



# Programmierungsmethodik 1

## Programmiertechnik

Wrapperklassen, Strings

- Einführung: Fehler + Testen
- Testen mit JUnit
- Fehlertypen
- Platzhalterobjekte

Ausblick für heute

- Ich möchte einen einfachen Datentyp über einen Referenztyp repräsentieren.
- Ich muss Zeichenketten verarbeiten.

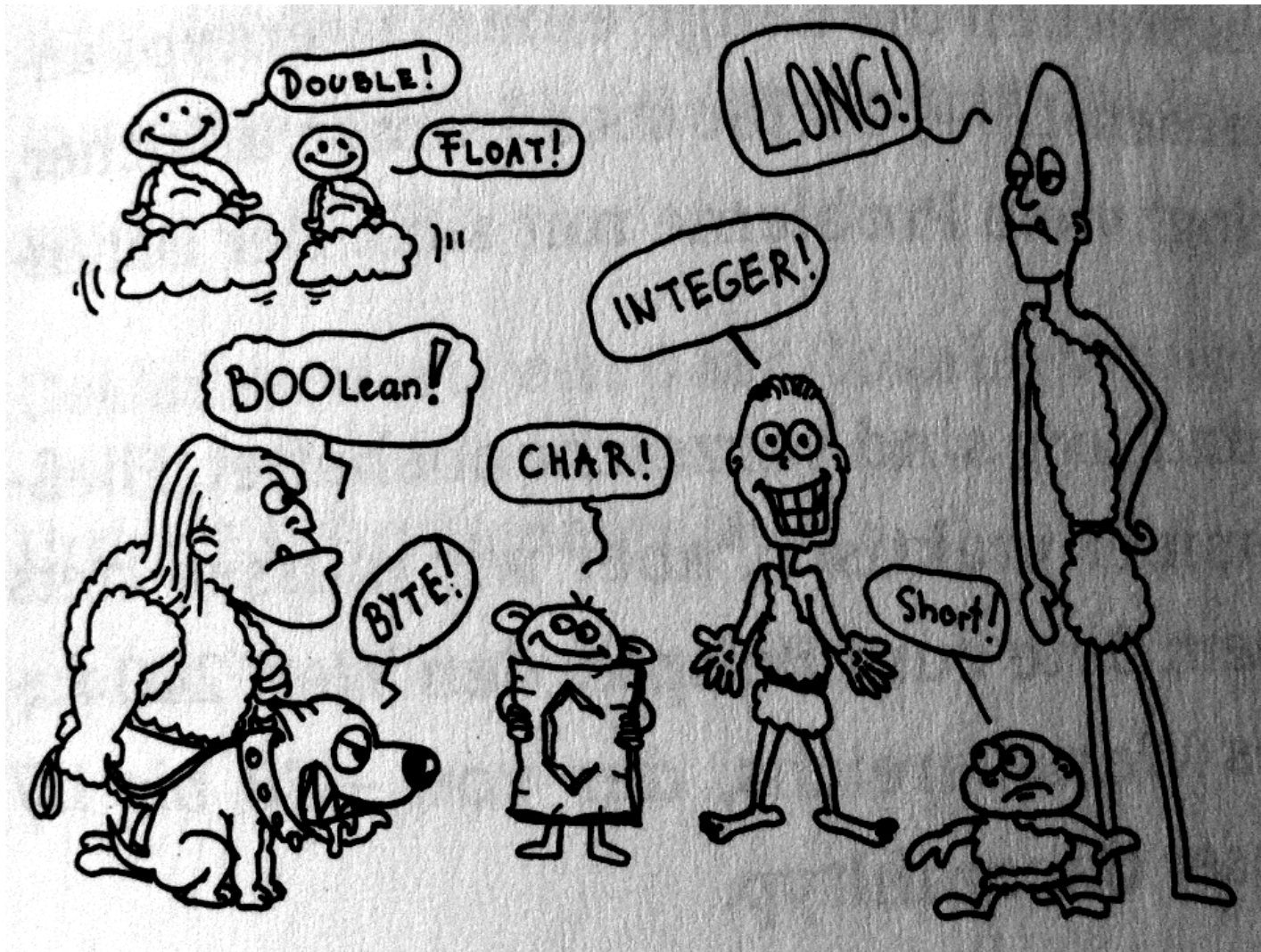
# Agenda



- Wrapperklassen
- Strings

Wrapperklassen

# Wrapperklassen



- Unterscheidung
  - primitive Datentypen vs. Referenztypen
- manchmal notwendig:
  - primitive Daten als Referenzobjekte ablegen
- Lösung
  - Wrapperklassen für einfache Datentypen
- Wrapperklassen sind Referenzklassen, die die primitiven Datentypen kapseln
- für jeden primitiven Datentyp gibt es eine Wrapperklasse



# Wrapperklassen für einfache Datentypen

Primitiver	Wrapperklasse
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

- implizite Wandlung zwischen primitivem Datentyp und Wrapperklassen möglich → Autoboxing
- Beispiel

```
// Autoboxing: 23 als Integer gewrapped  
Integer integer = new Integer(23);  
int i = integer.intValue();
```

- Umkehrung von Autoboxing
- Entpacken eines Referenztyps in einen primitiven Datentyp
- Beispiel

`// Auto(un)boxing: Integer-Objekt wird in primitiven  
Datentyp konvertiert`

```
Integer moreInteger = integer + 3;  
System.out.println(moreInteger);
```

- keine impliziten Typecasts
- Beispiel

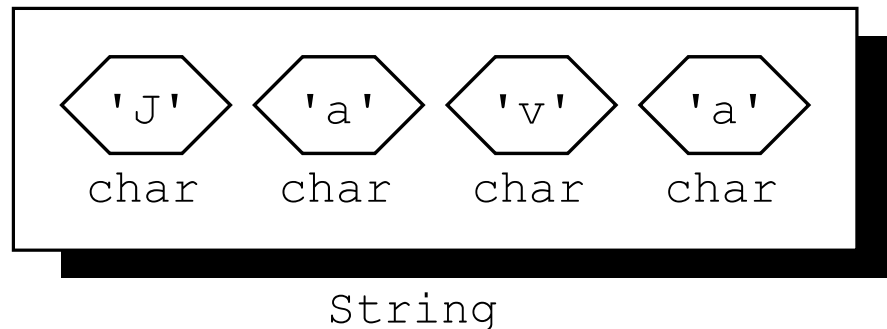
```
// Error: Impliziter Typecast nicht erlaubt  
Double d = 42; //Fehler
```

- Schreiben Sie ein Programm, das
  - eine Fließkommazahl von der Konsole einliest
  - diese Zahl als Referenztyp in einer Variable ablegt
  - den Wert der Referenzvariable um 1 erhöht
  - und das Ergebnis als Ganzzahlwert ausgibt.

Strings

- Typ `String` repräsentiert Folgen von Zeichen = Textstücke  
`String text;`

- vordefiniert, ebenso wie primitive Datentypen
- aber: kein primitiver Typ, sondern Referenztyp
  - Variablen sind Zeiger auf Objekte der Klasse String
- String ist ein Container-Objekt:
  - speichert Elemente anderer Typen
  - Elemente bei Strings: `char`
- Typ `string` legt die Anzahl `char`-Elemente nicht vorab fest






- String = Referenztyp, also Erzeugen mit new

```
String text = new String("Dies ist ein Text");
```

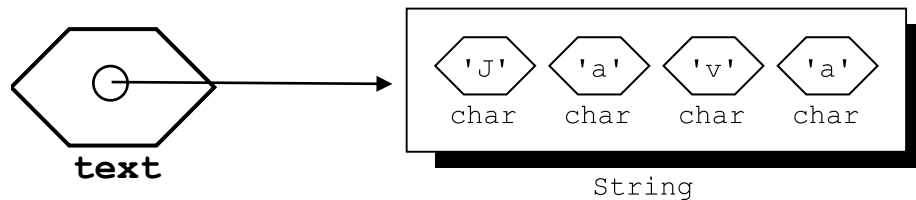
dies ist auch  
schon ein String



- Es gibt Literale für Strings (wie bei primitiven Datentypen)
  - Text zwischen Anführungszeichen
    - "Java"
    - "Sun Microsystems, Inc."
    - " "
    - "" (Leerstring)
- alle Zeichendarstellungen sind erlaubt:
  - "'a'"
  - "zwei-\n\tzeilig"
  - "M\u00FCnchen"

- Wertzuweisung an eine String-Variable durch String-Literale
  - Sonderfall: Objekterzeugung ohne new!

```
String text;  
text = "Java";
```

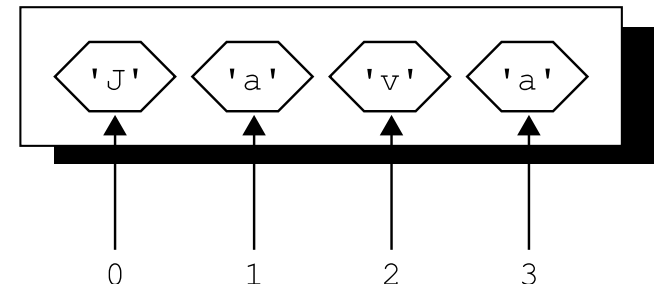


- Strings sind unveränderlich
  - Einfügen, Austauschen, Entfernen von Zeichen nicht möglich
  - siehe Wertesemantik
- "Verändernde" Methoden verändern nicht, sondern liefern neuen String zurück

```
String text = "Ein Text";
```

- Länge: 8 ('E', 'i', 'n', ' ', 'T', 'e', 'x', 't')
- Zugriff auf die Länge:
  - `int length()`
- Beispiel
  - `int laenge = text.length();`

```
String text = "Java";
```



- Methode `char charAt(int index)`
  - liefert ein einzelnes Zeichen aus einem String
- zulässige Werte für den Index: 0, 1, ..., Länge– 1
- $\text{Index} < 0$  oder  $\text{Index} \geq \text{Stringlänge}$ :
  - Indexfehler: Exception (Ausnahme-Fehler)
- hier:

kommt  
später

`text.charAt(2) → 'v'`

- finde das (erste) Vorkommen eines Zeichens

```
int indexOf(char c)
```

- Beispiel

```
String text = "Dies ist ein Text.";
int index = text.indexOf('i'); // → 1
```

- Index der letzten Fundstelle von eines Zeichens

```
int lastIndexOf(char c)
```

- funktioniert auch mit String als Parameter (→ Überladene Methoden)

- Beispiel

```
int index = text.indexOf("is"); // → 5
```

- Beispiele
  - Leerzeichen am Anfang und Ende entfernen  
`String trim()`
  - Teilstring von from bis to liefern
    - hier gibt es weitere Varianten  
`String substring(int from, int to)`
  - alles zu Groß- (bzw.) Kleinbuchstaben  
`String toUpperCase()`  
`String toLowerCase()`



- Schreiben Sie ein Programm, das für einen beliebige Eingabezeichenkette folgende Veränderungen vornimmt:
  - Ersetzen: e -> "E"
  - Entfernen aller Leerzeichen an Anfang und Ende
  - Anfügen der Länge der Eingabezeichenkette am Ende
- Beispiel
  - " Mein Heim! " -> "MEin HEim13"

## String toString()

- wird verwendet zur Erzeugung eines Textes, der Objekt beschreibt
- falls nicht implementiert:
  - es gibt immer automatisch eine Standard-Implementierung

```
Bruch bruch = new Bruch(23, 42);  
System.out.println(bruch.toString());  
// → Bruch@18fe7c3 (Standard-Implementierung)
```

- besser: Methode für jede eigene Klasse implementieren

```
public class Bruch {  
    public String toString(){  
        return zaehler + "/" + nenner;  
    }  
    ...  
}  
  
Bruch bruch = new Bruch(23, 42);  
System.out.println(bruch.toString()); // → 3/5
```

- weitere Vereinfachung: `.toString()` muss nicht angegeben werden, wenn der Compiler ein String-Objekt erwartet

```
System.out.println(bruch.toString());  
    verhält sich wie  
System.out.println(bruch);
```

- Überladener Operator +
- Strings können "addiert" werden = Konkatination
- Beispiel

```
String text1 = "Dies ist";
```

```
String text2 = " ein Text."
```

```
System.out.println(text1 + text2); // → Dies ist ein  
Text.
```

- Strings sind Referenztypen
- Operator `==` prüft Identität von String-Objekten, nicht den Inhalt
  - `"hello" == ("hell" + "o") → false`
- Test auf (inhaltliche) Gleichheit von Strings immer mit
  - `boolean equals(String andererString)`
  - also:
    - `"hello".equals("hell" + "o") → true`
- Vergleich ohne Berücksichtigung von Groß- und Kleinschreibung
  - `boolean equalsIgnoreCase(Object o)`
- Methode `equals` muss nicht selber implementiert werden
  - wie Standard-`toString()`-Methode

Begründung: Basisklasse  
Object (später)



- alphabetischer Vergleich
  - erst jeweils erste beide Zeichen
  - falls gleich: je zweite Zeichen
  - falls gleich: ...

```
int compareTo(String andererString)
```

- Ergebnis
  - $< 0$  Dieser String alphabetisch vor dem Argument (das erste unterschiedliche Zeichen ist kleiner)
  - $= 0$  Dieser String gleich dem Argument
  - $> 0$  Dieser String alphabetisch hinter dem Argument

- Beispiel

```
String eva = "Eva Zwerg";
```

```
String adam = "Adam Riese";
```

```
boolean vergleich = eva.compareTo(adam) < 0; // → false
```

- Überprüfen, ob ein String einen anderen beinhaltet  
`boolean contains(String andererString)`
- Beispiel  

```
String text = "Dies ist ein String";  
text.contains("st ei"); // → true  
text.contains("dein"); // → false
```



- Schreiben Sie ein Programm, das in zwei Strings den längsten gemeinsamen Teilstring findet.
- Beispiel

```
laengsterGemeinsamerTeilstring("Tischlerei", "Fische");  
// → "isch"
```

- Erinnerung:
  - Strings sind unveränderlich
  - falls ein Programm viele Strings verwendet: viele Objekte, viel Speicher, ggf. Performance-Verlust
- daher manchmal wünschenswert
  - veränderliche Strings
  - Lösung
    - Klasse `StringBuilder`
    - $\approx$  veränderliche Strings

- leider keine Sonderbehandlung für `StringBuilder`
  - keine Stringlitterale
  - kein Operator `+`
- einer der Konstruktoren: `StringBuilder(String str)`
  - einfache Konvertierung:

```
String text = "Java Compiler"
```

```
StringBuilder textPuffer = new StringBuilder(text);
```

- Methoden
  - erzeugen kein neues Objekt, sondern modifizieren das eigene Objekt (`this`)
  - liefern zusätzlich eine Referenz auf das eigene Objekt zurück!
- Methoden (Auszug)
  - fügt hinten das Zeichen `zeichen` an (überladen für alle Typen).  
`StringBuilder append(char zeichen)`
  - schiebt das Zeichen `zeichen` an Index `index` ein (überladen für alle Typen). Der Rest rutscht nach hinten  
`StringBuilder insert(int index, char zeichen)`
  - löscht den Teilstring ab Index `von` bis ausschließlich Index `bis`. Der Rest rutscht nach vorne  
`StringBuilder delete(int von, int bis)`

- Zusammensetzung aus Zeichenketten und Zahlen: Formatierung wünschenswert
- Beispiel
  - Text + Fließkommazahl (gerundet auf zwei Nachkommastellen)
- zwei Möglichkeiten (gleiches Regelwerk)
  - Erzeugen einer Zeichenkette: `String.format(...)`
  - Ausgabe einer formatierten Zeichenkette: `System.out.format(...)`  
genauer:  

```
static String format(String format, Object... args)
```

- Parameter :
  - Formatstring mit eingebetteten Formatangaben (erstes Zeichen %)
  - weitere Argumente entsprechend den Formatangaben im Formatstring (variable Argumentanzahl)
- Formatangaben (% + Buchstabe) sind Platzhalter für Werte nachfolgender Argumente
- wichtigste Platzhalter
  - Zeichenketten: %s
  - Ganzzahlen: %d
  - Fließkommazahlen: %f
- Beispiel

```
double zahl = 23.41;  
String text = "Mein Text";  
String string = String.format("%s - %f", text, zahl);  
// → "Mein Text - 23,42000"
```

- Formatierung von Ganzzahlen
  - Anzahl der Stellen als Platzhalter: %<Anzahl Stellen>d
  - Beispiel

```
int zahl = 23;
String.format("%3d", zahl); // → " 23"
```
- Formatierung von Fließkommazahlen
  - Anzahl Nachkommastellen: %.<Anzahl Nachkommastellen>d
  - Beispiel

```
double zahl = 23.42;
String.format("%.1f", zahl); // → "23,4"
```
- und viel mehr, siehe Dokumentation ...

- Geben Sie folgende Information gut lesbar auf der Konsole aus:

String name

int id

double wert

<Name> (<id>): <wert, zwei Nachkommastellen>

- Beispiel

name = "Inge";

id = 23;

Wert = 3.1415

- Ausgabe: "Inge (23): 3,14"



- Wrapperklassen
- Strings