

---

---

---

---

---

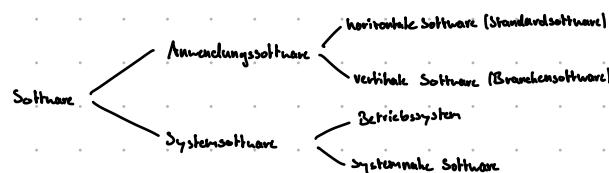


# Software

## Definition (eine):

- Sammlungseinheit für Programme, die für den Betrieb von Rechensystemen zur Verfügung stehen, einschl. der zugehörigen Dokumentation" (Brockhaus)
- => Programme und Daten mit Dokumentation, immateriell

## Kategorien:



**Systemsoftware:** Gesamtheit aller Programme die unmittelbar zum Betrieb des CSystems erforderlich sind.

**systemnahe:** Dienst- und Hilfsprogramme (Tools) die zur Administration und Pflege des Betriebssystems benötigt werden.

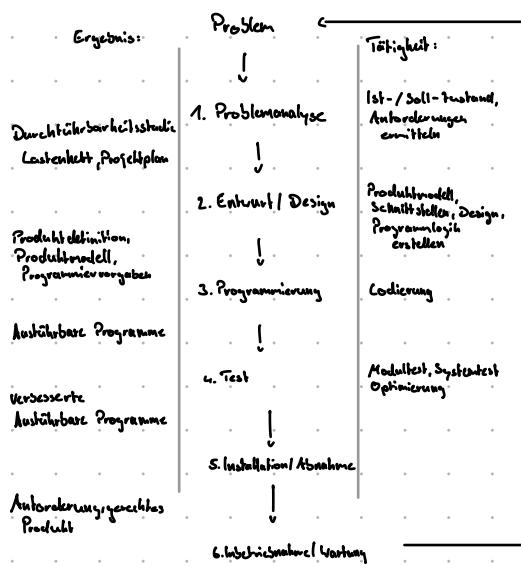
**Anwendungssoftware:** alle Prog. mit denen ein Computer für gewünschte Verwendungszwecke genutzt werden kann.

vertikale ... bsp: CAD, Ingenieurprogramme, Progr. f. Architekten, ...  
horizontale ... bsp: Textverarbeitung, Datenbank, Tabellenkalkulation, ...

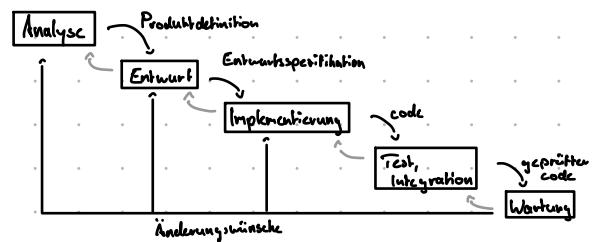
**freie Software:** - Freiheit des Nutzers im Mittelpunkt  
→ bei Erwerb der Software unterstehen der Nutzungsrrechte werden nicht vorbehaltet oder beschränkt.

**proprietäre Software:** - Recht und Möglichkeiten der Wieder- und Weiterverwendung, sowie Änderung und Anpassung durch Nutzer und Dritte stark eingeschränkt  
- Mechanismen: Softwarepatente, Urheberrecht, Lizenzbedingungen (EULA).

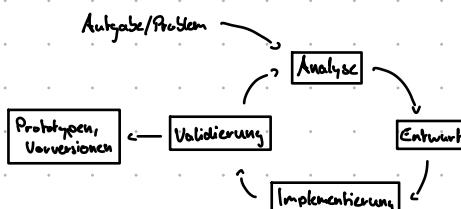
## Software Life cycle



## Wasserfallmodell



## Iterative Verfahren - Prototypbasierte Entwicklung



## Qualitätsmerkmale + Kriterien

Anforderungen an moderne Software müssen auch immer:

Anforderungen an die Softwarequalität sein

### Funktionalität "machen was es soll"

Vorhandensein von Funktionen mit festgelegten Eigenschaften  
Diese Funktionen erfüllen die definierten Anforderungen

#### Vollständigkeit

Alle spezifizierten Funktionen  
sind vorhanden

#### Richtigkeit

liefern der richtigen oder  
vereinbarten Ergebnisse  
oder Wirkungen

#### Angemessenheit

Eignung der Funktionen für  
spezifische Aufgaben

### Zuverlässigkeit "selten versagen"

Fähigkeit der Software, ihr Leistungsniveau unter  
festgelegten Bedingungen über einen festgelegten  
Zeitraum zu bewahren.  
Angemessene Anzahl von Defekten und Fehlern garantieren.

#### Reihe

Geringe Versagenshäufigkeit  
durch Fehlfunktionen

#### Robustheit

Fähigkeit, ein spezifisches  
Leistungsniveau bei Software-  
Fehlern oder Nicht-Einhaltung  
ihrer spezifizierten Schnittstelle  
zu bewahren

#### Wiederherstellbarkeit

Fähigkeit, bei einem Verlust des  
Leistungsniveaus wieder herzustellen und die  
direkt Schrottauen Daten wieder zu generieren  
Anwand und Zeit zu berücksichtigen

### Benutzbarkeit "einfach bedienen"

Aufwand der zur Benutzung erforderlich ist, und  
individuelle Beurteilung der Benutzung durch  
eine festgesetzte oder vorausgesetzte Benutzergruppe

#### Verständlichkeit

Aufwand für den Benutzer,  
das Konzept und die  
Anwendung zu verstehen

#### Erlebnisbarkeit

Aufwand für den Benutzer,  
die Anwendung zu entdecken

#### Bedienbarkeit

Aufwand für den Benutzer,  
die Anwendung zu bedienen

### Effizienz "schnell und sparsam"

Verhältnis zwischen dem Leistungsniveau der  
Software und dem Umfang der eingesetzten  
Betriebsmittel unter festgelegten Bedingungen

#### Zeitverhalten

Aufwand- und Verarbeitungszeiten  
sowie Durchsätze bei der  
Funktionsausführung

#### Verbrauchsverhalten

Anzahl und Dauer der  
benötigten Betriebsmittel  
für die Erfüllung der Funktionen

### Änderbarkeit "Kontext erweiterbar"

Nötiger Aufwand zur Durchführung von  
Änderungen  
↳ Korrekturen, Verbesserungen  
Anpassung der Umgebung, Anforderungen,  
der funktionalen Spezifikationen

#### Analysebarkeit

Aufwand um Mängel oder  
Ursachen von Versagern zu  
diagnostizieren  
oder um änderungsbedürftige  
Teile zu bestimmen

#### Prüfbarkeit

Aufwand der zur Prüfung  
der geänderten Software  
notig ist

#### Stabilität

Unschwanklichkeit des Auftretens  
unerwünschter Wirkungen von Änderungen

### Übertragbarkeit "Kontext einschätzbar"

Eignung der Software von einer Umgebung in  
eine andere übertragen zu werden  
Umgebung kann organisatorische, stand- und  
Software-Umgebung einschließen

#### Anpassbarkeit

Aufwand für die Änderungen  
die für die Anpassung an die  
andere Umgebung erforderlich sind

#### Installierbarkeit

gibt es geeignete Werkzeuge  
zur Installation und Anpassung  
der Installation in neuer Umgebung

## Struktogramme

**Schnittstelle:** wird über die gegebenen Parameter, Rückgebeteute  
und den Namen des Moduls festgelegt

**formale Parameter:** „Platzhalter“ im Kopf der Methode  
wird verwendet um einen Wert auszuzeichnen  
der vom Aufrüter eingegeben wurde

**aktuelle Parameter:** tatsächlicher Wert der vom Aufrüter  
an die Methode übergeben wird

**strukturierte Prog. Merkmale:** - Abstrakt → Kontext  
- top-down: vom umfassenden zum einzelnen Problem  
- Zerlegung problemorientiert  
- 4 Kontrollstrukturen: - Sequenz - Ablauf  
- - Schleife - - Programm-Modul/Ablauf

**Algorithmus:** eindeutige Handlungsvorschrift zur Lösung eines Problems  
- besteht aus endlich vielen, wohl definierten Eindringschritten  
- bestimmte Eingabe wird in bestimmte Ausgabe überführt

# Programmiersprachen

## Imperative...

Algorithmus durch festgelegte Reihenfolge von Anweisungen

### Interpreter-Sprachen

z.B. Python

Quellcode befehlssweise in MaschinenSprache und wird sofort ausgeführt

Langsame Ausführung: 1000 mal Übersetzen von Anweisungen  
→ 1000 mal Schleife durchlaufen

### Compiler-Sprachen

z.B. C, C++

erstellen einer ausführbaren Datei für einen Rechnertyp mittels Compiler (Rechnertypabhängig)

Schnelle Ausführung: direktes Ausführen der Maschinensprache

## Declarative... z.B. SQL

Beschreiben nicht Algorithmen sondern gewünschtes Ergebnis  
→ Lösung automatisch

## Java Allgemeines

### Ziele:

einfach	reduzierter Sprachumfang, im Vergleich zu z.B. C++, werden u.a. Mehrfachvererbung oder Zeiger nicht unterstützt
objektorientiert	Java gehört zu den objektorientierten Programmiersprachen.
verteilt	die Entwicklung von verteilten Anwendungen in einem Netzwerk wird durch umfangreiche Bibliotheken unterstützt, wie z.B. TCP/IP, RMI oder Webservices
vertraut	die syntaktische Ähnlichkeit zu anderen Programmiersprachen erlaubt ein einfaches Umstellen auf die Sprache
robust	Programmabstürze oder Systemfehler zur Laufzeit werden durch zahlreiche Schutzmechanismen vermieden oder verhindert (z.B. Garbage Collector, Exceptionhandling)
sicher	verschiedene Konzepte, die den unbefugten Zugriff auf Programmobjekte verhindern (z.B. Code-Verifier, Class-Loader, Security-Manager)
plattformunabhängig	Javaprogramme laufen architekturneutral auf allen Rechnerplattformen, für die eine entsprechende Laufzeitumgebung existiert
portabel	Java-Quellcode enthält keine betriebssystem- oder prozessorspezifischen Eigenschaften. So werden z.B. primitive Datentypen bzgl. Größe, interner Darstellung und arithmetischem Verhalten standardisiert. Ein int und float haben z.B. immer 4 Byte, eine grafische Benutzeroberfläche sieht auf allen Betriebssystemen gleich aus.
performant	Die im Vergleich zu Compilersprachen anfänglich langsame Ausführungsgeschwindigkeit wurde in letzter Zeit durch JustInTime-Compiler und HotSpot-Techniken deutlich verbessert und liefert ähnliche Geschwindigkeiten wie C++ oder C# Programme.
parallelisierbar	Java unterstützt Multithreading, also den parallelen Ablauf von eigenständigen Programm "fäden".
dynamisch	In Java können zur Laufzeit Module dynamisch geladen werden, ohne dass das gesamte Programm neu ausgeliefert werden muss. Interfaces gewährleisten die Schnittstelle, die Implementierung kann sich aber während der Laufzeit ändern.

## Variablen

### Definition

Eine Variable ist eine Speicherstelle im Hauptspeicher, an der ein Programm Werte speichern kann.

typischer: jede Variable - Datentyp welche Art ansprechbaren { Declarieren  
- Bezeichnung  
- Wert vor erster Verwendung } Initialisieren

### Interpreter/Compiler-Sprachen

z.B.: Java

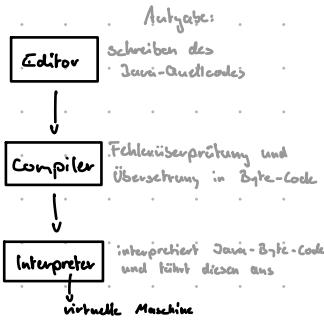
- erstellen einer plattformunabhängigen Datei (Java-Byte-Code)

- Rechnerorientierter Interpreter übersetzt Datei in Maschinensprache → führt Zeile aus

Mittlere Ausführungszeit:

- Übersetzen Java-Byte-Code schneller als
  - von Menschen geschriebener Sprache
- Aber langsamer als reine Maschinensprache

### Entwicklungsumgebung → Welche Programme notwendig



### Declaracion / Initialisierung

(<Zugriffsart>) <Typ> <Bezeichnung> [=<Wert>]

Declarieren

Initialisieren

### Datentypen kurz

einzelne / Referenzvariablen  
byte... string/object...

## Konsolenausgabe

Formuliert  $\rightarrow \text{x}, \text{usw.}$

<code>%d</code>	Ganzzahl als Dezimalzahl ausgeben
<code>%h</code>	Ganzzahl als Hexadezimalzahl ausgeben
<code>%f</code>	Fließkommazahl (double, float, ...) ausgeben
<code>%e</code>	Fließkommazahl (double, float, ...) in wissenschaftlicher Notation ausgeben
<code>%b</code>	boolean Wert ausgeben
<code>%s</code>	String wortwörtlich ausgeben
<code>%t</code>	Datum / Zeit
<code>%%</code>	Ersatzdarstellung wenn tatsächlich wortwörtlich ein Prozentzeichen ausgegeben werden soll
<code>%n</code>	Gibt <CR><LF> aus und erzeugt damit einen Zeilenvorschub

`%H` hash code  
`(%a` Integer Hexadezimal)  
`%C` Character

Fixstellige, füllende Nullen  
 $x \cdot 10^y \cdot z$   
 reservierter Platz  
 schneide nach y Zeichen ab  
 oder  
 runde auf y Nachkommastellen

$$26/10 = 3 \text{ da int/long}$$



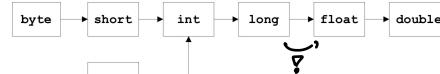
## Datentypen

		Länge in Bit	Wertebereich
<b>Ganzzahlen</b>	byte	8	$-2^7 \text{ bis } 2^7 - 1$
	short	16	$-2^{15} \text{ bis } 2^{15} - 1$
	int	32	$-2^{31} \text{ bis } 2^{31} - 1$
	long	64	$-2^{63} \text{ bis } 2^{63} - 1$
<b>Geitpunkt</b>	float	32	$-3,4 \cdot 10^{-38} \text{ bis } 3,4 \cdot 10^{38}$
	double	64	$-1,7 \cdot 10^{-308} \text{ bis } 1,7 \cdot 10^{308}$
<b>Zeichen</b>	char	16	alle Unicode-Zeichen 0-65535
<b>logisch</b>	boolean	2	true/false

## Typecasting

Ein Typ wird in einen anderen umgewandelt

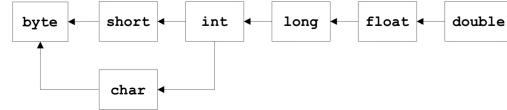
Erweitert: (implizit)  $\rightarrow$  Computer selber



Verlust möglich, da Mantisse nur auf 8 Stellen zu groß ist/long  $\Rightarrow$  ungenauer Wert  $\rightarrow$  Verlust

z.B.: Rechnung immer double, wenn double beteiligt ist

Einschränkend: type-mismatch kann entstehen werden



bsp: `int i2 = ...; i2 = t2; ? Type mismatch`  
`float t2 = 187.2f; i2 = (int) t2; kein Fehler aber`  
`i2 = 187 runden abgeschnitten?`

## Kontrollstrukturen

### if-else

```
if (Bedingung){ ... }
else{ ... }
```

### switch-case:

```
switch(variable){
    case Bed...: "1": break;
    case 2: ... break;
    default: ...
}
```

} es können auch mehrere Fälle dasselbe Ergebnis haben

### while:

```
int i=0;
while(Bedingung){
    Inhalt
    i++;
}
```

### for:

```
for (int i=0; i<10; i++){
    Inhalt
}
```

## Operatoren

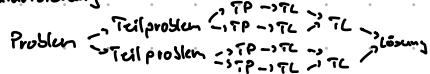
+	Addition	
-	Subtraktion	
*	Multiplikation	
/	Division	*
%	Modulo	Divisionsrest
++	Inkrement	<code>i++ entspr. i = i+1</code>
--	Dekrement	<code>i = i - 1</code>
==	Vergleich	= Zuweisung**
<	Kleiner	<= Kleiner gleich
>	Größer	>= Größer gleich
!=	Ungleich	! logisches NICHT
	logisches ODER	&& logisches UND

\*  $x / y$  ergibt den Quotienten von  $x$  und  $y$ . Sind  $x$  und  $y$  ganzzahlig, so ist auch  $x / y$  ganzzahlig (z.B.  $9 / 4$  liefert 2).  
\*\*  $a == b$  liefert `true` oder `false`

# Klassenmethoden

## Allgemein

Modularisierung:



Module lösen Teilprobleme, können beliebig oft wieder verwendet werden.

in Java: statische Methoden

<Zugriffsart> static <Rückgabetyp> <Bezeichner> (<Parameter>) { ... }  
 hier public                              void / typ  
   ↳ return typ;

## Aufbruch

Klassennamе.методенBerechnung (Parameter);

## Zusammenfassen

mehrere Klassenmethoden desselben Themas können in einer Klasse zusammengefasst werden  
 Klasse → Container für Sammlung

## Regeln Erstellung

- Gibt es eine sinnvolle (Teil-)aufgabe?
- Kann das Modul die (Teil-)Aufgabe vollständig lösen?
- Ist die Schnittstelle überschaubar?
  - Können oder müssen zur Lösung Parameter übergeben werden?
  - Ist die Anzahl der zu übergebenden Parameter "gering"?
  - Kann das Modul ein Ergebnis liefern?
  - Wird das Modul dadurch flexibler einsetzbar?
  - Hat das Modul Wiederverwendungscharakter?
  - Wird die Gesamtaufgabe dadurch überschaubarer?

## z.Zahl

$$\begin{aligned} \text{Math.Random();} &\Rightarrow [0.0; 1) \quad 0 - 0.999... \\ 1. \text{ Fehlernumm: } &z_1 \text{ bis } z_2 \quad (\underline{\text{int}}(\text{Math.Random()}) \cdot ((z_2 - z_1) + 1)) \\ &z_2 - z_1 + 1 \text{ verschiedene Werte } [0; z_2] \quad \text{Sop: } 0,6 \div 2 = 0,3 \Rightarrow 1 \\ 2. \text{ Verschieben um: } &z_1. \quad (\underline{\text{int}}(\text{Math.Random()}) \cdot ((z_2 - z_1) + 1)) + z_1 \quad [z_1; z_2] \\ &y = mx + c \quad y \in [z_1; z_2] \\ &m = \frac{z_2 - z_1}{z_2 - z_1} \end{aligned}$$

# OOP

## Grundlagen

OOP: Vorgehensmodell/Methode bei der die Ergebnisse der Phasen Analyse, Definition, Entwurf und Implementierung objektorientiert erstellt werden  
 reale Welt wird mit Objekten modelliert

Wiederverwendbarkeit: besser, da Objekte klar definiert sind (Daten, Methoden)

Datenkopplung: inkorrekte Zugriffe und Fehlermöglichkeiten werden ausgeschlossen.

**Klasse:** Setzt sich aus Attributien zusammen und besitzt Methoden

Dient als "Vorlage" für Objekte

eine Menge mit Objekten mit gleichen Attributien und Methoden

Definition: (<Zugriffsart>) class <Bezeichner> (**extends** <Oberklasse>) { ... }

**Objekt:** konkretes "Ding" einer Klasse. (z.B. Kollege <-> Lukas)

besitzt einen Zustand und reagiert mit einem Verhalten

**Zustand:** wird durch Attribute und -Werte eines Objektes bestimmt

**Attribut:** Daten die von einem Objekt angenommen werden können

**Verhalten:** wird durch Menge der Methoden bestimmt

**Methode:** Funktionen, die das Objekt machen kann. (z.B. fahren)

Definition: (<Zugriffsart>) <Rückgabetyp> <Bezeichner> (<Parameter>) { ... }

**Konstruktor:** Legt Attribute beim Anlegen des Objekts fest (-> Initialisierung)

weniger wichtig

**Referenzvariable:** Verweis auf einen Ort, wo sich der Wert/des Objekt befindet  
 erkenntlich: - haben als Typ eine Klasse (für einfacher Datentyp)  
 - Konstruktor mit Schlüsselwort new  
 - bei String implicit  
 => Objekte

**Initialwert:** einfache Datentypen 0 / 0.0 / false / \n\n\n\n\n

**Objekt (Inhalt):** nichts. (muss explizit initialisiert werden)

**Objektmethode:** - muss mit einem konkret initialisierten Objekt aufgerufen werden  
 ↳ muss mit Objektname aufgerufen werden  
 ↳ vor Methode steht neue

**Konstruktor:** - muss immer mit Schlüsselwort new aufrufen  
 - hat gleichen Namen wie die Klasse  
 -> Objekt initialisierung

**Klassenmethode:** wird mit dem Namen der Klasse aufgerufen  
 ↳ steht vor Methode  
 (in Klasse steht static vor Methode)

**Wrapperklasse:** - einfache Datentypen als Objekt  
 ↳ um Methoden (Sop: tostring()) auch mit diesen verwenden zu können  
 => Byte/Short/Integer/Float/Double/Char/Boolean

## Standardklassen

siehe 16\_OOP\_Standardklassen

- String / StringBuilder / StringTokenizer
- LocalDate / LocalTime / LocalDateTime
  - ↳ DateTimeFormatters
- Random
- Formatter (String / Decimal)

## Eigene Klassen

### Startklasse

- startet ein Programm
- enthält main-Methode

### Fachkonzeptklassen

- modellieren fachliche Logik
- beschreiben Menge von Methoden und Attributten

### UI-Klasse

- Bildschirmanzeigen
- Tastatureingaben

### Container-Klasse

- Datenhaltung, Speicherung
- der erlaubten Daten

## UML

Klassenattribute / -methoden unterstrichen

private / protected / public : - / + / +

Name : Typ

### Konstruktorverkettung

this(); gleiche Klasse  
super(); Vaterklasse

### Garbage Collector

Achtete auf die keine Referenzvariable referenziert.  
→ nicht mehr auffindbar

### Call by value

Wenden Parameter an Methode übergeben  
werden diese als Kopie übergetragen  
→ Wert der Variablen wird an  
/ einen anderen Speicherplatz kopiert  
value      ↗ ED: Wert i.e. toll  
              ↗ OD: Wert ist Referenz  
              ↗ original kann verändert werden

### Aktionsbereich

public: - Klasse selbst  
          - Vererbung  
          - seltener package  
          - in anderen Klassen

protected: - Klasse selbst  
          - Vererbung  
          - seltener package

nichts: - Klasse selbst  
          - seltener package

private: - Klasse selbst

private-Elemente werden immer dann verwendet, wenn implementierungsabhängige Details versteckt werden sollen und die auch in abgeleiteten Klassen nicht sichtbar sein sollen.

protected-Elemente werden dann verwendet, wenn auf sie innerhalb der Vererbungshierarchie zugegriffen werden soll. So können z.B. Attribute aus einer Oberklasse in einer Unterklasse direkt verwendet werden, ohne den entsprechenden Getter oder Setter zu benutzen.

public-Elemente stellen den für alle sichtbaren Teil der Klasse dar. Sie bilden sozusagen ihre Schnittstelle.

friendly-Elemente verhalten sich außerhalb des package wie private und innerhalb des package wie public.

## Achimsprinzip

direkter, unkontrollierter Zugriff

soll verhindert werden  
↳ Zugriff über Methoden

AFB Verwendung bis GUI  
22 - 27

# Vereinigung + Assoziationen (→ UML-Diagramme)

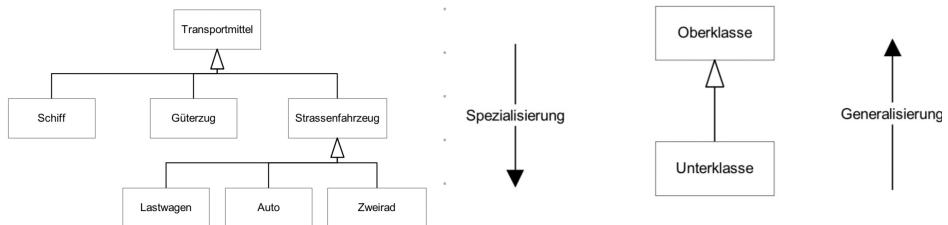
## Vereinigung "extends"

- Möglichkeit auf der Basis einer Oberklasse eine andere Klasse (→ Unterklasse) ableiten
- UK erhält dabei Methoden + Attribute der OK
- private wird nicht vereint
- UK können zusätzlich erweitert werden → Spezialisierung

- ! - Einheitvereinigung (keine OK)  
 - final → keine UK möglich  
 - UK verlustkompatibel in OK  
 ↳ Transportmittel [1] kann zu mehr Transportmittel [0..\*]  
 ↳ Fahrzeuge [0..\*] kann zu mehr Fahrzeugen [0..\*]

- Weiteres:  
 - Konstruktor wird nicht vereint  
 - geerbte Methoden können überschrieben werden  
 - super für OK Methoden + Konstruktor  
 - „late Binding“ → erst zur Laufzeit klar welche Meth. aufgerufen wird  
 - final → symbolische Konstanten  
 - viele Überschriften  
 - Klassen: keine UK

Beispiel:



→ : Leserichtung

- A  $\xrightarrow{\text{ist ein}} \rightarrow$  B Vereinigung  
 A  $\xrightarrow{\text{kein}}$  B Assoziation  
 A  $\xrightarrow{\text{hat}}$  B Aggregation  
 A  $\xrightarrow{\text{ist Teil von}}$  B Kompositum

Folgende Kardinalitäten sind möglich:

1	genau 1
0..1	0 bis 1
*	0 bis viele
3..*	3 bis viele
2, 4, 6	2, 4 oder 6
1..5, 7..*	nicht 6

## Abstrakte Klassen

Vorlage für UK; UK MUSS gebildet werden;  
 kein Objekt von abstrakter Klasse möglich  
 „abstract“ UK: kurz

## Interface „implements“

Klasse muss abstrakte Methoden implementieren,  
 sonst selbst abstrakte Klasse  
 1 Klasse → 1 Klasse einen / mehrere Interfaces  
 verwendet für Schnittstellen logisch unabhängiger Klassen

## Abstrakte Methode

keine Implementierung, nur Deklaration  
 1 abstrakte Methode => abstrakte Klasse  
 Implementierung muss in UK  
 → alle UK benötigen diese Methode

Mit Typangabe: Interface < Typ?

- ++: - Datentypangabe vom Compiler erwartet  
 - kein Typcasting notwendig

Vergleich abstrakter Klassen mit Interfaces

Abstrakte Klassen	Interfaces
Objektvariablen und Objektmethoden können vererbt werden. Enthalten abstrakte Methoden.	Enthalten symbolische Konstanten, Klassen- und abstrakte Methoden.
- Keine Mehrfachvererbung. Eine Kind-Klasse kann neben einer abstrakten Vaterklasse keine weiteren Vater-Klassen besitzen. werden abgeleitet (extends)	+ Eine Klasse kann beliebig viele Interfaces implementieren und zusätzlich eine Vaterklassen besitzen. Schlüsselwörter müssen angegeben werden (public, abstract, final,...)

## Assoziation

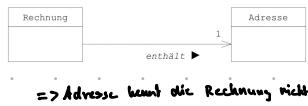
Verbindung zwischen Klassen (eig. Objekten)  
 hat Namen → Verb in URL unter Linie + kurzw.  
 + Mengenverhältnis → Kardinalität

Bsp:



- 1 Kunde besitzt 0 bis unendlich Konten } da kein „->“  
 1 Konto gehört 1 Kunden } bidirektional / umgedreht  
 ↳ Kunde kennt Konto  
 Konto kennt Kunde

## gewichtete Assoziationen



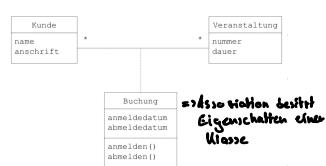
=> Adresse kennt die Rechnung nicht

## reflexive Assoziation



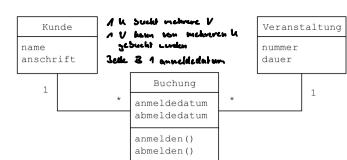
=> Assoziation zwischen Objekten derselben Klasse

## Assoziative Klasse



=> Assoziation besitzt Eigenschaften einer Klasse

## Realisierung:



## Collections

↳ Datenstrukturen, Zugriff nur über vorgegebene Methoden möglich  
 Speicherung + Verwaltung großer Datensätze

### List, Interface

ArrayList dynamisch vergrößert  
 Index

Set, Interface keine doppelten

Map, Map, KV, Map, KV Interface Key-Value-Paare  
 keine doppelten Keys

HashMap, KV Hash-Funktion auf Schlüssel  
 sollte immer nur ein FSK geliefert sein

## Exceptions

↳ Fehler während der Laufzeit

↳ kann abgeleitet oder gelungen werden

throws

try-catch

↳ try-with-resources  
 (nach try/catch-Block um Ressourcen zu schließen)

(um Ressourcen nach automatisch zu schließen)

Klausur

AS Animationen alles (außer Seile + Rechtecke/Kreise zeichnen)

Multithreading

↳ (programme umschreiben)

30+29 Skript

Buffered Reader / Writer

h