Programmiertechnik 1

Teil 4: Java Klassen

Pakete / Methoden / Variablen / Objekte

Prof. Dr. H. Drachenfels
Hochschule Konstanz

Version 23.0
13.2.2025

Programmiertechnik 1 - Teil 4

1) **Pakete und Klassen**
2) Klassenmethoden und Klassenvariablen
3) Blick in die Java Klassenbibliothek: Utility-Klassen
4) Instanziierbare Klassen
5) Blick in die Java Klassenbibliothek: Klassen für Wertobjekte und Entitäten
6) Eingebettete Klassen
7) Empfehlungen

Java Programme: Aufbau

Ein Java-Programm besteht aus in **Paketen** organisierten **Klassen**.

- Pakete enthalten logisch zusammengehörige Klassen und Unterpakete

  Klassen- und Unterpaketnamen müssen nur in ihrem umschließenden Paket eindeutig sein.
  Klassen ohne explizite Paketzuordnung liegen in einem unbenannten Standardpaket.

  Das Paket ***`java`*** *mit seinen Unterpaketen bildet den Kern der Java-Klassenbibliothek:*

  `java.lang` *enthält Fundamentalklassen zur Sprachunterstützung, z.B.* ***`String`***
  `java.io` *enthält Klassen für die Ein-/Ausgabe, z.B.* ***`PrintStream`***
  `java.util` *enthält nützliche Klassen für Standardprobleme, z.B.* ***`Scanner`***
  `...`

- Klassen enthalten logisch zusammengehörige Methoden, Variablen, Hilfsklassen

  Methoden sind Klassenmethoden, Instanzmethoden oder Konstruktoren

  Variablen sind Klassenvariablen, in Objekten Instanzvariablen und innerhalb der Methoden Parameter oder lokale Variablen

  Hilfsklassen sind statisch eingebettete Klassen oder innere Klassen und innerhalb der Methoden lokale Klassen

Java Pakete: Syntax (1)

- eine **Paket-Deklaration** ordnet eine Klasse einem Paket zu:

```
package Paketname;  // muss erste Anweisung in der Quelldatei der Klasse sein

public final class Klassenname {
    ...
}
```

der voll qualifizierte Klassenname lautet dann: *Paketname*.*Klassenname*

der Paketnamen darf auch untergliedert sein: *Paket*.*Unterpaket*.*Unterunterpaket*

Konvention: Paketnamen (der obersten Ebene) sollten nur aus Kleinbuchstaben bestehen

- Pakete werden im Dateisystem als geschachtelte Verzeichnisse (Ordner) realisiert:

Paket/
└ *Unterpaket*/
└ *Unterunterpaket*/
└ *Klassenname*.`java` *und / oder* *Klassenname*.`class`

Java Pakete: Syntax (2)

Voll qualifizierte Klassennamen können sehr lang und damit unhandlich werden.

- **import-Deklarationen** ermöglichen einfache Klassennamen:

```
import Paketname.Klassenname;
public final class Beispiel {
    ...
    Klassenname.Klassenmethode(); // statt Paketname.Klassenname.Klassenmethode()
    ...
}
```

*Jede Quelldatei enthält implizit **import**-Anweisungen für alle Klassen aus java.lang (dadurch System statt java.lang.System, String statt java.lang.String usw.).*

- mit **import-static-Deklaration** können Klassennamen ganz entfallen:

```
import static Paketname.Klassenname.Klassenmethode;
public final class Beispiel {
    ...
    Klassenmethode(); // statt Paketname.Klassenname.Klassenmethode()
    ...
}
```

Beispiel-Programm Paket

```
package de.htwg.ain.prog1.teil4;
public final class Gruss {
    private Gruss() { }
    public static void hallo() {
        System.out.println("Hallo");
    }
}
```

de/htwg/ain/prog1/teil4/Gruss.java: Klasse liegt in benanntem Paket

Klassenmethode hallo

```
import de.htwg.ain.prog1.teil4.Gruss;
import static de.htwg.ain.prog1.teil4.Gruss.hallo;
public final class GrussTest {
    private GrussTest() { }
    public static void main(String[] args) {
        de.htwg.ain.prog1.teil4.Gruss.hallo();
        Gruss.hallo();
        hallo();
    }
}
```

GrussTest.java: Klasse liegt in unbenanntem Standardpaket

ohne **`import`** *voll qualifizierter Klassenname erforderlich*

mit **`import`** *reicht der einfache Klassenname*

mit **`import static`** *reicht der Methodenname*

Java Klassen: Syntax

- **Klassendefinition**:

```
Zugriffsrecht final class Klassenname {
    ...
}
```

Zugriffsrecht	**`public`**	die Klasse ist für alle anderen Klassen sichtbar
		ohne Zugriffsrecht ist die Klasse nur für Klassen im gleichen Paket sichtbar
`final`		kennzeichnet die Klasse als nicht erweiterbar
		erweiterbare Klassen sind Thema der Objektorientierung in Teil 5
Klassenname		muss im Paket eindeutig sein
		Konvention: sollte mit einem Großbuchstaben beginnen

- der Name einer Klasse legt die Dateinamen für deren Code fest

Klassenname`.java`	Quellcode der Klasse
Klassenname`.class`	mit **`javac`** aus dem Quellcode erstellter Bytecode der Klasse

Der Quellcode von Klassen ohne Zugriffsrecht darf auch in einer Datei zusammengefasst werden.
Die erste Klasse (die als einzige auch public sein darf) bestimmt dann den Dateinamen.
Der Bytecode wird von javac trotzdem klassenweise getrennt abgelegt.

Java Klassen: Verwendung

Klassen können in Java verwendet werden,

- um Startpunkte von Programmen zu definieren → **Main-Klassen**

  Main-Klassen haben als einzige öffentliche Methode die Klassenmethode main

  alle Beispielklassen aus Teil 2 und Teil 3 fallen in diese Kategorie

- um Module zu definieren → **Utility-Klassen** *(Dienstklassen)*

  Utility-Klassen enthalten nur Klassenmethoden und (konstante) Klassenvariablen

  Beispiele aus der Bibliothek sind java.lang.System, java.lang.Math, java.util.Arrays

- um Baupläne für Objekte zu definieren → **instanziierbare Klassen**

  instanziierbare Klassen legen mit Instanzvariablen den Aufbau von Objekten fest, mit Konstruktoren deren Initialisierung und mit Instanzmethoden deren Verhalten

  Beispiele aus der Bibliothek sind java.lang.String, java.util.Scanner

Programmiertechnik 1 - Teil 4

1) Pakete und Klassen
2) **Klassenmethoden und Klassenvariablen**
3) Blick in die Java Klassenbibliothek: Utility-Klassen
4) Instanziierbare Klassen
5) Blick in die Java Klassenbibliothek: Klassen für Wertobjekte und Entitäten
6) Eingebettete Klassen
7) Empfehlungen

Java Klassen: Klassenmethoden

Klassenmethoden *(Funktionen, Unterprogramme, Prozeduren)*
fassen Folgen von Anweisungen zusammen, die immer wieder gebraucht werden.

- Eine Klassenmethode hat eine Signatur

  Die **Signatur** besteht aus dem *Namen* der Methode und der Liste ihrer *Parametertypen*. Methodensignaturen müssen innerhalb einer Klasse eindeutig sein.

  Die **Parameter** einer Methode sind Variablen, die beim Methodenaufruf mit *Argumenten* initialisiert werden müssen. Sie dienen der Übergabe zu verarbeitender Daten.

- Eine Klassenmethode hat einen Rückgabetyp

  Der Rückgabetyp legt fest, welche Werte ein Methodenaufruf als Ergebnis liefern kann.

- Eine Klassenmethode hat einen Rumpf

  Der Rumpf enthält die auszuführenden Anweisungen.

- Eine Klassenmethode kann Ausnahmen werfen

  Für sogenannte geprüfte Ausnahmen *(Checked Exceptions)* muss das deklariert werden.

Klassenmethoden: Syntax (1)

- **Methodendefinition**:

Konvention: Name sollte mit Kleinbuchstabe beginnen

```
public final class Klassenname {
    Zugriffsrecht static Rückgabetyp Methodenname(Parameterliste)
        throws Ausnahmeliste {
        ... // Anweisungen
        return Rückgabewert; // return; bei Rückgabetyp void
    }
    ...
}
```

Kopf / *Rumpf*

Zugriffsrecht: **`private`** Methode ist nur innerhalb ihrer Klasse aufrufbar
`public` Methode ist von überall aufrufbar
ohne Zugriffsrecht ist die Methode nur für alle Klassen im gleichen Paket sichtbar

`static`: kennzeichnet die Methode als Klassenmethode

Rückgabetyp: Rückgabewert der **`return`**-Anweisung muss zum Rückgabetyp passen
*Eine **`return`**-Anweisung ohne Rückgabewert darf am Rumpfende entfallen.*
*Bei Verzweigungen im Rumpf kann es auch mehrere **`return`**-Anweisung geben.*

Parameterliste: durch Komma getrennte Variablendeklarationen oder leer **`()`**

Ausnahmeliste: durch Komma getrennte Ausnahmen *(throws-Deklaration kann oft entfallen)*

Klassenmethoden: Syntax (2)

- **Methodenaufruf**:

  durch Komma getrennte Ausdrücke initialisieren die Parameter der Methode

  Klassenname **.** *Methodenname* **(** *Argumentliste* **)**

  Der Aufrufoperator **`()`** veranlasst die Ausführung der Klassenmethode und liefert als Ergebnis deren Rückgabewert.

  Die Ausdrücke der Argumentliste müssen zu den Typen der Parameter passen.

  Bei Methoden mit **`throws`***-Deklaration Aufruf eventuell nur in einem* **`try`***-Block möglich.*

- Kurzschreibweise ohne Klassenname bei Methoden der gleichen Klasse:

  Methodenname **(** *Argumentliste* **)**

  auch für Methoden einer anderen Klasse, sofern diese einem Paket zugeordnet ist und der Methodenname im Kopf der Quelldatei importiert wird:

  ```
  import static Paketname.Klassenname.Methodenname;
  ```

- bei Klassenmethoden ohne Parameter entfällt die Argumentliste:

  Klassenname **.** *Methodenname* **()**

Klassenmethoden: main

- die Ausführung eines Programms beginnt in der Klassenmethode **main (String [ ])** :

```
public final class Programmname {
    private Programmname() { }

    public static void main(String[] args) {
        ...
    }
}
```

Stilempfehlung: privater Konstruktor

Zugriffsrecht immer public

Rückgabetyp immer void

Parameterliste immer String [] args

- Beispielaufruf:

```
java Programmname mit drei Argumenten
```

args [0] *args [1]* *args [2]*

Die virtuelle Maschine **`java`** *liest den Bytecode aus Programmname*`.class` *ein, instanziiert ein String-Feld mit den angegebenen Kommandozeilen-Argumenten und ruft dann die Klassenmethode* **`main`** *mit dem String-Feld als Argument auf.*

Beispielprogramm Klassenmethode (Aufruf aus gleicher Klasse)

```
public final class Maximum {
    private Maximum() { }

    private static int max(int a, int b) {
        return a > b ? a : b;
    }

    public static void main(String[] args) {
        if (args.length == 0) {
            System.err.println("Aufruf: java Maximum Zahl ...");
            System.exit(1); // Programmabbruch mit Fehler
        }
        int m = Integer.parseInt(args[0]);
        for (int i = 1; i < args.length; ++i) {
            int n = Integer.parseInt(args[i]);
            m = max(m, n);
        }
        System.out.println(m);
    }
}
```

Gibt das Maximum seiner Argumente aus

Beispielaufruf:

```
java Maximum 1 2 3
```

Ausgabe:

```
3
```

Parameter a wird mit Wert von m, Parameter b mit Wert von n initialisiert

Beispielprogramm Klassenmethode (Aufruf aus anderer Klasse)

```
public final class Maximum {  // Main-Klasse
    private Maximum() { }
    public static void main(String[] args) {
        ... // siehe vorige Folie
            m = IntegerMethods.max(m, n);
        ... // siehe vorige Folie
    }
}

public final class IntegerMethods {  // Utility-Klasse
    private IntegerMethods() { }
    public static int max(int m, int n) {
        return m > n ? m : n;
    }
    public static int min(int m, int n) {
        return m < n ? m : n;
    }
}
```

beim Aufruf ist der Klassenname erforderlich

Methode ist public, damit sie aus anderer Klasse aufrufbar ist

Klassenmethoden: variable Anzahl von Argumenten

Der letzte Parameter einer Methode kann beliebig viele Argumente aufnehmen, wenn nach dem Typ drei Punkte folgen: *Typ... Name*

- Methodendefinition:

```
public static int max(int a, int... b) {
    int m = a;
    for (int n: b) {
        if (m < n) {
            m = n;
        }
    }
    return m;
}
```

den Parameter ***int... b***
realisiert der Compiler als ***int[] b***

- Methodenaufruf:

```
max(10);          // vom Compiler realisiert als  max(10, new int[0])
max(11, 12);      // vom Compiler realisiert als  max(11, new int[] {12})
max(13, 14, 15);  // vom Compiler realisiert als  max(13, new int[] {14, 15})
```

Klassenmethoden: Overloading

- Überladen von Methoden (**Method Overloading**):

  Mehrere Methoden dürfen denselben Namen haben, wenn sich ihre Parameterliste unterscheidet.

Achtung:
gleiche Methodensignaturen mit lediglich verschiedenem Rückgabetyp sind nicht erlaubt!

- Definition überladener Methoden:

```
public static int max(int a, int b) { ... }
public static int max(int a, int b, int c) { ... }
public static double max(double a, double b) { ... }
```

- Aufruf überladener Methoden

  Der Compiler sucht die Methode aus, deren Signatur am besten zu den Argumenten passt:

```
max(1, 2);      // Signatur max(int, int) passt am besten
max(1, 2, 3);   // Signatur max(int, int, int) passt am besten
max(1, 2.3) ;   // Signatur max(double, double) passt am besten
```

Passen mehrere Methoden gleich gut, meldet der Compiler einen Fehler!

Java Klassen: Variablen

- **Lokale Variablen** werden innerhalb eines Anweisungsblocks { } definiert

  Lebensdauer: Ausführung des Anweisungsblock (z.B. Methodenrumpf, Schleifenrumpf)
  Speicherort: Stack
  Anfangswert: undefiniert, wenn ohne explizite Initialisierung definiert

- **Parameter** werden im Kopf einer Methode bzw. eines catch-Blocks definiert

  Lebensdauer: Ausführung des Methodenrumpfs bzw. des catch-Blocks
  Speicherort: Stack
  Anfangswert: zugehöriges Argument des aktuellen Methodenaufrufs bzw. die gefangene Ausnahme

- **Klassenvariablen** werden auf Klassenebene außerhalb der Methoden definiert

  Lebensdauer: solange wie die Klasse in der virtuellen Maschine geladen ist
  Speicherort: Heap
  Anfangswert: Standardwert des Typs, wenn explizite Initialisierung fehlt

- **Instanzvariablen** werden auf Klassenebene außerhalb der Methoden definiert

  Lebensdauer: existieren pro erzeugtem Objekt, solange wie das jeweilige Objekt existiert
  Speicherort: Heap
  Anfangswert: Standardwert des Typs, wenn explizite Initialisierung fehlt

Java Klassenvariablen: Syntax

- **Variablendefinition**:

```
public final class Klassenname {
    ...
    Zugriffsrecht static Typ Name = Wert;
    Zugriffsrecht static Typ AndererName;
    ...
}
```

Konvention:
Name sollte mit Kleinbuchstabe beginnen bzw. bei ***`static final`*** *nur aus Großbuchstaben bestehen*

ist kein Wert angegeben, dann wird automatisch mit dem Standardwert des Typs (bei Zahlen 0) initialisiert (nicht bei ***`static final`****)*

Zugriffsrecht: **`private`** Variable ist nur für Methoden ihrer Klasse zugreifbar
`public` Variable ist von überall zugreifbar
ohne Zugriffsrecht ist die Variable nur für die Klassen im gleichen Paket sichtbar

`static`: kennzeichnet die Variable als Klassenvariable

- **Variablenzugriff**:

Klassenname`.`*Name*

Bei Zugriff innerhalb einer Klasse darf der Klassenname weggelassen werden:

Name *kein Unterschied zum Zugriff auf eine lokale Variable*

Beispiel-Programm Klassenvariable

```
public final class ClassVar {
    private ClassVar() { }

    private static int global = 1;   // Klassenvariable

    private static int aClassMethod(int param) {
        int local = param + 1;
        global = param + 2;
        return local;
    }

    public static void main(String[] args) {
        int local = 1;
        local = aClassMethod(local);                    // local wird 2
        global = aClassMethod(local);                   // global wird 3
        local = ClassVar.aClassMethod(ClassVar.global); // local wird 4
    }
}
```

der Klassenname darf hier auch weggelassen werden

Java Klassenvariablen: statische Initialisierungsblöcke

Ein **statischer Initialisierungsblock** erlaubt Initialisierungen für Klassenvariablen, die nicht mit Initialisierungsausdrücken in Variablendefinitionen ausdrückbar sind.

- Syntax:

```
public final class Klassenname {
    ...
    static {
        ... // Anweisungen
    }
    ...
}
```

Ein statischer Initialisierungsblock ist eine Klassenmethode ohne Zugriffsrecht, ohne Rückgabetyp, ohne Name und ohne Parameter

- eine Klasse kann auch mehrere statische Initialisierungsblöcke haben

  Ausführungsreihenfolge ist von oben nach unten

  Ausführungszeitpunkt ist das Laden der Klasse in die virtuelle Maschine (also vor allen benannten Klassenmethoden, insbesondere auch vor main)

Beispiel-Programm statischer Initialisierungsblock

Erzeugt zufällig 1 bis 10 Zufallszahlen und gibt sie aus

```
public final class Zufall {
    private Zufall() { }

    private static final int[] ZUFALLSZAHLEN;

    static {
        java.util.Random r = new java.util.Random();
        ZUFALLSZAHLEN = new int[r.nextInt(10) + 1]; // maximal 10 Zahlen
        for (int i = 0; i < ZUFALLSZAHLEN.length; ++i) {
            ZUFALLSZAHLEN[i] = r.nextInt();
        }
    }

    public static void main(String[] args) {
        for (int n: ZUFALLSZAHLEN) {
            System.out.println(n);
        }
    }
}
```

Programmiertechnik 1 - Teil 4

1) Pakete und Klassen
2) Klassenmethoden und Klassenvariablen
3) **Blick in die Java Klassenbibliothek: Utility-Klassen**
4) Instanziierbare Klassen
5) Blick in die Java Klassenbibliothek: Klassen für Wertobjekte und Entitäten
6) Eingebettete Klassen
7) Empfehlungen

Java Klassenbibliothek: Utility-Klasse java.lang.System

Die Pakete der Java-Klassenbibliothek sind in verschiedenen Modulen zusammengefasst.
Die Pakete aller hier und im Folgenden besprochenen Klassen liegen im Modul java.base.

java.lang.System ermöglicht den Zugriff auf die Laufzeitumgebung
(z.B. Standard-Ein-/Ausgabe-Kanäle, Systemuhr, ...)

- Klassenvariablen:

```
public static final java.io.InputStream in
public static final java.io.PrintStream out
public static final java.io.PrintStream err
```

- Klassenmethoden:

```
public static void exit(int status)
```

beendet das aktuell laufende Programm (status ungleich 0 bedeutet Fehlerabbruch)

```
public static long currentTimeMillis()
```

liefert die Anzahl Millisekunden seit dem 1. Januar 1970, 0 Uhr UTC

```
... // noch viele weitere Klassenmethoden
```

Java Klassenbibliothek: Utility-Klasse java.lang.Math

java.lang.Math enthält mathematische Konstanten und Operationen

- Klassenvariablen:

```
public static final double E     Basis des natürlichen Logarithmus e
public static final double PI    die Kreiskonstante π
```

- Klassenmethoden:

```
public static double sin(double a)
```

liefert den Sinus von a (a in Radiant, nicht Grad)

```
public static double sqrt(double a)
```

liefert die positive Quadratwurzel von a

```
...  // noch viele weitere Klassenmethoden
```

Java Klassenbibliothek: Utility-Klasse java.util.Arrays

java.util.Arrays erlaubt grundlegende Feldoperationen

- Klassenmethoden:

```
public static boolean equals(int[] a1, int[] a2)
```
vergleicht zwei Felder ganzer Zahlen

```
public static void fill(int[] a, int val)
```
setzt alle Elemente eines Felds ganzer Zahlen auf den Wert val

```
public static void sort(int[] a)
```
sortiert ein Feld ganzer Zahlen aufsteigend

```
public static String toString(int[] a)
```
liefert String-Darstellung eines Felds ganzer Zahlen

```
...  // das gleiche für Felder aller Grundtypen
...  // außerdem viele weitere Klassenmethoden
```

Programmiertechnik 1 - Teil 4

1) Pakete und Klassen
2) Klassenmethoden und Klassenvariablen
3) Blick in die Java Klassenbibliothek: Utility-Klassen
4) **Instanziierbare Klassen**
5) Blick in die Java Klassenbibliothek: Klassen für Wertobjekte und Entitäten
6) Eingebettete Klassen
7) Empfehlungen

Java Klassen: Objekte

Instanziierbare Klassen definieren Aufbau, Initialisierung, Verhalten von **Objekten** *(umgekehrt gesagt: Objekte sind Instanzen einer Klasse)*:

- Objekterzeugung mit einem **`new`-Ausdruck**:

```
Klassenname einObjekt = new Klassenname(Argumentliste)
```

Der new-Ausdruck reserviert auf dem Heap Speicher für das neue Objekt und initialisiert den Speicher über einen Konstruktoraufruf mit der angegebenen Argumentliste.

Der Platzbedarf des Objekts hängt von Anzahl und Typ der in der Klasse definierten Instanzvariablen ab.

*Wird das Objekt von keiner Variablen mehr referenziert, gibt der **Garbage-Collector** der virtuellen Maschine **`java`** den Speicher des Objekts wieder frei.*

- Objektbenutzung über Referenzvariablen und den **Punkt-Operator**:

```
einObjekt.Variablenname = Wert;
einObjekt.Methodenname(...);
```

Java Objekte: Verwendung

Objekte können in Java verwendet werden,

- um Werte zu repräsentieren → **value objects**

  pro Objekt ein unveränderlicher Wert *(immutable object)*

  der Wert setzt sich aus den Werten der Instanzvariablen zusammen
  und die Instanzvariablen sind `final`

  Objekte gelten als gleich, wenn sie den gleichen Wert speichern

  Beispiele aus der Bibliothek sind java.lang.String, java.time.LocalDate

- um Entitäten zu repräsentieren → **entity objects**

  die Instanzvariablen dürfen veränderlich sein *(mutable object)*

  jedes Objekt hat eine Identität, die es von jedem anderen Objekt unterscheidet,
  selbst wenn die Werte der Instanzvariablen gleich sind

  Beispiele aus der Bibliothek sind java.util.Scanner, java.lang.StringBuilder

Java Klassenbibliothek: Klasse java.lang.Object (1)

java.lang.Object definiert die Gemeinsamkeiten aller Java-Objekte
Jede Klasse ist eine direkte oder indirekte Erweiterung von java.lang.Object

- die wichtigsten Instanzmethoden:

```
public final Class<? extends Object> getClass()
```

bei jedem Objekt ist abfragbar, zu welcher Klasse es gehört

```
public String toString()
```

jedes Objekt hat eine Darstellung als String

```
public boolean equals(Object that)
```

jedes Objekt kann auf Gleichheit mit einem anderen Objekt geprüft werden

```
public int hashCode()
```

jedem Objekt kann eine ganze Zahl zugeordnet werden, wobei Objekte, die laut equals als gleich gelten, dieselbe Zuordnung haben müssen

```
...  // weitere Methoden für Objektkopie und Thread-Synchronisation
```

Java Klassenbibliothek: Klasse java.lang.Object (2)

- java.lang.Object enthält Standardimplementierungen der Instanzmethoden, von denen aber einige nur für Entitäten passen und nicht für Wertobjekte:

```
public String toString() {
    return getClass().getName() + '@' + Integer.toHexString(hashCode());
}
```

bei value objects muss der String aus dem gespeicherten Wert abgeleitet werden

```
public boolean equals(Object obj) {
    return this == obj; // Objektidentität (gleiche Referenz)
}
```

bei value objects müssen die gespeicherten Werte der Objekte verglichen werden

```
public int hashCode() {
    ... // leitet den Hashcode aus der Speicheradresse des Objekts ab
}
```

bei value objects muss der Hashcode konsistent zu equals aus dem gespeicherten Wert abgeleitet werden

Java Instanzvariablen: Eigenschaften und Syntax

Instanzvariablen werden pro Objekt einer Klasse angelegt

- Variablendefinition wie Klassenvariablen, nur ohne den Zusatz **`static`**:

```
public final class Klassenname {
    ...
    Zugriffsrecht  Typ  Name  =  Wert;
    Zugriffsrecht  Typ  AndererName;
    ...
}
```

Konvention: Name sollte mit Kleinbuchstabe beginnen

*ist kein Wert angegeben, dann wird automatisch mit dem Standardwert des Typs (bei Zahlen 0) initialisiert (nicht bei **`final`**)*

*hier kein **`static`***

Zugriffsrecht: **`private`** Variable ist nur für Methoden ihrer Klasse zugreifbar

`public` Variable ist von überall zugreifbar

ohne Zugriffsrecht ist die Variable nur für Klassen im gleichen Paket sichtbar

- Variablenzugriff:

Objektreferenz`.`*Name*

`this``.`*Name* oder Kurzschreibweise *Name* `//` *in den Instanzmethoden der Klasse*

Java Instanzmethoden: Eigenschaften und Syntax

Instanzmethoden sind Methoden einer Klasse, die nur bei Objekten aufrufbar sind.

- Methodendefinition wie Klassenmethoden, nur ohne den Zusatz **`static`**:

```
public final class Klassenname {
    ...
    Zugriffsrecht Rückgabetyp Name (Parameterliste) {   // Kopf
        ...                                             // Rumpf
    }
    ...
}
```

hier kein **`static`**

`javac` fügt am Anfang der Parameterliste zusätzlich einen Parameter **`this`** ein:

```
final Klassenname this      // Referenz auf das Objekt des Methodenaufrufs
```

- das Objekt des Methodenaufrufs ist das Argument für den Parameter **`this`**:

```
Objektreferenz.Name (Argumentliste)    // this = Objektreferenz
```

Java Konstruktoren: Eigenschaften und Syntax

Konstruktoren sind Instanzmethoden für die Objektinitialisierung.

- Konstruktoren haben als Name den Klassennamen.

  Eine Klasse darf mehrere Konstruktoren mit unterschiedlichen Parameterlisten haben.

- Konstruktoren haben keinen Rückgabetyp.

- Ein parameterloser Konstruktor (einer mit nur dem heimlichen Parameter **`this`**) wird als Standardkonstruktor *(Default-Konstruktor)* bezeichnet:

  Zugriffsrecht Klassenname **`() { ... }`**

  wird eine Klasse ganz ohne Konstruktoren deklariert, dann erzeugt der Compiler implizit einen Standardkonstruktor mit Zugriffsrecht ***`public`***

- Konstruktoren werden über einen **`new`**-Ausdruck aufgerufen:

  Klassenname Objektreferenz **`=`** **`new`** *Klassenname* **`();`** *`// Standardkonstruktor`*

  Klassenname Objektreferenz **`=`** **`new`** *Klassenname* **`(`***Argumentliste***`);`**

Java Konstruktoren: Implementierung

Die **Implementierung eines Konstruktors** besteht aus mehreren Teilen:

- Aufruf eines anderen Konstruktors (muss die erste Anweisung im Rumpf sein)

entweder explizit	`this(`*Argumentliste*`);`	*anderer Konstruktor der gleichen Klasse*
oder explizit	`super(`*Argumentliste*`);`	*Konstruktor der Oberklasse*
ansonsten implizit	`super();`	*Standardkonstruktor der Oberklasse*

zum Konzept Oberklasse siehe das Kapitel Objektorientierung

- Initialisierer der Instanzvariablen und Initialisierungblöcke der Klasse

```
instanzvariable = wert;

{
   ... // Anweisungen
}
```

die Initialisierer aus den Variablendefinitionen und die nicht-statischen Initialisierungsblöcke der Klasse werden in alle Konstruktoren übernommen, die nicht mit einem `this(...)`-Aufruf beginnen

nicht konstante Instanzvariablen ohne Initialisierung erhalten den Standardwert Ihres Typs

- sonstige explizite Anweisungen im Rumpf

Beispiel-Programm Wertobjekt (1)

- **Datumswerte** lassen sich mit ganzzahligen Instanzvariablen für Tag, Monat und Jahr repräsentieren:

```
public final class Datum {
    public final int tag;
    public final int monat;
    public final int jahr;

    private Datum(/* final Datum this, */
                  int tag, int monat, int jahr) {
        this.tag = tag;
        this.monat = monat;
        this.jahr = jahr;
    }

    ... // Fabrikmethoden (siehe 4-32)

    ... // Reimplementierung von java.lang.Object-Methoden (siehe 4-33)
}
```

*alle Instanzvariablen **final**, denn Wertobjekte müssen unveränderlich sein*

javac** ergänzt einen Parameter **this

*Konstruktor ist **private**, damit neue Instanzen nur innerhalb der Klasse erzeugt werden können (von den Fabrikmethoden)*

Beispiel-Programm Wertobjekt (2)

- Fabrikmethoden sind Klassenmethoden, die eine Instanz ihrer Klasse liefern *(bei Wertobjekten muss das nicht zwingend eine neu erzeugte Instanz sein)*:

```
public static Datum valueOf(int tag, int monat, int jahr) {
    // uebergebenes Datum pruefen (stark vereinfacht)
    if (tag < 1 || tag > 31 || monat < 1 || monat > 12) {
        throw new IllegalArgumentException("ungueltiges Datum");
    }
    // value object erzeugen
    return new Datum(tag, monat, jahr);
}

public static Datum heute() {
    // Systemkalender ablesen
    java.util.Calendar c = java.util.Calendar.getInstance();
    // value object erzeugen
    return new Datum(c.get(java.util.Calendar.DAY_OF_MONTH),
                     c.get(java.util.Calendar.MONTH) + 1,
                     c.get(java.util.Calendar.YEAR));
}
```

Beispiel-Programm Wertobjekt (3)

- java.lang.Object-Methoden müssen die Instanzvariablen verwenden:

```
@Override public String toString(/* final Datum this */) {
    return String.format("%04d-%02d-%02d", this.jahr, this.monat, this.tag);
}

@Override public boolean equals(/* final Datum this, */ Object o) {
    if (o instanceof Datum) {
        Datum that = (Datum) o;
        return this.tag == that.tag
            && this.monat == that.monat
            && this.jahr == that.jahr;
    }
    return false;
}

@Override public int hashCode(/* final Datum this */) {
    return (this.jahr << 9) + (this.monat << 5) + this.tag;
}
```

Beispiel-Programm Wertobjekt (4)

- Benutzung von Datumswerten in einer Main-Klasse:

```
import java.util.Scanner;
public final class DatumTest {
    private DatumTest() { }
    private static final Scanner IN = new Scanner(System.in);
    public static void main(String[] args) {
        System.out.println("Welches Datum ist heute?");

        Datum d = Datum.valueOf(IN.nextInt(), IN.nextInt(), IN.nextInt());
        Datum heute = Datum.heute();

        if (d.equals(heute)) {
            System.out.printf("Richtig, %s ist das heutige Datum!", d.toString());
        } else {
            System.out.printf("Falsch, %s ist das heutige Datum, nicht %s!", heute, d);
        }
    }
}
```

unbedingt equals statt == nutzen!

printf ruft automatisch toString auf

Beispiel-Programm Entität (1)

- **Termine** lassen sich als Entitäten mit Datum und Beschreibung repräsentieren:

```
import java.util.Objects;

public final class Termin {
    private Datum wann;
    private final String was;

    public Termin(/* final Termin this, */ Datum wann, String was) {
        this.wann = Objects.requireNonNull(wann, "ungueltiger Termin");
        this.was = Objects.requireNonNull(was, "ungueltiger Termin");
    }

    public void verschieben(/* final Termin this, */ Datum wann) {
        this.wann = Objects.requireNonNull(wann, "ungueltiger Termin");
    }

    public Datum wann(/* final Termin this */) { return this.wann; }

    public String was(/* final Termin this */) { return this.was; }
}
```

Termine sollen verschiebbar sein, deshalb Datum ohne final

auch wenn ein Termin verschoben wird, ist es noch immer derselbe Termin (es wird kein neues Termin-Objekt erzeugt)

Beispiel-Programm Entität (2)

- Benutzung in einer Main-Klasse:

weil beide Terminkalender auf dasselbe Terminobjekt verweisen, ist die Verschiebung konsistent in beiden Kalendern sichtbar

```
public final class TerminTest {
    private TerminTest() { }

    public static void main(String[] args) {
        Collection<Termin> prueferKalender = new LinkedList<Termin>();
        Collection<Termin> kandidatenKalender = new LinkedList<Termin>();

        Termin pruefung = new Termin(Datum.heute(), "Pruefung Programmiertechnik 1");
        prueferKalender.add(pruefung);
        kandidatenKalender.add(pruefung);

        pruefung.verschieben(Datum.valueOf(11, 11, 2111));

        for (Termin t: prueferKalender) {
            System.out.printf("Pruefer: %s, %s%n", t.wann(), t.was());
        }

        for (Termin t: kandidatenKalender) {
            System.out.printf("Kandidat: %s, %s%n", t.wann(), t.was());
        }
    }
}
```

Programmiertechnik 1 - Teil 4

1) Pakete und Klassen
2) Klassenmethoden und Klassenvariablen
3) Blick in die Java Klassenbibliothek: Utility-Klassen
4) Instanziierbare Klassen
5) **Blick in die Java Klassenbibliothek: Klassen für Wertobjekte und Entitäten**
6) Eingebettete Klassen
7) Empfehlungen

Java Bibliothek: Wrapper-Klassen (1)

Wrapper-Klassen verpacken Werte der primitiven Datentypen in value objects

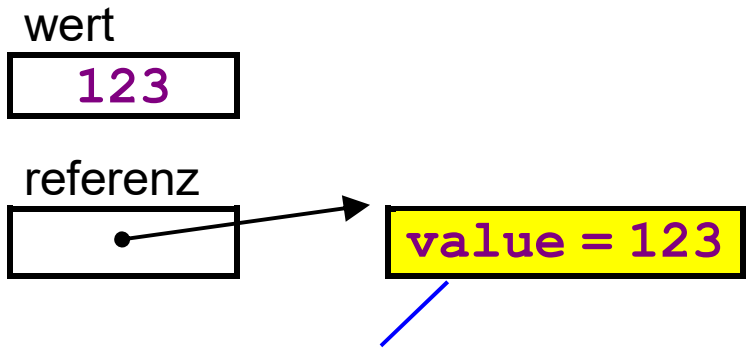
- pro primitivem Datentyp gibt es im Paket java.lang eine Wrapper-Klasse:

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

- Wrapper-Klassen ermöglichen Referenzen auf Werte der primitiven Datentypen:

```
int wert = 123;

Integer referenz = Integer.valueOf(123);
```



*Das Verpacken eines Werts in ein Objekt wird **Boxing** genannt, bzw. **Autoboxing**, wenn der Compiler das Wrapper-Objekt unter der Hand erzeugt.*

Der Wert im Wrapper-Objekt kann nach der Initialisierung nicht mehr geändert werden

Java Bibliothek: Wrapper-Klassen (2)

- Wrapper-Klassen sind nützlich für Typwandlungen, z.B.:

```
String zahlAlsString = "123";
Integer zahlAlsObjekt = Integer.valueOf(zahlAlsString);
int zahl = zahlAlsObjekt.intValue();
```

oder kürzer und effizienter per Klassenmethode ohne Objekterzeugung:

```
int zahl = Integer.parseInt(zahlAlsString);
```

Konstruktor und **`parseInt`** *werfen eine* **`NumberFormatException`**,
wenn der String keine ganze Zahl enthält.

- Wrapper-Klassen liefern Informationen über die zugehörigen primitiven Typen, im Fall von Integer über **`int`**:

```
public static final int MIN_VALUE      -2^31
public static final int MAX_VALUE      2^31-1
public static final int SIZE           Anzahl Bits: 32
```

Java Bibliothek: Klasse java.lang.Boolean 

Aufbau der Wrapper-Klasse **java.lang.Boolean**:

- Klassenvariablen:

```
public static final Boolean TRUE
public static final Boolean FALSE
...
```

- Fabrikmethoden:

Boxing

```
public static Boolean valueOf(boolean b)
public static Boolean valueOf(String s)
```

- weitere Klassenmethoden:

```
public static boolean parseBoolean(String s)
public static String toString(boolean b)
...
```

- Instanzvariable:

```
private final boolean value
```

- Konstruktoren: *aus historischen Gründen public, seit Java 9 Deprecated*

```
public Boolean(boolean b)
public Boolean(String s)
```

- Instanzmethoden:

Unboxing

```
public boolean booleanValue()
public String toString()
public boolean equals(Boolean b)
public int compareTo(Boolean b)
...
```

Beispiel-Programm java.lang.Boolean

```
public final class BooleanWrapper {
    private BooleanWrapper() { }
    public static void main(String[] args) {
        Boolean reference;
        reference = Boolean.TRUE;
        reference = true; // Autoboxing mit Boolean.valueOf(true)
        reference = Boolean.valueOf("true");

        boolean value;
        value = reference; // Autounboxing mit reference.booleanValue()
        value = Boolean.parseBoolean("true");
        value = reference.equals(Boolean.TRUE);
        value = reference.compareTo(Boolean.TRUE) == 0;

        String s;
        s = Boolean.toString(true);
        s = reference.toString();
    }
}
```

Wrapper-Objekt Boolean.TRUE

Wert true

String "true"

Java Bibliothek: Klasse java.lang.StringBuilder

java.lang.StringBuilder-Entitäten erlauben das schrittweise Bauen von Strings:

- Konstruktoren:

```
public StringBuilder()
public StringBuilder(String s)
...
```

- Instanzmethoden:

```
public StringBuilder append(Typ s)
public StringBuilder insert(int offset, Typ s)
...
```

*die **append**- und **insert**-Methoden gibt es unter anderem für alle Java-Grundtypen und für String*

Bis Java 8 hat der Compiler die **StringBuilder**-Klasse verwendet, um die Konkatenationen von konstanten Zeichenketten **s + t** zu realisieren:

```
new StringBuilder().append(s).append(t).toString()
```

neues StringBuilder-Objekt mit leerem String — *String s anhängen* — *String t anhängen* — *neues String-Objekt mit verkettetem String*

Beispiel-Programm java.lang.StringBuilder

```
public final class StringBuilderVar {
    private StringBuilderVar() { }

    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < args.length; ++i) {
            if (i > 0) {
                sb.append(',');
            }
            sb.append(i).append(":\"").append(args[i]).append('\"');
        }
        System.out.println(sb.toString());
    }
}
```

Welche verschiedenen StringBuilder-Methoden ruft das Programm auf?

Was gibt der folgende Beispielaufruf auf der Konsole aus?

```
java StringBuilderVar mit einigen Argumenten
```

Programmiertechnik 1 - Teil 4

1) Pakete und Klassen
2) Klassenmethoden und Klassenvariablen
3) Blick in die Java Klassenbibliothek: Utility-Klassen
4) Instanziierbare Klassen
5) Blick in die Java Klassenbibliothek: Klassen für Wertobjekte und Entitäten
6) **Eingebettete Klassen**
7) Empfehlungen

Java eingebettete Klassen: Übersicht

Hilfsklassen, die nur gemeinsam mit einer anderen Klasse gebraucht werden, können in diese Klasse eingebettet werden:

```
public class OuterClass {
    ...
    public static class NestedClass {
        ...
    }
    public class InnerClass {
        ...
    }
    ...
}
```

alle drei Klassen haben wechselseitig vollen Zugriff auch auf die privaten Teile

- Beispiel:

  Hilfsklassen, die als Rückgabetyp für Methoden der umgebenden Klasse dienen

Java eingebettete Klassen: statische Einbettung

- Klassen-Definition:

```
public final class OuterClass {
    ...
    Zugriffsrecht static final class NestedClass {
        ...
    }
    ...
}
```

eine statisch eingebettete Klasse darf alles enthalten, was bei Klassen erlaubt ist, aber üblicherweise Verzicht auf öffentliche Konstruktoren

- Klassen-Benutzung:

  außerhalb von OuterClass muss die eingebettete Klasse
  mit dem qualifizierten Namen OuterClass.NestedClass angesprochen werden

  ansonsten in der Benutzung keine Unterschiede zu gewöhnlichen Klassen

Java eingebettete Klassen: innere Klassen

- Klassen-Definition:

```
public final class OuterClass {
    ...
    Zugriffsrecht final class InnerClass {
        ...
    }
    ...
}
```

hier kein **`static`**

eine innere Klasse darf keine ***`static`*** *markierten Elemente (außer Konstanten) enthalten*

jedes InnerClass-Objekt enthält implizit eine private Instanzvariable OuterClass**`.this`**, deren Wert eine Referenz auf das erzeugende OuterClass-Objekt ist

- Klassen-Benutzung:

InnerClass-Objekte können nur mit Hilfe eines OuterClass-Objekts erzeugt werden

```
OuterClass outerObject = ... ;
OuterClass.InnerClass innerObject = outerObject.new InnerClass(...);
```

innerhalb einer Instanzmethode von OuterClass auch ***`this.new`*** *oder kurz* ***`new`***

Beispiel-Programm eingebettete Klassen (1)

IntList verwaltet ganze Zahlen in einer einfach verketteten Liste

```
public final class IntList {
    private Element head = null; // leere Liste

    public IntList insert(/* final IntList this, */ int n) {
        this.head = new Element(this.head, n);
        return this;
    }

    private static final class Element {
        private final Element next; // Verkettung
        private final int n;        // Wert des Listenelements

        private Element(/* final Element this, */ Element e, int n) {
            this.next = e;
            this.n = n;
        }
    }

    ...
```

statisch eingebettete Hilfsklasse für die Listenelemente

Beispiel-Programm eingebettete Klassen (2)

```
    ...
    public final class Iterator {
        // private IntList IntList.this;
        private Element current = IntList.this.head;

        public boolean hasNext(/* final Iterator this */) {
            return this.current != null;
        }

        public int next(/* final Iterator this */) {
            if (this.current == null) {
                throw new java.util.NoSuchElementException();
            }
            Element e = this.current;
            this.current = this.current.next;
            return e.n;
        }
    }
}
```

innere Klasse für die Iteration über die Listenelemente

Beispiel-Programm eingebettete Klassen (3)

ListVar erzeugt eine Liste ganzer Zahlen und iteriert darüber

```
public final class ListVar {
    private ListVar() { }

    public static void main(String[] args) {
        int[] anIntArray = {3421, 3442, 3635, 3814};

        // Liste anlegen
        IntList anIntList = new IntList();
        for (int i = anIntArray.length; i > 0; --i) {
            anIntList.insert(anIntArray[i - 1]);
        }

        // Liste ausgeben
        IntList.Iterator i = anIntList.new Iterator();
        while (i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

Programmiertechnik 1 - Teil 4

1) Pakete und Klassen
2) Klassenmethoden und Klassenvariablen
3) Blick in die Java Klassenbibliothek: Utility-Klassen
4) Instanziierbare Klassen
5) Blick in die Java Klassenbibliothek: Klassen für Wertobjekte und Entitäten
6) Eingebettete Klassen
7) **Empfehlungen**

Java Klassen: Empfehlungen (1)

- Klassen und Pakete verwenden, um Programme zu modularisieren

  dem guten Beispiel der Java Klassenbibliothek folgen und Klassen im unbenannten Standardpaket vermeiden

- Java Klassenbibliothek gegenüber Eigenimplementierungen bevorzugen

  z.B. java.util.Arrays.sort *verwenden, statt selbst ein Sortierverfahren zu programmieren*

- Main-Klassen, Utility-Klassen und instanziierbare Klassen auseinanderhalten und bei instanziierbaren Klassen solche für Wertobjekte und für Entitäten

  ältere Teile der Java Klassenbibliothek halten sich leider nicht immer daran

- Klassen, Klassenvariablen und Instanzvariablen bevorzugt **`final`** deklarieren

  final reduziert Fehlerquellen und erleichtert Optimierungen

Java Klassen: Empfehlungen (2)

- Methoden, Variablen und Hilfsklassen wenn immer möglich **`private`** deklarieren

  nur was privat ist, kann nachträglich leicht geändert werden, weil es außerhalb der Klasse nicht verwendet wird

  für private Instanzvariablen können konsistente Werte garantiert werden

  öffentliche Methoden und Hilfsklassen sowie (vermeiden) Variablen bilden die Schnittstelle einer Klasse

- Klassenvariablen vermeiden, es sei denn sie sind **`final`**

  Klassenvariablen machen Programme tendenziell unflexibel und fehleranfällig

- Scope von Namen so klein wie möglich halten 

  Variablen immer so lokal und so spät wie möglich definieren und gleich sinnvoll initialisieren, z.B. Laufvariablen erst im Schleifenkopf

  z.B. bei Methodenaufrufen Werte per Parameter übergeben statt per Klassen- oder Instanzvariablen

Java Klassen: Empfehlungen (3)

Gültigkeitsbereiche (***Scopes***) regeln, in welchem Bereich Namen von Paketen, Klassen, Variablen und die Signaturen von Methoden eindeutig sein müssen.
Bei Variablen regelt ihr Scope außerdem die Lebensdauer:

Name / Signatur	***Scope***
Paket	umschließendes Paket
Klasse	umschließendes Paket bzw. umschließende Klasse bzw. umschließende Methode
Methode	umschließende Klasse
Klassenvariable	umschließende Klasse
Instanzvariable	umschließende Klasse, Lebensdauer wie umschließendes Objekt
Parameter	Methodenrumpf bzw. `catch`-Block
lokale Variable	umschließender Anweisungsblock bzw. `for`-Schleife *

* *bei verschachtelten Anweisungsblöcken dürfen Variablennamen aus umschließenden Anweisungsblöcken der gleichen Methode nicht verdeckt werden*

Java Klassen: Index (1)

Java Klassen: Index (2)