Übersicht

- Beispiel: Klasse Einfamilienhaus

UML

Einfamilienhaus
<<Attribute>> - haustyp : String - besitzer : String - adresse : String - wohnflaeche : double - anzBaeder : int -verkaufsPreis : double
<<Methoden>> + verkaufspreisAnzeigen() : double …

Java

```
public class Einfamilienhaus
{
  // Attribute
  /** Der Haustyp des Hauses. */
  private String haustyp;

  /** Der Besitzer des Hauses. */
  private String besitzer;

  /** Die Adresse des Hauses. */
  private String adresse;

  /** Die Wohnfl&auml;che des Hauses. */
  private double wohnflaeche;

  /** Die Anzahl der B&auml;der im Haus. */
  private int anzBaeder;

  /** Der Verkaufspreis des Hauses. */
  private double verkaufsPreis;

  // Methoden
  ...
}
```

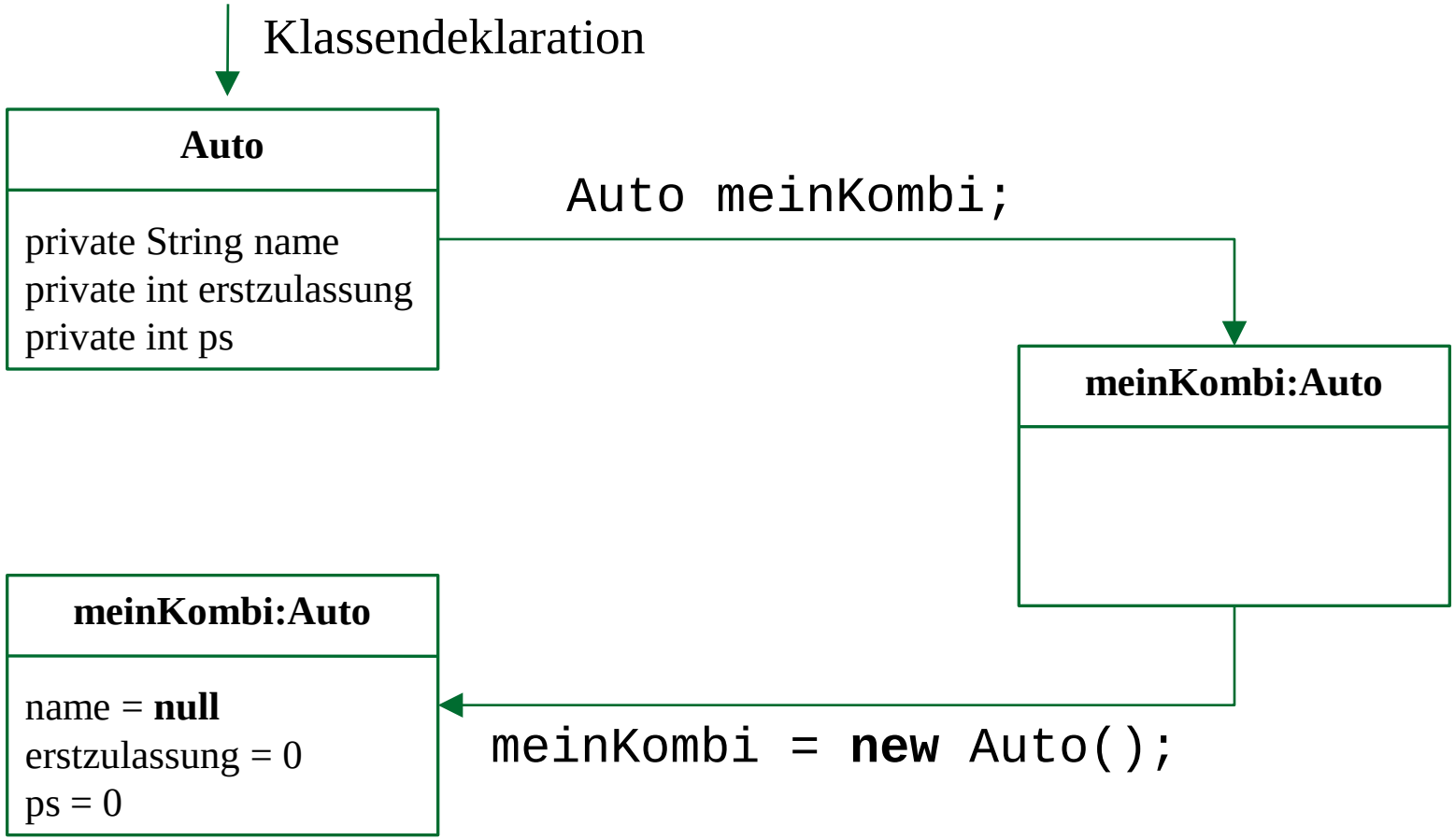
- Um ein Objekt der Klasse Einfamilienhaus zu erzeugen, muss man an der ensprechenden Stelle im Programm eine Variable vom Typ der Klasse deklarieren und ihr mit Hilfe des **new**-Operators ein neu erzeugtes Objekt zuweisen:

```
Einfamilienhaus efh4;
efh4 = new Einfamilienhaus();
```

- 1. Anweisung: Klassische Deklaration einer Variablen vom Typ der Klasse (Objekttyp). Anstelle eines primitiven Datentyps wird hier der Name einer zuvor definierten Klasse verwendet.
- Die Variable `efh4` wird angelegt, darauf steht nun eine *Referenz* (*Zeiger*) auf einen speziellen Speicherplatz für das Objekt (das noch nicht existiert).
- 2. Anweisung: Generiert das Objekt mittels des **new**-Operators.
- Da Arrays auch Objekte sind, wissen Sie nun auch, warum man bei der Erzeugung eines Arrays den **new**-Operator benötigt.

- Nach der Generierung eines Objekts haben alle Attribute mit primitiven Datentypen zunächst ihre entsprechenden Standardwerte (siehe Arrays), Attribute mit Objekttypen haben den Standardwert **`null`**, die "leere Referenz".
- Zugriff auf Zustand / Methoden eines Objekts:
  - **`private`**: Auf diese Attribute / Methoden kann außerhalb der Klasse *nicht* zugegriffen werden.
  - **`public`**: Auf diese Attribute / Methoden kann außerhalb der Klasse über die Punktnotation zugegriffen werden: Wäre beispielsweise das Attribut **`int`** `verkaufsPreis` mit der Sichtbarkeit **`public`** versehen, so könnte man mit `efh4.verkaufsPreis` dessen aktuellen Wert lesen und diesen auch verändern (durch eine Wertzuweisung).
- Offensichtlich widerspricht der direkte Zugriff auf den Zustand (Attribute) eines Objekts dem Prinzip der Kapselung. Daher ist es guter OO Programmierstil, Attribute grundsätzlich als **`private`** zu deklarieren.

- Noch einmal bildlich der Vorgang der Objekterzeugung am Beispiel einer Klasse `Auto`:



- Nochmals zu Arrays:
  - Eigentlich (im Sinne der Kapselung) sollten Attribute als **`private`** vereinbart werden.
  - Was verbirgt sich hinter `beispielArray.length`?
  - Eigentlich ist das ein konstantes *Attribut*, auf das man von außerhalb der Klasse zugreifen kann, also **`public`** ist.
- Tatsächlich sind Arrays besondere Objekte, zu denen es keine "echte" Klassendefinition gibt.
- Daher unterscheidet sich die "Schnittstelle" (die **`public`** Methoden / Attribute) von Arrays zu den Schnittstellen klassischer Objekte.
- Arrays werden ansonsten genauso behandelt wie "normale" Objekte.

- Was bedeutet, dass zwei Objekte gleich sind?
- Intuitiv: Ihre Attribute haben dieselben Werte, d.h. sie haben denselben Zustand.

```
Einfamilienhaus villaKahn;
villaKahn = new Einfamilienhaus();
Einfamilienhaus hausBender;
hausBender = new Einfamilienhaus();

boolean vergleich = villaKahn == hausBender; // (*)

villaKahn = hausBender;

vergleich = villaKahn == hausBender; // (**)
```

- ***Was ist der Wert der Variablen*** `vergleich` ***an der Stelle (*)?***

- Der Wert der Variablen `vergleich` an der Stelle (*) ist **`false`**!
- **Warum?** Sowohl das Objekt `villaKahn` als auch das Objekt `hausBender` haben doch als Attributwerte die Standardwerte, also haben sie insgesamt denselben Zustand.
- Richtig! Aber wir sind einem call-by-reference-Effekt bei Objekten auf den Leim gegangen.
- `villaKahn` ist eine Variable, die die Referenz zu einem speziellen Objekt speichert. Ebenso `hausBender`. Beide speichern unterschiedliche Referenzen, denn die Variablen repräsentieren ja unterschiedliche Objekte.
- Auf den beiden Zetteln stehen daher unterschiedliche "Speicherplätze"
- Vorsicht also: bei Objekttypen stehen auf den Variablenzetteln nicht die tatsächlichen Objekte, sondern die entsprechenden Referenzen darauf.
- Der Wert der Variablen `vergleich` an der Stelle (**) ist dagegen **`true`**.

- **Achtung**: Es gibt also zwei "Arten" von Gleichheit von Objekten bzw. Objektvariablen (Variablen mit Objekttyp):
- **Gleichheit**: Der Zustand der entsprechenden Objekte beider Objektvariablen ist gleich. Dies bezieht sich also auf die Objekte.
- **Identität**: Beide Objektvariablen verweisen auf die gleiche Speicheradresse. Dies beszieht sich also auf die Variablen.
- Der Operator == prüft den zweiten Fall (Identität). Er kann also nicht dazu benutzt werden, abzufragen, ob die Objekte zweier Objektvariablen gleich bzgl. ihres Zustands sind.
- Dies wird oft übersehen und ist daher eine häufige Fehlerquelle!
- Um Gleicheit von Objekten zu prüfen, benötigt man andere Konstrukte, die wir später kennenlernen werden.

- Die Methoden einer Klasse definieren das Verhalten der Objekte der Klasse.
- Innerhalb eines Methodenrumpfs einer Klasse hat man Zugriff auf alle Attribute und Methoden der Klasse (auch auf den **`private`**-Teil).
- Methoden können wiederum von außen sichtbar (**`public`**) oder nicht sichtbar (**`private`**) sein. Die Methoden, die das Verhalten der Objekte spezifizieren, sollten alle sichtbar sein. Darüberhinaus kann es natürlich auch noch nicht sichtbare Hilfsmethoden geben.
- Kapselung:
  - Der Zustand (alle Attribute) des Objekts sind gekapselt (nicht sichtbar, **`private`**).
  - Nur über die sichtbaren (**`public`**) Methoden kann man auf den Zustand zugreifen bzw. kann ihn verändern.
  - Zugriff auf und Veränderung von Objektzuständen ist damit kontrolliert und wohldefiniert.

- Beispiel

```
public class Auto
{
  /** Der Modellname des Autos. */
  private String name;
  /** Das Jahr der Erstzulassung des Autos. */
  private int erstzulassung;
  /** Die Motorleistung des Autos in PS. */
  private int ps;

  /**
   * Gibt das Jahr der Erstzulassung des Auto-Objekts zur&uuml;ck.
   * @return  das Jahr der Erstzulassung des Autos.
   */
  public int getErstzulassung()
  {
    return erstzulassung;
  }

  /**
   * Ver&auml;ndert das Jahr der Erstzulassung des Auto-Objekts.
   * @param jahr  das Jahr der Erstzulassung des Autos.
   */
  public void setErstzulassung(int jahr)
  {
    erstzulassung = jahr;
  }

  /**
   * Berechnet das Alter (in Jahren) des Auto-Objekts.
   * @param aktuellesJahr  das aktuelle Jahr, z.B. 2007.
   * @return  das Alter des Autos.
   */
  public int alter(int aktuellesJahr)
  {
    return aktuellesJahr - erstzulassung;
  }
```

- Alle Methoden sind sichtbar und haben ihrerseits Zugriff auf alle nicht sichtbaren Attribute der Klasse `Auto`.
- Kapselung: Das Jahr der Erstzulassung des Autos ist in diesem Beispiel nur über die Methoden `getErstzulassung` bzw. `setErstzulassung` von außerhalb lesend / schreibend zugreifbar.
- Dieser Zugriff ist durch die beiden Methoden wohldefiniert.

- Beispiel:

```
public class Konto {

        private int kontonummer;
        private double kontostand;
        private double dispolimit;
        …
        public void abheben(double betrag) {
                if(kontostand – betrag > dispolimit) {
                        kontostand = kontostand – berag;
                } else {
                        System.out.println(“Abheben nicht moeglich!!!“);
                }
        }
}
```

Die Veränderung des Zustands durch das Geldabheben ist nur über die Methode möglich abheben und damit wohldefiniert. Ist die Methode einmal richtig implementiert, kann man sie überall wiederverwenden. Dies ist offenbar ein Schutz vor fehlerhaftem Verhalten.

- Definierter Zugriff auf Attribut kontostand

```
public class Konto {

        private int kontonummer;
        private double kontostand;
        private double dispolimit;
        …
        public double kontostandAbrufen() {
                // Autorisierung pruefen
                boolean ok = ATM.autorisierungPruefen(kontonummer)
                // statische Methode der Klasse ATM, die die Autorisierung
                // des Kontos am ATM prueft und ein boolean zurueckgibt
                if(ok) {
                        return kontostand;
                } else {
                        … <Ziehe Karte ein und schicke Benachrichtigung>
                }
        }
}
```

- Die **`public`** Methoden können wie bereits erwähnt mit der Punktnotation aufgerufen werden.

```
Auto golf = new Auto();
golf.setErstzulassung(1998);
int e = golf.getErstzulassung(); // Wert von e: 1998
int a = golf.alter(2012); // Wert von a: 14
```

- Wie man auf der Folie davor sieht, darf eine Methode auf die Attribute (und auch auf die anderen Methoden) der Klasse ohne Punktnotation zugreifen.
- Tatsächlich bezieht der Compiler alle Variablen x, die nicht in Punktnotation verwendet werden, auf das Objekt **`this`**, d.h. x wird eigentlich als **`this.`**x interpretiert.
  Ausnahme: Es gibt eine lokale Variable (z.B. in einem Methodenrumpf), die genauso heißt wie ein Klassenattribut.

- Bei **this** handelt es sich um eine Art Objektvariable (also eine Referenz auf ein Objekt), die beim Anlegen eines Objekts automatisch generiert wird.
- **this** zeigt auf das aktuelle Objekt und wird dazu verwendet, die eigenen Methoden und Attribute anzusprechen.
- **this** kann auch explizit verwendet werden:

```
public int alter(int aktuellesJahr)
{
        return aktuellesJahr - this.erstzulassung;
}
```

- **this** muss explizit verwendet werden, wenn gleichnamige lokale Variablen (z.B. Parameter) verwendet werden.
- Die Verwendung von **this** auch da, wo es von der Syntax nicht verlangt wird, ist guter Stil, da die Lesbarkeit des Programmes verbessert wird.

- Die *Signatur* einer Methode setzt sich zusammen aus
  - dem Namen der Methode,
  - der Reihenfolge und Typen der Eingabeparameter,
  - dem Ausgabetyp.
- Methoden können grundsätzlich den selben Namen habe, solange sie sich in ihrer Parameterliste unterscheiden. Man spricht in diesem Fall von *Überladen*.
- Beispiel (natürlich etwas sinnfrei):

```
public int alter(int aktuellesJahr)
{
   return aktuellesJahr - this.erstzulassung;
}
public int alter(double aktuellesJahr)
{
   return (int) aktuellesJahr - this.erstzulassung;
}
```

- Dagegen sind die beiden Methoden

```
public int alter(int aktuellesJahr)
{
   return aktuellesJahr - this.erstzulassung;
}
```

und

```
public double alter(int aktuellesJahr)
{
   return (double) (aktuellesJahr - this.erstzulassung);
}
```

nicht verschieden! **Warum?**

- Bei unserer Beispielklasse `Auto` ist das Jahr der Erstzulassung von außen über die Methoden `setErstzulassung` veränderbar.
- Dies ist vermutlich nicht besonders glücklich: Der Wert des Attributs `erstzulassung` sollte einmal initialisiert werden können, dann aber nicht mehr verändert werden können.
- D.h., die Methode `setErstzulassung` sollte nicht zur Verfügung stehen.
- Dann kann das Attribut `erstzulassung` aber auch nicht verändert werden, schließlich ist es von außerhalb nicht sichtbar.
- Ist der Wert des Attributs dann immer nur mit dem Standardwert initialisiert? Kann der Wert irgendwie anders initialisiert werden?

- Die Antwort auf diese Fragen sind die sog. *Konstruktoren*.
- Konstruktoren sind spezielle Methoden, die zur Erzeugung und Initialisierung von Objekten aufgerufen werden können.
- Konstruktoren sind Methoden *ohne* Rückgabwert (nicht einmal **`void`**), die als Methodenname den Namen der Klasse erhalten.
- Konstruktoren können eine beliebige Anzahl von Eingabe-Parametern besitzen und überladen werden.
- Ein Konstruktor, der z.B. das Jahr der Erstzulassung als Parameter übergeben bekommt und das Attribut `erstzulassung` entsprechend initialisiert, löst also unser oben angesprochenes Problem.

```
public class Auto
{
  /** Der Modell-Name des Autos */
  private String name;
  /** Das Jahr der Erstzulassung des Autos */
  private int erstzulassung;
  /** Die Leistung des Autos in PS */
  private int ps;

  /**
   * Konstruktor. Generiert ein neues Objekt Auto.
   * @param name            der Name des Modells.
   * @param erstzulassung   das Jahr der Erstzulassung.
   * @param ps              die Leistung in PS.
   */
  public Auto(String name, int erstzulassung, int ps)
  {
    this.name = name;
    this.erstzulassung = erstzulassung;
    this.ps = ps;
  }

  // Methoden (jetzt ohne "setErstzulassung")
  ...
}
```

- Da `erstzulassung` ein Attribut ist, dessen Wert sich nach der Initialisierung nicht mehr ändern soll, zeigt man das im Code am besten durch Verwendung einer Konstante an:

```
public class Auto {
  /** Der Modell-Name des Autos */
  private String name;
  /** Das Jahr der Erstzulassung des Autos */
  private final int ERSTZULASSUNG;
  /** Die Leistung des Autos in PS */
  private int ps;

  /**
   * Konstruktor. Generiert ein neues Objekt Auto.
   * @param name           der Name des Modells.
   * @param erstzulassung  das Jahr der Erstzulassung.
   * @param ps             die Leistung in PS.
   */
  public Auto(String name, int erstzulassung, int ps)
  {
    this.name = name;
    this.ERSTZULASSUNG = erstzulassung;
    this.ps = ps;
  }

  // Methoden (jetzt ohne "setErstzulassung")
  ...
}
```

- Aufruf:

```
Auto golf;
golf = new Auto("Golf",1998,102);
int e = golf.getErstzulassung(); // Wert von e: 1998
int a = golf.alter(2007); // Wert von a: 9
```

- Wird kein expliziter Konstruktor deklariert, gibt es einen *Default-Konstruktor*, der *keine* Eingabeparameter hat, um überhaupt ein Objekt zu erzeugen (den Default-Konstruktor haben wir übrigens bisher benutzt – `Auto()` in unserem Beispiel).
- Wird mindestens ein expliziter Konstruktor vereinbart, steht *kein* Default-Konstruktor zur Verfügung!
- Möchte man trotzdem einen Konstruktor ohne Eingabeparameter zur Verfügung stellen, muss dieser explizit programmiert werden!

Bemerkungen:

- Während die Methoden eine wohldefinierte Schnittstelle zur Veränderung von Objektzuständen zur Verfügung stellen, stellen die Konstruktoren eine wohldefinierte Schnittstelle zur Erzeugung von Objekten dar.
- Neben den Konstruktoren gibt es noch die Möglichkeit, initiale Attributwerte in der Klassendefinition direkt zu vereinbaren (Die Attributsdefinitionen sehen dann so aus wie Variablenvereinbarungen mit Initialisierung).

Übersicht

6.4 Zusammenspiel von imperativen und oo Aspekten in Java

- Wie passen unsere imperativen Konstrukte aus früheren Teilen der Vorlesung hier dazu?
- Es gibt in Java *statische* und *nicht-statische* Elemente einer Klasse.
- Die statischen Elemente (Methoden oder Attribute) sind mit dem Schlüsselwort **`static`** gekennzeichnet.
- Fehlt **`static`**, ist das entsprechende Element (Methode / Attribut) nicht-statisch.
- Statische Elemente existieren *unabhängig* von Objekten, wogegen nicht-statische Elemente an die Existenz von Objekten gebunden sind.
- Werden statische Variablen von einem Objekt verändert, ist diese Veränderung auch in allen anderen Objekten der gleichen Klasse sichtbar.

6.4 Zusammenspiel von imperativen und oo Aspekten in Java

- Ein imperatives Programm `Programm` in Java besteht aus einer Klassendefinition für die Klasse `Programm` mit einer statischen `main`-Methode, die unabhängig von Objekten der Klasse `Programm` existiert (es gibt insbesondere keine nicht-statischen (Objekt-)Attribute und Methoden und es ist auch nicht vorgesehen Instanzen der Klasse (Objekte der Sorte) `Programm` zu erzeugen, auch wenn dies theoretisch möglich wäre).
- Die Klasse `Auto` in unserem vorherigen Beispiel spezifiziert ein nicht-statisches Attribut `name`.
- Dieses Attribut existiert nur dann, wenn es mindestens ein Objekt der Klasse `Auto` gibt. Für jedes existierende Objekt der Klasse `Auto` existiert ein Attribut `name` mit evtl. unterschiedlichen Werten
- Statische und nicht-statische Elemente können "nebeneinander" verwendet werden. Für beide stehen die selben Konzepte zur Spezifikation der Sichtbarkeit zur Verfügung.

6.4 Zusammenspiel von imperativen und oo Aspekten in Java

Beispiel (Kontoverwaltung):

- Klasse `Konto` spezifiziert die Konten einer Bank.
- Für jedes neueröffnete Konto wird eine neue, fortlaufende Kontonummer vergeben.
- **Wie kann man dies realisieren?**
- Z.B. mit einer statischen Variablen `aktuelleKNR`, die bei jeder neuen Kontoeröffnung gelesen wird und anschließend inkrementiert wird. Dieses Attribut sollte **`private`** sein, damit nur die Objekte der Klasse darauf zugreifen können.
- D.h. die Klasse `Konto` wird neben den nicht-statischen Attributen, die für jedes neue Objekt neu angelegt werden (z.B. für den Namen des Kontoinhabers) auch das statische Attribut `aktuelleKNR` haben, das unabhängig von den existierenden Objekten der Klasse `Konto` existiert und verwendet werden kann.

6.4 Zusammenspiel von imperativen und oo Aspekten in Java


```
public class Konto
{
  /* ** Statische (objekt-unabhaengige) Attribute ** */
  private static int aktuelleKNR = 1;

  /* ** Nicht-statische (objekt-abhaengige) Attribute ** */
  private String kundenName;
  private double kontoStand;
  private final int KONTO_NR;
  ... // weitere Attribute

  /* ** Konstruktor ** */
  public Konto(String kundenName)
  {
    this.kundenName = kundenName;
    kontoStand = 0.0;
    KONTO_NR = aktuelleKNR;
    aktuelleKNR++;
  }

  /* ** Methoden ** */
  ...
}
```

6.4 Zusammenspiel von imperativen und oo Aspekten in Java

- Zu jedem primitiven Datentyp in Java gibt es eine korrespondierende (sog. *Wrapper-*) Klasse, die den primitiven Typ in einer OO Hülle kapselt.
- Es gibt Situationen, bei denen man diese Wrapper-Klassen anstelle der primitiven Typen benötigt, z.B. werden in Java einige Klassen zur Verfügung gestellt, die eine (dynamische) Menge von beliebigen Objekttypen speichern können (als Alterative zum Array). Um darin auch primitive Typen ablegen zu können, benötigt man die Wrapper-Klassen.
- Zu allen numerischen Typen und zu den Typen **`char`** und **`boolean`** existieren Wrapper-Klassen.

6.4 Zusammenspiel von imperativen und oo Aspekten in Java

Wrapper-Klasse	Primitiver Typ
Byte	**byte**
Short	**short**
Integer	**int**
Long	**long**
Double	**double**
Float	**float**
Boolean	**boolean**
Character	**char**
Void	**void**

6.4 Zusammenspiel von imperativen und oo Aspekten in Java

- Zur Objekterzeugung stellen die Wrapper-Klassen hauptsächlich zwei Konstruktoren zur Verfügung:
  - Für jeden primitiven Typ `type`: Konstruktor, der einen primitiven Wert des Typs `type` als Argument fordert, z.B.:

    ```
    public Integer(int i)
    ```

  - Zusätzlich gibt es bei den meisten Wrapper-Klassen die Möglichkeit, einen String zu übergeben, z.B.:

    ```
    public Integer(String s)
    ```

    wandelt die Zeichenkette `s` in einen Integer um, z.B. "123" in 123.
- Kapselung:
  - Der Zugriff auf den Wert des Objekts erfolgt ausschließlich lesend über entsprechende Methoden, z.B.

    ```
    public int intValue()
    ```

  - Die interne Realisierung ist dem Benutzer verborgen.
  - Insbesondere kann der Wert des Objekts nicht verändert werden.

6.4 Zusammenspiel von imperativen und oo Aspekten in Java

- Statische Elemente der Wrapperklassen sind u.a.:
  - Wichtige Literale aus dem enstprechenden Wertebereich, z.B. Konstanten

    ```
    public static int MAX_VALUE bzw.
    public static int MIN_VALUE
    ```

    für den maximal / minimal darstellbaren **`int`**-Wert
  - oder z.B. Konstanten

    ```
    public static double NEGATIVE_INFINITY bzw.
    public static double POSITIVE_INFINITY
    ```

    für –∞ und +∞
- Hilfsmethoden wie z.B.

  ```
  static double parseDouble(String s)
  ```

  der Klasse `Double` wandelt die Zeichenkette `s` in ein primitiven **`double`**-Wert um und gibt den **`double`**-Wert aus.

6.4 Zusammenspiel von imperativen und oo Aspekten in Java

- Nochmal der Typ `String`:
  - Dahinter verbirgt sich eine Klasse die allerdings einige Besonderheiten hat:
    - Ein String kann aus Literalen erzeugt werden.
    - Auf Strings ist ein Operator „+“ (Konkatenation) definiert.
    - String-Objekte sind nicht dynamisch (dazu später).
  - Außer aus Literalen kann man Strings auch durch verschiedene Konstruktoren erzeugen. Die wichtigsten sind:
    - `String()` – erzeugt ein neues String-Objekt, das den leeren String repräsentiert (ein **`char`**-Array der Länge 0).
    - `String(String original)` – erzeugt einen neuen String, der die gleiche **`char`**-Array repräsentiert wie das angegebene Original. Es wird also eine Kopie (des String-Objekts, nicht der Referenz!) erzeugt.
    - `String(char[] value)` – erzeugt einen String aus dem gegebenen Array von chars. Das neue String-Objekt ist danach unabhängig vom **`char`**-Array, d.h. Änderungen am **`char`**-Array in der Variablen `value` haben keinen Auswirkung auf den String.
    - `String(char[] value, int offset, int count)` – wie vorher, wobei man aber noch einen Ausschnitt des Arrays spezifiziert.

6.4 Zusammenspiel von imperativen und oo Aspekten in Java

- Ein einmal erzeugter String kann nicht mehr verändert werden.
- Das bedeutet, String-Objekte sind nicht dynamisch, auch wenn das manche Methoden suggerieren.
- Ein paar Methoden:
  - Ein String-Objekt gibt über seine *Länge* Auskunft durch die Methode **`int`** `length()`.

```
String s = new String("Hello, World!");
int l = s.length();
System.out.println("Laenge von \""+s+"\": "+l);
// Ausgabe:
// Laenge von "Hello, World!": 13
```

  - Die Methoden `String toLowerCase()` bzw. `String toUpperCase()` geben den String zurück, der entsteht, wenn man alle Großbuchstaben im aufrufenden Objekt durch Kleinbuchstaben ersetzt bzw. umgekehrt.
  - Die Methode String `replace(`**`char`** `oldChar, `**`char`** `newChar)` gibt den String zurück, der durch Ersetzen aller Vorkommen von `oldChar` durch `newChar` entsteht.