



# Tutorium - OOP 2

Standardklassen



# Agenda

- Standardklassen
- Kleiner Vorgriff zu Generics



# Hüllklassen

Wrapperklassen

# Hüllklassen

- **Jeder elementare Datentyp** besitzt eine Hüllklasse.
  - Die **Objekte** der Hüllklasse **enthalten Werte** des eigentlichen Datentyps.
- Wrapperklassen besitzen nützliche Methoden zum Umgang mit den umhüllten Variablen.

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

# Nützliche Methoden (nicht vollständig)

- Konstruktoren

```
Integer(int val);  
Integer(String s);
```

- Methoden:

```
static Integer valueOf(String s)
```

- wandelt String in Integer Objekt um

```
static Integer valueOf(String s,  
int base)
```

- wandelt einen String eines Zahlensystems (Binär, Oktal, Hexa) in Dezimal um
- Bsp: `valueOf(539, 16);`  
liefert 1337 im Dezimalsystem

```
static int parseInt(String s)
```

- wandelt String in int Wert um

```
static String toBinaryString (int i)
```

```
static String toOctalString (int i)
```


```
static String toHexString (int i)
```

- Methoden der Klasse Integer erzeugen aus Dezimalzahl einen String im Zahlensystem

# Autoboxing

- Vereinfacht das Umwandeln von Werten der Datentypen zu Objekten der Hüllklassen
- Boxing: Zuweisen eines Datentyp-Wert zu einem Wrapperobjekt
- Unboxing: Umwandeln des Werts eines Wrapperobjekts zu Standarddatentyp

```
int i = 1337;  
Integer iObj = i;    // boxing  
i = iObj;            // unboxing
```



# String(Builder)

# String

- String selbst ist eine „Hüllklasse“ für Zeichenketten. Es existiert aber kein Basisdatentyp
- String hat einen **Haufen an nützlichen Methoden** (siehe Skript).
- Spielt am besten etwas mit den verschiedenen Methoden der Hüllklassen und explizit String herum!



# String – Nützliche Methoden

- `String concat(String s)`
  - `s` wird an den String angehängen (alternativ konkatenieren mit „+“)
  - **Achtung!** Jedes Mal wird hier ein neues String Objekt erzeugt!
- `boolean equals (Object obj)`  
`boolean equalsIgnoreCase (String s)`
  - Der String für den die Methode aufgerufen wird, wird mit `s` verglichen
- `String substring(int start)`  
`String substring(int start, int end)`
  - Aufspalten des Strings in einen neuen Substring
  - Liefert einen neuen String, der am `x`-ten Zeichen (`start`) startet und (optional) bis zum Zeichen vor dem `end`-Zeichen geht
  - `end = 5` → es wird bis zum 4ten Zeichen ausgegeben

# Beispiel – .equals() Methode

- Hier geht es um Groß- und Kleinschreibung bei Stringvergleichen

```
String s1 = „Test“;
```

```
String s2 = „test“;
```

```
s1.equals(s2);
```

```
s1.equalsIgnoreCase(s2);
```

# StringBuilder

- StringBuilder erweitert die Funktionalität von String
- Auch StringBuilder hat viele nützliche Methoden (siehe Skript)
- Angehängt wird mit der Methode `append([datentyp] text);`
  - Es wird kein neues Objekt erzeugt! StringBuilder ist also speichereffizienter
- StringBuilder erzeugen:

```
StringBuilder sb = new StringBuilder(String)
```

# StringBuilder – Wichtige Methoden

- `int length();`
  - Gibt die Länge des Strings im StringBuilder aus
- `String toString();`
  - Umwandlung des Builders in ein Stringobjekt
- `StringBuilder append(datentyp x);`
  - Hängt x (viele Datentypen möglich) an den aktuellen String an
- `StringBuilder insert(int i, datentyp x);`
  - Fügt x an der Stelle i im aktuellen String ein

# StringBuilder – Wichtige Methoden

- `StringBuilder delete(int begin, int end)`
  - löscht die Zeichen von begin bis end(-1) (wieder ein Zeichen vor end)
- `StringBuilder replace(int begin, int end, String s)`
  - ersetzt die Zeichen von begin bis end(-1) durch s
- `char charAt(int i)`
  - gibt das Zeichen an Position i aus. Das erste Zeichen hat die Position 0
- `char setCharAt(int i, char c)`
  - ersetzt das Zeichen and der Stelle i durch c

# StringTokenizer

- StringTokenizer wird genutzt, um Strings anhand von Trennzeichen aufzuspalten
- Konstruktor:
  - `StringTokenizer(String s, String delim);`
  - s ist der String, der getrennt werden soll, delim ist (optional!) der String, an dem getrennt werden soll (Z.B. `"`, `"`)

# Beispiel Tokenizer

- Der folgende Text (CSV) soll aufgespalten werden

```
Max Mustermann, 12345, 02161 8408923, .....
```

- Der Text ist in einem String gespeichert und ", " (Komma und Leerzeichen) werden als Trenner genutzt

```
StringTokenizer(s, ", ");
```

- Wir erhalten:

```
Max Mustermann  
12345  
02161 8408923  
...
```



# Code Beispiele

Besprochen in der IDE





# Datum und Zeit

# Date und SimpleDateFormat

- SimpleDateFormat verwendet ein Pattern aus den Zeichen rechts, um Datum und Zeit zu formatieren

- Beispielcode:

```
Date jetzt = new Date();  
SimpleDateFormat f = new  
    SimpleDateFormat("dd.MM.yyyy");  
String s = f.format(jetzt);
```

Pattern	Bedeutung
d	Tag als Zahl, dd zweistellig
M	Monat als Zahl, MM zweistellig, MMM abgekürzter Text, MMMM Volltext
yy	Zweistelliges Jahr, yyyy vierstellig
E	Tag als abgekürzter Text, EEEE Volltext
H	Stunde(0-23), HH zweistellig
m	Minute, mm zweistellig
s	Sekunde, ss zweistellig

# Übung

- Implementiert eine Klasse `AktuellesDatum` mit einer Klassenmethode, die beim Aufruf die aktuelle Uhrzeit und das Datum im folgenden Format ausgibt:

`Do, 10.6.2021 - 11:45:31`



# List / Vector

# List

- `List` ist ein Interface, dass eine dynamische Datenstruktur umsetzt
  - `ArrayList` und `Vector` implementieren beide `List`
- In Klassen, die `List` implementieren, können beliebig viele Werte eines Datentyps gespeichert werden
- `Vector` VS `ArrayList`
  - Nutzt im Normalfall `ArrayList`, da `ArrayList` performanter ist (und sicher für threading)

# Vector und ArrayList

- Konstruktoren:

```
Vector<Datentyp> vectorName = new Vector<>();
```

```
ArrayList<Datentyp> arrayListName = new ArrayList<>();
```

- Datentyp ist String, eine Hüllklasse oder allgemein eine Klasse! Kein einfacher Datentyp wie int!

```
Vector <Integer> integerVector = new Vector<>();
```

```
Vector <Konto> kontoVector = new Vector<>();
```

```
Vector <int> intVector = new Vector<>();
```

# Nützliche Methoden des Interface List

- `void add (int i, Object obj)`
  - Object an Stelle i der Liste einfügen
- `Object get (int i)`
  - Objekt an Stelle i ausgeben
- `Object remove (int i)`
  - Objekte an Stelle i entfernen
- `boolean remove (Object obj)`
  - entfernt ersten Wert von „obj“ der auftritt
- `void clear ()`
  - leert die Liste
- `int size ()`
  - gibt Anzahl der Elemente in der List zurück



# Map / Hashmap



# HashMap

- Implementiert funktionell eine Tabelle mit einem Schlüssel (`key`), dem ein Wert (`value`) zugewiesen ist
- Vergleichbar mit einem Wörterbuch
  - Dem Schlüssel „Buch“ ist der Wert „book“ zugewiesen

Deutsch	English
Buch	book
Apfel	apple
laufen	to run

# HashMap

- Kontruktor:

```
HashMap<DatentypKey, DatentypValue> h1 = new HashMap<>();
```

- Die Datentypen von Key und Value können sich unterscheiden!

# Nützliche Methoden von HashMap

- `boolean containsKey (Object key)`
  - Gibt zurück, ob dieser Schlüssel vorhanden ist
- `boolean containsValue (Object value)`
  - Gibt zurück, ob dieser Wert vorhanden ist
- `Object get (Object key)`
  - liefert den Wert zum eingegebenen Schlüssel (key)
- `Object put (Object key , Object value)`
  - Fügt ein Schlüssel/Wert Paar zur HashMap hinzu (falls Schlüssel vorhanden, wird Wert ausgegeben)
- `Object remove (Object key)`
  - entfernt den Schlüssel und gibt den Wert zum Schlüssel zurück
- `int size ()`
  - gibt die Anzahl der Schlüssel/Wert Paare als int zurück



# Code Beispiele

Besprochen in der IDE



# Aufgaben

Heute mal etwas anders (nur Aufgaben aus dem Skript)



# Ende für heute