keine Möglichkeit, einen `String` nachträglich zu ändern. Ersetzt man z.B. Zeichen eines `String`s, wird immer ein neues `String`-Objekt erzeugt
- Die Klasse `StringBuffer` stellt einen Datentyp bereit, bei dem `String`s verändert werden können und sich die Größe dynamisch der gespeicherten Zeichenkette anpasst

## ■ Eigenschaften der Klasse `StringBuffer`:

- Verwalten und bearbeiten von veränderlichen Zeichenketten
- Schnelligkeit, Faktor 10-20 schneller als `String`s aufgrund der dynamischen Verwaltung der Zeichenketten
- Dynamische Anpassung des Puffers an die Größe des `String`s, (i.d.R. größer als die gespeicherte Zeichenkette)



# StringBuffer erzeugen und umwandeln

- **StringBuffer()** ;  
Erzeugt ein `StringBuffer`-Objekt mit der vorgegebenen Kapazität von 16 Zeichen und einer leeren Zeichenkette.
- **StringBuffer(int length)** ;  
Erzeugt einen `StringBuffer` mit der Initialkapazität von `length` Zeichen.
- **StringBuffer(String str)** ;  
Eine Kopie von `str` wird im `StringBuffer` erzeugt. Die Kapazität wird der Größe von `str` angepasst.

```
StringBuffer sb = new StringBuffer("Text");  
String s = sb.toString();  
s = new String(sb);
```

# Methoden der Klasse StringBuffer (1)

- `StringBuffer append(type value)`  
value wird in einen `String` gewandelt und an den Puffer angehängt.

```
StringBuffer s;  
s = new StringBuffer();  
s.append("1 + 2 = ");  
s.append(1 + 2);
```

- `int capacity()`  
Liefert die aktuelle Kapazität des Puffers.
- `char charAt(int index)`  
Liefert das Zeichen mit dem betreffenden Index.
- `void setCharAt(int index, char c)`  
Ändert das Zeichen mit dem betreffenden Index.

## Methoden der Klasse StringBuffer (2)

- `StringBuffer insert(int index, type value)`  
Fügt ab der Position `index` die String-Repräsentation des Parameters `value` ein

```
StringBuffer s = new StringBuffer();  
s.append("1 + 2 = 3");  
s.insert(0, "(");  
s.insert(6, ")");  
// im Puffer: "(1 + 2) = 3"
```

- `int length()`  
Liefert die Länge des gespeicherten Strings
- `String toString()`  
Liefert die String-Repäsentation des Puffers

# Die Klasse `StringBuilder`

- Im JDK 5 wurde die Klasse `StringBuilder` ergänzt
- Diese Klasse entspricht in der Funktionsweise der Klasse `StringBuffer`
- `StringBuilder` sind allerdings nicht synchronisiert. Das bedeutet, dass sie *nicht* für den Zugriff von mehreren Threads (gleichzeitig ausgeführten Programmmodulen) geeignet sind
- Da gleichzeitig ablaufende Programmmodule nicht überwacht werden, ergeben sich in vielen Anwendungsfällen durch den Einsatz der Klasse `StringBuilder` nochmals Reduktionen der Ausführungszeit (`StringBuilder` ist noch schneller als `StringBuffer`)

 **Ab Java 1.5**

# Wrapper-Klassen

- Viele Methoden innerhalb Java erwarten als Parameter einen `Object`-Typ. Da von primitiven Datentypen keine Objekte gebildet werden können, stellt Java für jeden primitiven Datentyp eine so genannte *Wrapper*-Klasse.
- Die Klassen besitzen ein gekapseltes Attribut, das den Wert speichert und Methoden um beispielsweise den entsprechenden primitiven Datentyp oder eine Konvertierung von `String`-Typen in den betreffenden Datentyp vorzunehmen.
- Wrapper-Klassen im package `java.lang`:

```
Void  
Byte      Short   Integer  Long  
Float     Double  
Boolean  
Character
```



**Heusch 7.4**  
**Ratz 12.2**

# Wrapper-Klassen verwenden

- Am Beispiel der Wrapper Klasse `Integer` werden im Folgenden einige Methoden und Konstruktoren der Klassen vorgestellt.
  - Für die anderen Wrapper-Klassen der anderen primitiven Datentypen existieren ähnliche Methoden.
  - Informationen hierzu findet man in der Java-Dokumentation ([java.oracle.com](http://java.oracle.com))
- Konstruktor der Wrapper-Klasse:
  - Mit `new` wird ein Objekt der betreffenden Wrapper-Klasse erstellt. Dazu stehen mehrere Konstruktoren zur Verfügung.

```
Integer(int value)
```

```
Integer(String s)
```

Der Wert des Integer-Objektes kann im Konstruktor als `int` oder als `String` angegeben werden. Der Konstruktor versucht den `String s` in einen `int`-Wert zu konvertieren. Ist dies nicht möglich tritt ein Laufzeitfehler ein (`NumberFormatException`)

# Methoden der Wrapper-Klasse Integer

Methode	Beschreibung
<code>int intValue()</code>	Diese Methode liefert den gespeicherten int-Wert zurück
<code>static int parseInt(String s)</code>	Versucht den String <code>s</code> in einen int-Wert zu konvertieren. Falls eine Konvertierung nicht möglich ist, wird eine <code>NumberFormatException</code> ausgelöst.
<code>String toString()</code>	Liefert einen String zurück, der den gespeicherten Wert repräsentiert.
<code>static String toString(int value)</code>	Liefert einen String zurück, der den Wert von <code>value</code> repräsentiert.
<code>static Integer valueOf(int value)</code>	Erzeugt mit dem Wert von <code>value</code> ein Integer-Objekt und liefert dieses zurück.
<code>static Integer valueOf(String s)</code>	Versucht den String <code>s</code> in einen int-Wert zu konvertieren, erzeugt mit diesem Wert ein Integer-Objekt und liefert dieses zurück. Falls eine Konvertierung nicht möglich ist, wird eine <code>NumberFormatException</code> ausgelöst.

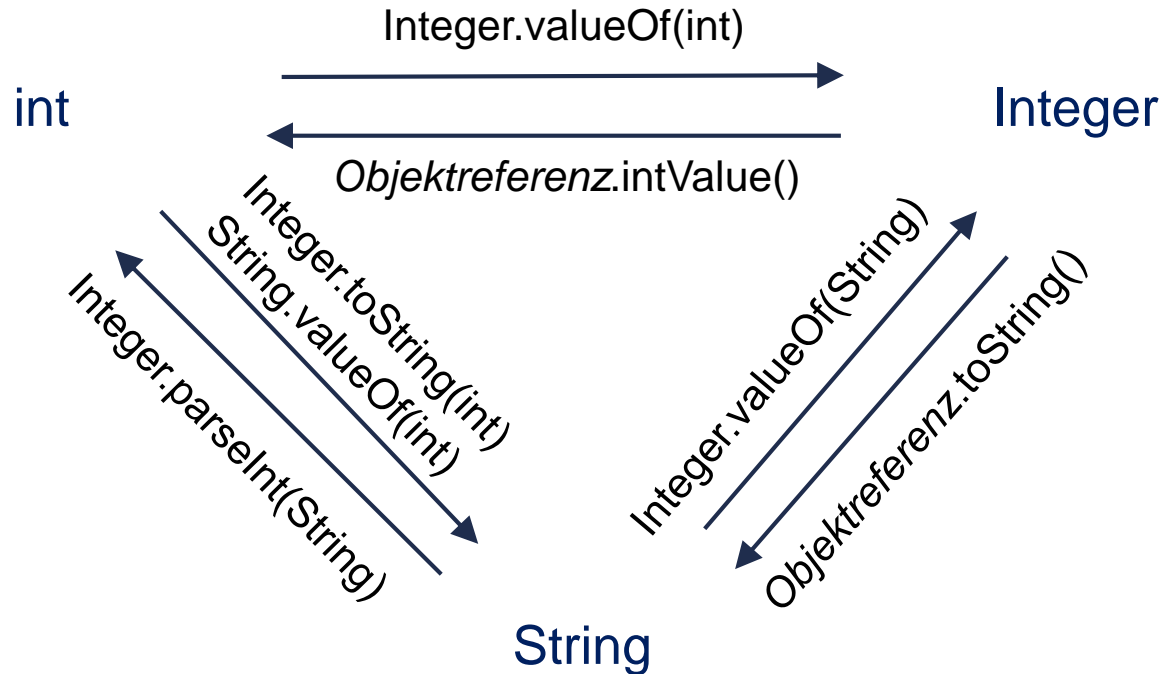
# Beispiele

```
Integer i1 = new Integer(10);           // neues Integer-Objekt mit Wert 10
Integer i2 = new Integer("12");          // neues Integer-Objekt mit Wert 12
Integer i3 = Integer.valueOf("24");      // neues Integer-Objekt mit Wert 24
int i = i1.intValue();                   // gespeicherten int-Wert von i1 liefern
int j = Integer.parseInt("35");          // Stringkonvertierung nach int
String s1 = i2.toString();                // int-Wert von i2 in einen String umwandeln
String s2 = Integer.toString(74);        // Umwandlung einer int-Zahl in einen String
```

- Die Wrapper-Klassen besitzen keine Methoden zum Ändern des gespeicherten Wertes (Setter-Methoden). Durch die Verwendung von Wrapper-Klassen können keine Parameter an Methoden übergeben werden, die veränderlich sind



# Übersicht: Konvertierung zwischen verschiedenen Datentypen, am Beispiel von int / Integer / String



- Konstruktoren mit Typumwandlungen sind hier nicht aufgeführt.
- Bei Klassen-Methoden ist die Klasse angegeben.
- Bei Instanzmethoden wird über eine entspr. Objektreferenz aufgerufen.

# Autoboxing

- Die Konvertierung in ein Objekt der entsprechenden Wrapper-Klasse wird **Boxing** genannt.
- Die Rückumwandlung in einen primitiven Datentyp heißt **Unboxing**.
- Ab Java 5 gibt es eine automatische Umwandlung (**Autoboxing**).

 **Ab Java 1.5**

	Ausführliche Schreibweise	Vereinfachte Schreibweise mit Autoboxing
	<code>int i = 10;</code>	<code>int i = 10;</code>
<b>Boxing</b>	<code>Integer j = Integer.valueOf(i);</code>	<code>Integer j = i;</code>
<b>Unboxing</b>	<code>int k = j.intValue();</code>	<code>int k = j;</code>