SE I Zusammenfassung
Version 3.0, Januar 2012

Java Level 1 - 4

© Maike Paetzel
(8paetzel@informatik.uni-hamburg.de)

Formaler Aufbau eines Programms

Objekte:

- Exemplare von Klassen (i*nstance of*)
- erzeugt durch Konstruktoraufrufe
- werden durch Methodenaufrufe benutzt, die den Zustand des Objektes verändern
- haben zu jedem Zeitpunkt einen internen, veränderlichen Zustand (wird durch die aktuellen Werte seiner Zustandsfelder/ Exemplarvariablen bestimmt)
- haben durch Speicheradressen eine eindeutige und feste Identität
- haben *keine* Namen
- können mit anderen Objekten interagieren
- können sowohl Dienstleister als auch Klienten sein
- können mehr als einen Typ haben

Klassen:

- „Bauplan“ für gleichartige Objekte
- definiert für ihre Exemplare, welche Methoden an ihnen aufrufbar sind, also ihr prinzipielles Verhalten
- von einer Klasse können beliebig viele Exemplare erzeugt werden
- legt fest, durch welche Methoden der Zustand der Exemplare beobachtet werden darf
- legt die möglichen Zustände und Zustandsänderungen fest
- mit dem Schlüsselwort *implements* zeigt eine Klasse an, dass sie ein Interface implementiert
- existieren zur Laufzeit auch selbst als Objekte **(Klassenobjekt)** mit internem Zustand **(über Klassenvariablen)**, die Methoden anbieten **(Klassenmethoden)**
    - Klassenvariablen und Klassenmethoden werden immer als *static* deklariert

```
<Klassenname>.<Klassenoperation> (<aktuelle Parameter>);
```

Konstruktor:

- hat keinen Rückgabetyp
- heißt immer genauso wie die Klasse selbst
- bei Aufruf wird ein neues Objekt erzeugt und diesem oft auch Werte zugewiesen - es wird initialisiert

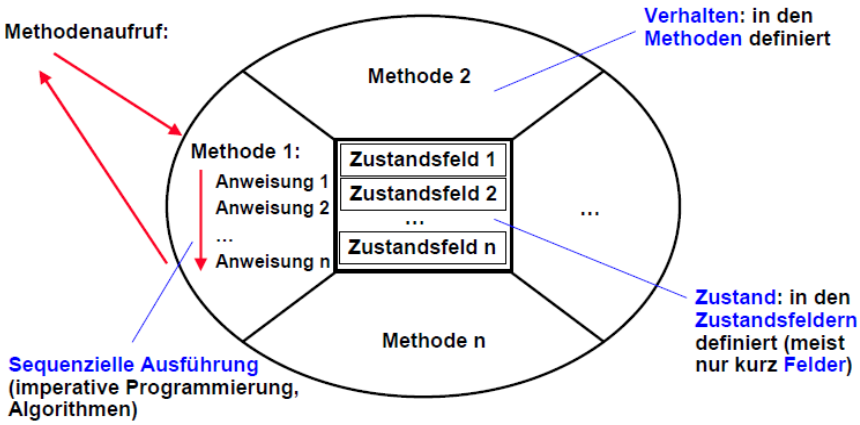
Methode:

- besteht aus Kopf (Header), der die Signatur definiert, und die Schnittstelle bildet, und dem Rumpf (Body) mit einer Sequenz von Anweisungen und Deklarationen, die Implementiation der Methode. Zum Kopf der Methode gehört auch der Methodenkommentar!
- Anweisungen definieren die eigentliche Aktion des Programms
- Mainmehtode: (sollte so wenig Code wie möglich enthalten)

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

    - Schnittstelle zum Betriebssystem
    - Einstiegspunkt für Java-Programme
- definieren das Verhalten der Exemplare
- gehören im Gegensatz zu Prozeduren immer zu einem Objekt und können auf die zugehörigen Felder zugreifen

- **verändernde:** ändern den Zustand der Methode (keine Rückgabewerte)
- **sondierende:** fragen den Zustand ab
- **public:** Methode ist von anderen Klassen aufrufbar, bilden die Schnittstelle definieren die Operationen ihres Typs
- **private:** nur innerhalb der definierten klasse aufrufbar, Hilfsmethode



Signatur: Name und Parametertypen der Methode und die Reihenfolge der Parametertypen

```
public boolean name (int i)  →  name(int)
```

Überladen: Ein Name wird für mehrere Methoden oder mehrere Konstruktoren verwendet, die sich nur durch ihre Signatur unterscheiden und ähnliche Semantik haben sollten

Punktnotation:

- üblicher Aufruf von Methoden

```
aufzurufendesObjekt.MethodenAufruf(aktuelle Parameter)
```

- innerhalb der eigenen Klasse ist die Punktnotation nicht nötig

Gleichheit:

- zwei Objekte können sich gleichen, sind aber nicht identisch
- Test auf Gleichheit bei Strings nie durch „==“, da damit nur die Referenzen verglichen werden

Identität: ein Objekt ist nur mit sich selbst identisch; impliziert Gleichheit

equals(Object obj):

- equals aus der Klasse Object vergleicht die Referenzen
- Überprüft ob Element in Sammlung enthalten
- sind zwei Elemente laut equals gleich, müssen sie den gleichen HashWert liefern

Klient:

- aufrufendes Objekt
- Objekt, das eine konkrete Dienstleistung eines anderen Objektes in Anspruch nimmt

Dienstleister:
- aufgerufenes Objekt
- Objekt, das bei einer bestimmten Aufgabe einen Dienst leistet
- Objekte bieten Dienstleistungen als Methoden an ihrer Schnittstelle an

Quelltext = Programmtext
- wird durch den Editor bearbeitet
- darin ist beschrieben, wie die Methoden realisiert sind

Klassendefinition: textuelle Beschreibung einer Klasse

Compiler: Wandelt den menschenlesbaren Quelltext in eine maschinenausführbare Form um

Sammlungsbibliotheken:
- stellen verschiedene Interfaces und Klassen zur Verfügung
- in packages organisiert
- können mittels der *import*-Anweisung eingebunden werden, es kann jedoch auch bei jedem Aufruf der vollständig qualifizierte Name wie *java.util.Set* verwendet werden

Block: Sequenz von Anweisungen in geschweiften Klammern {} in denen auch lokale Variablen deklariert werden können, bilden einen eigenen Sichtbarkeitsbereich.
Programmiersprachen, die Blöcke kennen, heißen blockstrukturiert.

Zusammengesetzte Anweisungen: Blöcke ohne lokale Variablen

Schnittstelle: Methoden, die an den Exemplaren einer Klasse aufrufbar sind

Benannte Schnittstellen = Interfaces:
- programmiersprachliche Formulierung der Schnittstelle einer Klasse (interface)
- abstrahiert von konkreten Implementierungsdetails,
- nur die Köpfe der öffentlichen Methoden
- kann keine Konstruktoren definieren => mit Interfaces können keine Objekte erzeugt werden
- **Operationen** = Methoden eines Interfaces

Aufrufstack: Ablage von lokalen Variablen, werden beim Methodenaufruf darauf geschrieben und beim *return* wieder entfernt

Stack-Limit:
- maximale Größe des Aufrufstacks
- throws *StackOverflowException*
- legt fest, wie tief geschachtelte Methodenaufrufe vorkommen dürfen

Heap:
- dient zum Verwalten der erzeugten Objekte
- throws *OutOfMemoryException*
- ein mittels *new* erzeugtes Objekt wird hineingeschrieben und bleibt mindestens solange wie es Referenzen vom Stack darauf gibt

Funktion:

```
<modifier><Ergebnistyp> <Name> (Parameterliste) {Rumpf}
```

Prozedur:

```
<modifier> void <Name> (Parameterliste) {Rumpf}
```

Prozeduraufrufe:

- explizite Anweisung, dass eine Prozedur ausgeführt werden soll
- Prozedur ist aktiv, nachdem sie aufgerufen wurde bis zu ihrem Ende oder einem erreichten Abbruchkriterium
- Kontrolle wechselt vom Rufer zur Prozedur
- Werte der aktuellen Parameter werden an die formalen gebunden
- Kontrolle ist immer nur bei einer Prozedur
- Nach Abarbeitung kehrt die Kontrolle zum Rufer zurück

Kommentare: Jedes Programm muss dokumentiert werden, denn ein ungenügend kommentiertes Programm ist genau so unbrauchbar wie ein falsches Programm. Ein Programm ohne Kommentare kann nicht weiterentwickelt oder vernünftig gewartet werden, denn die Kommentare dienen zum Verständnis des Programms. Ohne Kommentare geht das Wissen über die Implementation verloren und kann nur sehr schwer und mit viel Mühe und Aufwand wiederhergestellt werden. Deshalb sollte immer frühzeitig dokumentiert werden, damit man sich selbst die Weiterentwicklung erleichtert.

- **Zeilenkommentar:** // hinter diesen Zeichen steht ein Zeilenkommentar, der mit Ende der Zeile ebenfalls endet
- **Methodenkommentare** /* Dieser Kommentar
  erstreckt sich
  über mehrere Zeilen (wenn auch grundlos) */

Spezifikation: Beschreibung einer gewünschten Funktionalität entweder informell (natürlichsprachlich) oder formal (bsp: mathematisch)

Imperative Programmierung:

- Programme werden als Folgen von Anweisungen formuliert
- Ausführungsreihenfolge durch textuelle Reihenfolge oder Sprunganweisung festgelegt
- Höhere Programmkonstrukte fassen Anweisungsfolgen zusammen und bestimmen die Ausführungsreihenfolge
- Variablen können Werte annehmen, die sich durch Anweisungen ändern lassen

Effektivität: Ein Programm ist effektiv, wenn die zur Verfügung stehenden Ressourcen an Zeit und Speicherplatz ausreichen

Effizient: Ein Programm ist effizient, wenn es dem theoretisch möglichen Minimalaufwand für die Lösung des Problems sehr nahe kommt

Laufzeit: Zeit, in der ein Programm von seinem Start bis zur Termination ausgeführt wird.
Relevante **dynamische** Eigenschaften: Semantik, Anzahl und Lebensdauer der Objekte, Methodenaufrufe und Datenmanipulation

Übersetzungszeit: Zeit, die ein Compiler benötigt um einen Quelltext in Maschinencode zu übersetzen. Relevante **statische** Eigenschaften: Syntaxregeln und Sichtbarkeitsbereiche, Struktur, Konventionen

Exemplarvariablen = Felder: halten den Zustand eines Objektes, werden in der Klassendefinition für alle Exemplare der Klasse deklariert

Token: aus Sicht einer Grammatik die kleinste unteilbare syntaktische Einheiten: Bezeichner, Literale, Operatoren, reservierte Wörter, Sonderzeichen (Klammern, Semikolon)

Kapselung:

- programmiertechnischer Mechanismus, der bestimmte Programmkonstrukte vor Zugriff von Außen schützt
- geschieht durch public – oder private Deklaration
- Ausblenden von Details vereinfacht die Benutztung
- Implementationsdetails können geändert werden, ohne dass das den Klienten betrifft
- angestrebt: Klassen als Black Box, die nur relevante Informationen nach außen zeigen

Ausdruck = Term:

- Verarbeitungsvorschrift, deren Ausführung einen Wert liefert
- Operanden werden mit Operatoren verknüpft
- reguläre Ausdrücke steuern u.A. die lexikalische Analyse eines Compilers

Von Variablen und Parametern

Variablen:

- Abstraktion eines physischen Speicherplatzes
- hat einen Namen, über den sie angesprochen werden kann
- hat einen Typ, der Wertemenge, zulässige Operationen und Eigenschaften festlegt
- hat eine Belegung, die sich ändern kann
- **lokale Variablen:** Hilfsvariable, innerhalb einer Methode deklariert und existiert nur während der Ausführung dieser Methode und ist nur in dieser sichtbar

Deklaration: Einführung von Variablen durch Angabe eines Namens und eines Typs, in Java wird der Variable auch gleich ein Standardwert zugewiesen

Variableninitialisierung: erstmalige Wertzuweisung einer Variable

Typ einer Variable: legt fest, welche Werte sie annehmen kann und die zulässigen Operationen

Primitive Typen

- **ganze Zahlen - int:** -2^(31) bis 2^(31)-1
- **reelle Zahlen - float:** 32 bit Gleitkommazahl (einfache Genauigkeit) oder **real**
- **double:** 64 bit Gleitkommazahl (doppelte Genauigkeit)
- **long:** 64 bit Ganzzahl
- **short:** 16 bit Ganzzahl
- **Zeichen – char**
- **Wahrheitswerte –** boolean

Objekttypen

- **String**
- **Objekname**

Genauigkeitsgrenze: Werte, die keine Summe von Zweierpotenzen sind, können nicht exakt dargestellt werden

Typumwandlungen = Typkonversion = Typkonvertierung

- Anpassung von einem Typ in einen anderen
- **implizit:** automatische Umwandlung, wenn der der Zieltyp eine höhere Genauigkeit als der Typ des Ausdrucks hat, da keine Informationen verloren gehen können
- **explizit:** erzwungene Umwandlung, da der Typ des Ausdrucks eine höhere Genauigkeit als der des Zieltyps hat, gewünschter Typ wird in Klammern vor den umzuwandelnden Ausdruck geschrieben

Operatoren:

- verknüpfen Operanden zu Ausdrücken
- haben verschieden hohe Präzedenzen (Ränge)
- Vorrangregeln bestimmen Auswertungsreihenfolge
- defensives Klammern erhöht Lesbarkeit!

Operator	Funktion, arithmetisch
*	Multiplikation
/	Division
%	Modulo
+	Addition
-	Subtraktion
++	Inkrement
--	Dekrement

Operator	Funktion, boolesche
!	logisches NICHT
<	„kleiner als“
<=	„kleiner gleich“
>	„größer als“
>=	„größer gleich“
&&	logisches UND
||	logisches ODER

Operator	Funktion
==	Gleichheit
!=	Ungleichheit
=	Zuweisung

Parameter: bestehen aus Name und Typ

1. **aktuell:** tatsächliche Werte an der Aufrufstelle (beim Klienten)
2. **formal:** Namen durch den der Parameter verändert werden kann, Parametrisierung von Methoden, werden in den jeweiligen Methodenköpfen deklariert

Parameterübergabe: aktuelle Parameter werden an formale gebunden

Zuweisung:

- Ausdruck auf der rechten Seite wird Variablen auf der linken Seite zugewiesen
- rechte Seite: arithmetische oder boolsche Ausdrücke, Vergleiche und Zeichen oder Zeichenketten
- linke Seite: Bezeichner einer Variablen
- Zuweisungsoperator: „=“
- Typen müssen kompatibel sein

Sichtbarkeitsbereiche:

- Bereiche in einem Java-Programm, in denen eine Variable angesprochen und benutzt werden kann (Exemplarvariablen in der gesamten Klasse, formale Parameter in seiner Methode, lokale Variablen nach ihrer Deklaration bis zum Ende des Blocks),
- können geschachtelt sein, wobei übergeordnete Variablen auch in den geschachtelten sichtbar sind
- statisch feststellbar (Sichtbarkeitsbereich = Programmiereinheit, in der die Variable deklariert ist)

Lebensdauer:

- Zeit während einer Programmausführung, in der der Variablen oder dem Objekt Speicher zugeteilt ist, sie also existieren.
- (Lebensdauer einer Referenzvariable != Lebensdauer referenziertes Objekt)
- dynamische Eigenschaft

Literal: Zeichenfolge, die einen Wert repräsentiert und einen Typ hat

(Exemplar-)Konstanten: Variablen mit dem Zusatz final, die nicht mehr veränderbar sind

static: Variablen oder Operationen, die mit *static* deklariert sind, heißen Klassenvariablen oder Klassenoperationen. Sie existieren unabhängig von einem konkreten Objekt , das heißt ihre Lebensdauer erstreckt sich vom Laden der Klasse bis zum Beenden des Programms.

Klassenkonstante: Variablen, die als public, static und final deklariert werden

Strings:

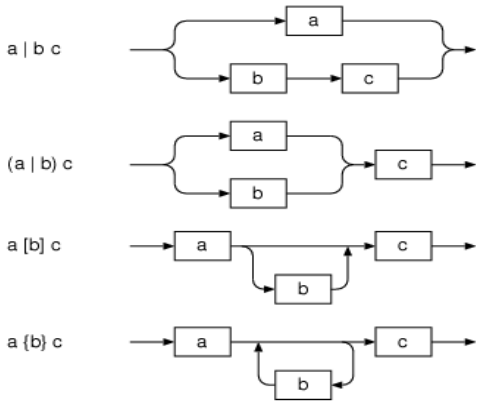
- Zeichenketten aus java.lang.String
- Folge einzelner Zeichen
- (untypisches) Objekt, da es keinen veränderbaren Zustand hat
- Vergleich immer durch *equals*
- Infix- Operator (ein Operator, der zwischen zwei Operanden steht) „+“ verkettet zwei Strings zu einem neuen String
- Länge unbegrenzt
- nach dem Erzeugen nicht mehr veränderbar , alle scheinbaren Veränderungen liefern in Wirklichkeit ein neues String-Objekt zurück!
- können durch String-Literale zwischen doppelten Anführungszeichen erzeugt werden

Wann ist Quelltext korrekt?

Syntax:

- Gibt den Aufbau des Quelltextes vor
- definiert in kontextfreier Grammatik
- Struktur von Klassendefinitionen, Methoden und Anweisungen
- besteht aus Ableitungsregeln
- es gibt ein festgelegtes Nichtterminal als Startpunkt
- Notation: EBNF (erweiterte Backus-Naur-Form)
- ein übersetzbarer Quelltext muss der Syntax entsprechen

- **Wiederholbare Elemente** werden in geschweifte Klammern geschrieben
- **Optionale Elemente** in eckige Klammern
- **Auswahlelemente** in runden Klammern



Ableitungsregel: Nichtterminal (Element) auf der linken Seite kann durch das Nichtterminal oder Terminal auf der rechten ersetzt werden

- **Terminal:** Basisbaustein
- **Nichtterminal:** eine Regel muss dafür definiert sein, bis es zu einem Terminal wird

Semantik: Semantik ist die inhaltliche Bedeutung, die festlegt, was das Programm bei Ausführung bewirkt

Pragmatik: Gebrauch einer Sprache, bestimmt durch Zweck, Aufgabenstellung und Verwendung

Konventionen: u.A. Klassennamen groß, Variablen- und Methodennamen klein

Kontrollstrukturen

Strukturierte Programmierung: Beschränkung auf die Kontrollstrukturen Sequenz, Wiederholung und Auswahl, werden durch **Flussdiagramme** oder **Struktugramme** dargestellt

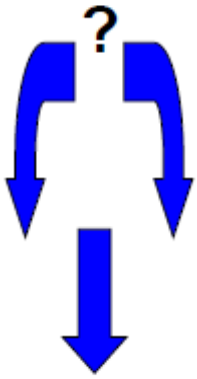
Konrollstrukturen: Reihenfolge der Ausführung von Anweisungen im Programm

Sequenz:

- Aneinanderreihung von Ausdrücken getrennt durch Semikolon
- Anweisungen werden nacheinander abgearbeitet
- Anweisung kann auch eine leere Aktion sein

Fallunterscheidung: Abhängig von einer Bedingung (boolscher Ausdruck) wird eine Anweisung ausgeführt

```
if (<Bedingung>)        if (<Bedingung>)
{                       {
     <Anweisung>;            <Anweisung>;
     <Anweisung>;            <Anweisung>;
}                       }
                        else
                             {
                                  <Anweisung>;
                                  <Anweisung>;
                              }
```



Auswahlanweisung: (Mehrweg- Verzweigung)

```
switch (<Ausdruck>)
{
     case <Wert1> : <Anweisung1>; break;
     case <Wert2> : <Anweisung2>; break;
     default      : <Anweisung3>; break;
}
```

Schleifenkonstrukte: eine Reihe von Anweisungen wird keinmal, einmal oder mehrfach ausgeführt

Zählschleife:

```
For (<Initialisierung>; <Abbruchtest>; <Schritt>)
{
     <Anweisung>;
     <Anweisung>;
}
```

Bedingte Schleife: Anweisungen werden wiederholt ausgeführt, das Ende ist mit einer logischen Bedingung verknüpft

```
while                   do
(<Bedingung>)           {
{                            <Anweisung>;
     <Anweisung>;            <Anweisung>;
     <Anweisung>;       }
}                       while (<Bedingung>);
```

Erweiterte For-Schleife: Durchläuft alle Elemente einer Sammlung

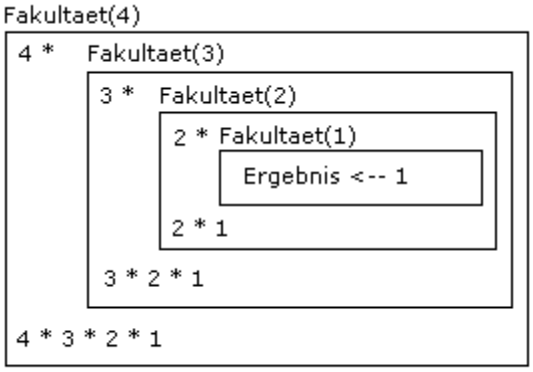
```
for (<Sammlungstyp> s: <Sammlungsname>)
{
      <Anweisung>;
      <Anweisung>;
}
```

Iteration: Wiederholung durch Schleifenkonstrukte

Rekursion: Verweis auf sich selbst, selbstaufrufende Methode

- **direkte Rekursion:** im Rumpf einer Methode x wird die Methode x selbst wieder aufgerufen
- **indirekte Rekursion:** im Rumpf einer Methode x wird eine Methode y aufgerufen, die direkt oder indirekt wieder zum Aufruf der Methode x führt
- **Rekursion oder Iteration?** - Lesbarkeit, Verständlichkeit, Rechen- und Speicheraufwand

```
Int fac (int n)                 Fac(4)
{                               // größer null
   if (n > 0)                   = 4*fac(4-1)
  {                             = 4*3*fac(3-1)
    return n* fac(n-1);         = 4*3*2*fac(2-1)
  }                             = 4*3*2*1*fac(1-1)
     else                       // gleich null
    {                           = 4*3*2*1*1= 24
      return 1;
    }
}
```



Referenzen

Referenztyp:

- legt die Menge seiner Elemente und der möglichen Operationen (Methoden die an den Exemplaren der Klasse aufgerufen werden können)
- Variable mit dem Typ einer Klasse
- Wertemenge unbeschränkt
- enthält kein Objekt sondern nur eine Referenz auf ein Objekt der Klasse

Referenzvariable:

- Belegung mit einer Referenz auf ein Objekt
- referenziertes Objekt

null: Referenzvariable, die auf kein Objekt verweist (throws *NullPointerException* bei Methodenaufruf darüber)

statischer Typ:

- deklarierter Typ einer Referenzvariablen (steht zur Übersetzungszeit fest)
- legt die Operationen fest

dynamischer Typ:

- hängt von der Klasse des Objekts ab, auf die die Refernz zur Laufzweit verweist (kann zur Laufzeit ermittelt werden und sich während der Laufzeit verändern)
- welche konkreten Methoden werden bei einem Operationsaufruf ausgeführt (dynamisches Binden)

Alias- Problem: Wenn mehrere Referenzvariablen auf dasselbe Objekt verweisen, dann kann man oft lokal nicht entscheiden, ob an dem referenzierten Objekt Änderungen vorgenommen wurden oder nicht. Das bedeutet, dass man eine Annahme über den Zustand eines Objekts trifft und dieser sich zur Laufzeit ändert. Benutzt man das Objekt, dann können sich dadurch falsche Werte ergeben, die zu fehlerhaften Verhalten führen.

Typtest: *instanceof*-Operator ermittelt, ob das referenzierte Objekt ein Exemplar einer implementierenden Klasse ist

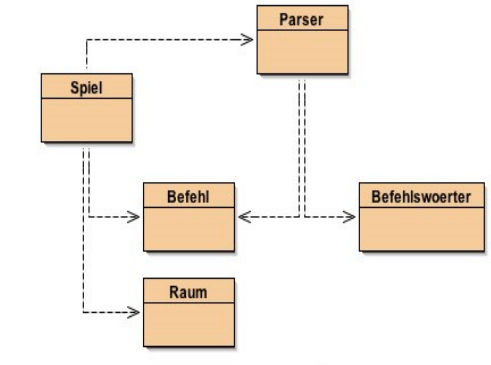
Typzusicherung = Typ Cast,: sichert zu, dass eine Operation an einem Objekt aufgerufen werden kann, die nicht im statischen Typ definiert ist

Garbage-Collection: Beseitigung von nicht mehr benötigten Objekten aus dem Speicher eines Programms (üblicherweise, wenn keine Referenzen mehr darauf sind)

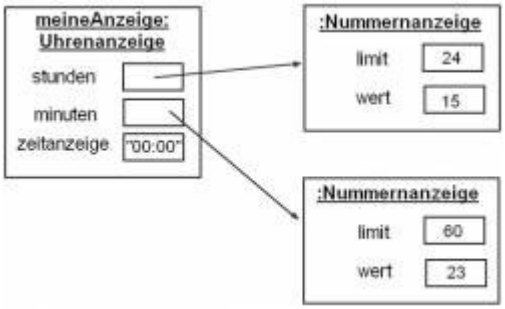
Darstellungen

UML (Unified Modeling Language): Diagramme für die Darstellung verschiedener Ansichten auf ein System

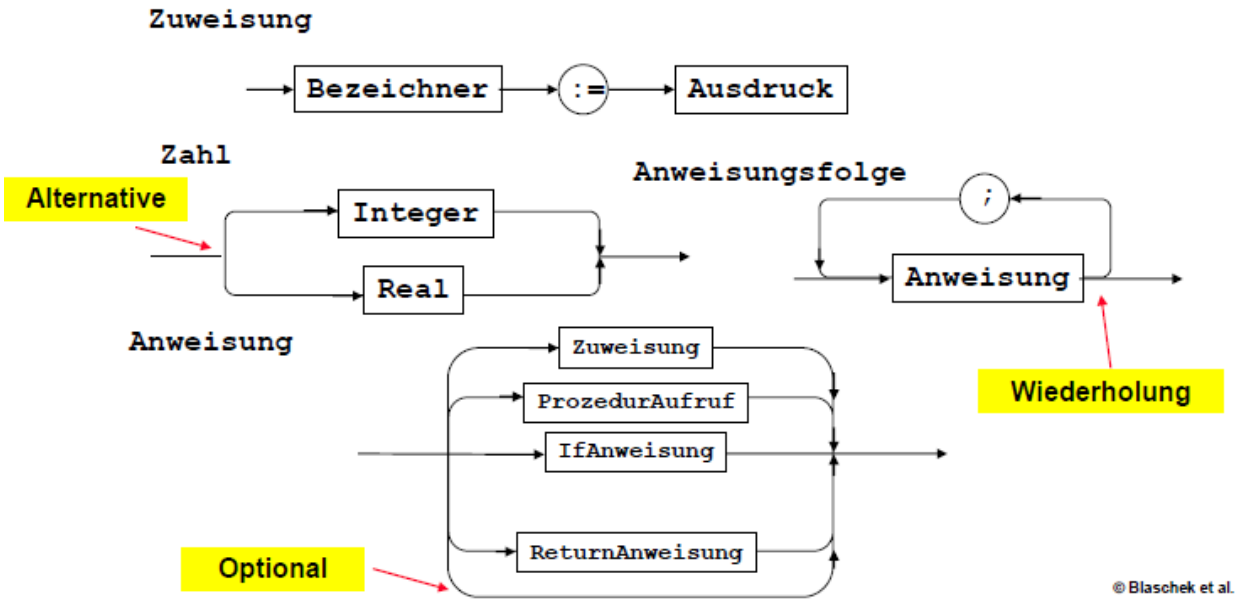
- **Klassendiagramm:** statische Sttruktur eines Systems, stellt Beziehung zwischen Klassen dar



- **Objektdiagramm:** Struktur ein Teil des Systems zu einem bestimmten Zeitpunkt während der Laufzeit, Objekte mit den Belegungen ihrer Felder



Syntaxdiagramme Um die Syntax des Programms darzustellen



Testen

Debugger:

- Unterstützt beim Finden von Problemen
- bietet die Möglichkeit den Aufruf-Stack, den Heap und Ähnliches zu überwachen

Testen:

- identifiziert Fehler, kann aber nicht die Korrektheit der Software nachweisen
- erhöht Vertrauen in die Implementation
- Maßnahme zur Qualitätssicherung
- **erschöpfende Tests:** alle gültigen Eingabewerte werden gestestet (meist nicht realistisch)
- ausreichend: wenige Vertreter der Äquivalenzklassen werden getestet, hierbei ist das Testen von Grenzwerten besonders wichtig

Äquivalenzklassen: Eingabewertbereich wird in Kategorien unterteilt (Werte eines Wertebereichs = äquivalent)

Modultests:

- isoliertes Testen softwaretechnischer Einheiten (Methoden, Klassen, Teilsysteme)
- jede Klasse sollte eine eigene Testklasse bekommen, die alle Methoden einzeln testet
- reproduzierbar
- automatisiert wiederholbar
- **Positivtest:** Überprüft die geforderte Funktionalität durch korrekte Benutzung, testet Korrektheit
- **Negativtest:** testet Robustheit, testet das Verhalten bei fehlerhafter Benutzung
- **Black-Box-Test:** Test bezieht sich nur auf die Schnittstelle
- **White-Box-Test:** versuchen alle Teile der Implementierung zu testen, auch beide Pfade von Fallunterscheidungen, stärker technisch orientiert, was tut die Software tatsächlich?
- **Schreibtischtest:** Programmtext wird auf dem Papier durchgegangen und der Programmablauf nachvollzogen, hinzuziehen einer zweiten Person sinnvoll
- **Integrationstest:** alle getesteten Einzelteile eines Systems werden in ihrem Zusammenspiel getestet

Zu viele Objekte? –> Sammlungen anlegen

Dynamische Datenstrukturen: Organisationsformen von veränderbaren Sammlungen von Objekten

- Objekte in einer dynamischen Datenstruktur werden als Elemente bezeichnet
- **Duplikat:** Einfügen eines bereits enthaltenen Elements
- **Struktur:** Gebilde aus Elementen mit Beziehungen
  - Mengen: ohne Relation
  - Listen: lineare oder sequentielle Struktur
  - Bäume: nur ein Vorgängerelement
  - Graph: beliebig verbunden
- gleichartig rekursiv aufgebaut
- Ändern einer Struktur = Hinzufügen, Löschen, Modifizieren und Ändern von Beziehungen der Elemente
- dynamisch = Datenstrukturen wachsen oder schrumpfen mit dem Löschen oder Einfügen von Elementen

Sammlungen = Collections:

- Objekt, das eine Gruppe von anderen Objekten zusammenfasst
- enthalten Objekte vom selben Typ, der als Eigenschaft der Sammlung angesehen wird
- werden benutzt um Objekte zu speichern und gemeinsam zu manipulieren oder weiterzugeben
- dynamische Behälter, in die beliebig viele Elemente eingefügt werden können und wieder entfernt werden können

Liste:

- Aneinanderreihung von gleichartigen Elementen zu Folgen
- entweder leer oder ein Listenelement gefolgt von einer Liste
- manipulierbare Reihenfolge
- Duplikate Zugelassen
- unbeschränkte Sammlung

Menge:

- Ordnung irrelevant
- Duplikate verboten

Array:

- geordnete Form gleichartiger Sammlung von Elementen
- Deklaration:
  Eindimensional:

Type [] Identifier

  Zweidimensional

Type [][] Identifier

- Zugriff auf Elemente über Index
- werden direkt auf dem zugrunde liegenden Speicher abgebildet => sehr schneller, wahlfreier Zugriff
- in Java immer in Größe festgelegt (bei ihrer Erzeugung)
- Array = Objekt - Arrayvariable = Referenzvariable
- werden mit *new*- Anweisung erzeugt

New Type[LenghExpression]	New Type[LenghExpression][LenghExpression]

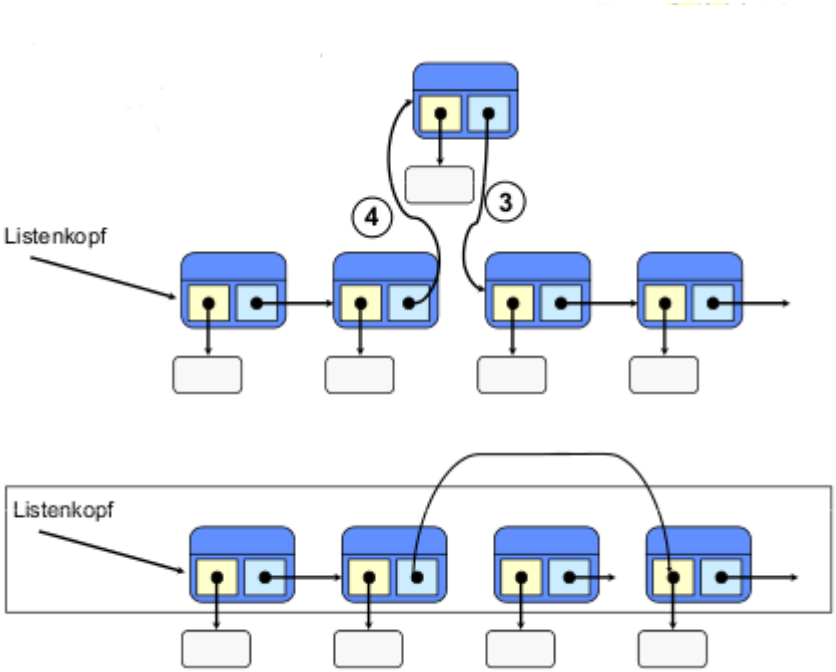
- *Arrayname.length* liefert die Länge des Arrays zurück
- Komplette Iteration über ein zweidimensionale Arrays kann durch zwei verschachtelte for-Schleifen geschehen
- **Index:** in eckigen Klammern direkt hinter dem Element, natürliche Zahl zwischen 0 und Arraygröße-1 (throws *IndexOutOfBoundsException* bei Zugriff auf nicht vorhandenen Index*)*
- **Elemente**: müssen Werte der Basistypen haben (alle Elemente vom gleichen Typ), Referenzen auf Objekte

ArrayListe:

- „wachsendes Array“, das bei initialisierung eine Größe k hat und bei bedarf mit 2*k vergrößert wird, in dem alle Elemente kopiert werden
- **Kapazität:** aktuelle Größe des Arrays, gehört nicht zur Schnittstelle
- **Kardinalität:** momentane Anzahl an Elementen in der liste, abfragbar mit *size()*
- *Kapazität >= Kardinalität*

LinkedList:

- doppelt verkettete Liste
- Verkettung von Knoten/Kettengliedern über Referenzen
- Referenz auf eigentliches Element und Referenz auf Vorgänger und Nachfolger
- **Einfach verkettete Liste:** Keine Referenz auf Vorgänger
- **Wächterknoten:** vereinfachen Sonderfälle am Listenanfang und Listenende

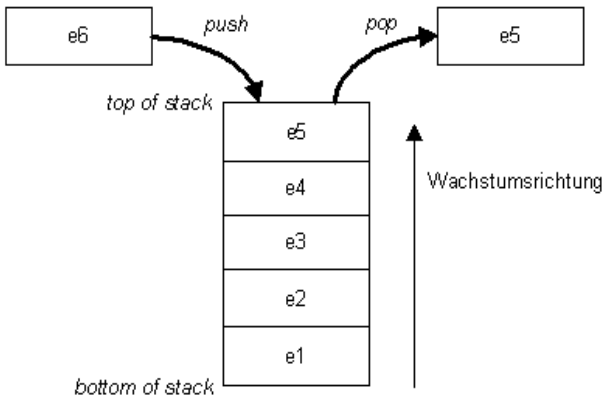


Hash-Verfahren:

- Elemente werden in einem Array von Überlaufbehältern gespeichert (Hash-Tabelle)
- Wird ein Element gesucht, wird zuerst der Index in der Hashtabelle ermittelt und nach einem schnellen indexbasierten Zugriff werden alle Elemente linear durchsucht
- mit Hilfe der Hash-Funktion wird für ein gegebenes Element ein ganzzahliger Wert berechnet und danach in eine der Listen eingefügt und geeignet auf einem Index abgebildet
- **Güte:** möglichst gute Verteilung der Elemente auf die Überlaufbehälter wird abgestrebt (ideal: Ein Element pro Überlaufbehälter)
- der Aufwand ist nicht mehr von der Kardinalität abhängig sondern aus der Indexberechnung und dem indexbasierten Zugriff auf den Überlaufbehälter
- **Die Hash-Funktion:** Sie bildet ein Element auf einem Integer ab, der die künstliche Kategorie bildet. Der berechnete Wert muss auf einen Index der Tabelle abgebildet werden
- **Kollision:** Abbildung zweier Elemente auf denselben Index

Stack = Stapel = Kellerspeicher:

- Sammlung von Elementen eines Datentyps mit eingeschränkten Einfüge- und Ausfügeoperationen



Beispiel Aufrufstack

- Verwaltung nach dem LIFO-Prinzip = Operationen beziehen sich immer auf das zuletzt eingefügte Element der Folge
- ***push***(legt ein Element ab)
- ***pop*** (entfernt das oberste Element)
- ***peek*** (fragt oberstes Element ab)
- ***isEmpty*** (prüft ob der Stack leer ist)

Queue: Warteschlange, bei der die hineingeschrieben Elemente in der gleichen Reihenfolge wieder entnommen werden (FIFO- Prinzip), es wird als stets das erste Element der Folge betrachtet

- gewichtete Schlangen sortieren die Elemente nach ihrer Gewichtung

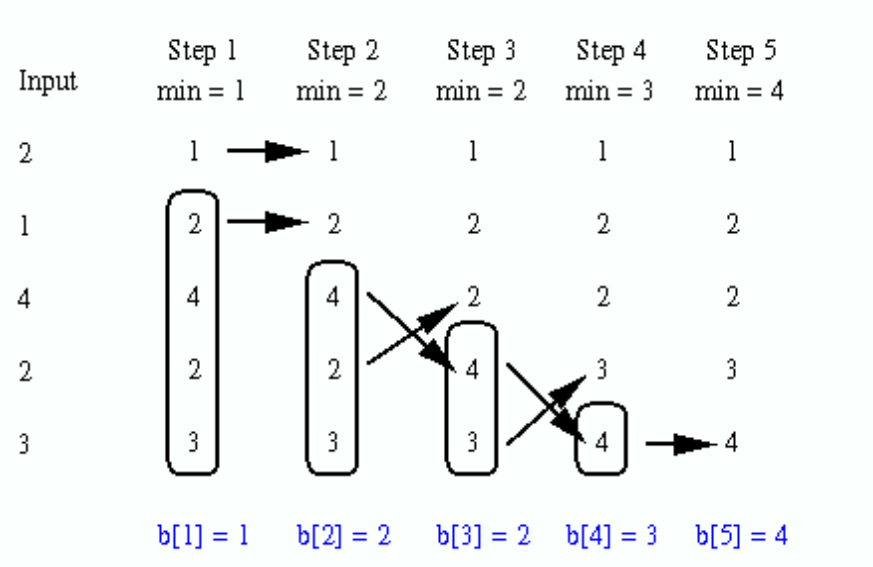
Map:

- Schlüssel-Element-Paare (*Key*-*Value*) mit eindeutigen Schlüsseln
- Schlüsseltyp und Elementtyp sind Referenztypen
- über einen Schlüssel kann auf ein Element zugegriffen werden (*get*)
- modelliert eine Abbildung von Schlüsseln auf Elemente (Telefonbuch)
- Beispiel HashMap

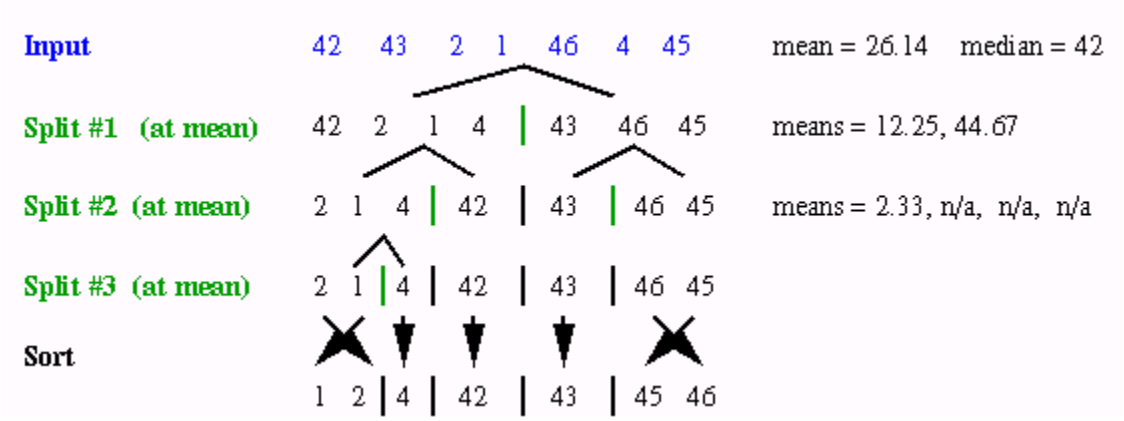
Wer suchet der findet

Sortieralgorithmen: klassifiziert nach:

- Komplexität (Wie viele Schritte sind notwendig)
- Speicherbedarf(in situ/ in place = ohne zusätzlichen Speicherbedarf)
- Stabilität: Wenn das Sortierverfahren die ursprüngliche Reihenfolge von zwei gleichen Elemente nicht verändert, gilt es als stabil
- Art: Vergleichsbasiert (meist) oder die Reihenfolge der Elemente wird direkt aus den Sortierschlüsseln berechnet
- **Bubble-Sort:** einfaches Sortierverfahren mit quadratischem Aufwand $O(n^2)$



- **Quick-Sort:** sehr schneller Sortieralgorithmus, im günstigsten Fall mit O(n log(n)), im ungünstigsten mit $O(n^2)$



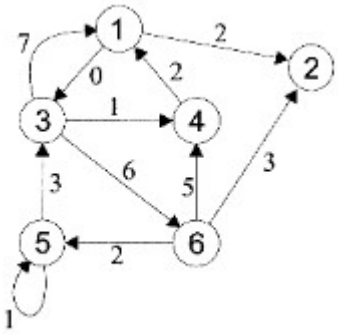
Aufwand: Der Aufwand eines Programms ist der Bedarf der Zeitressourcen, den seine Abläufe verursachen, er wird in Größenordnung O von einer Größte n der Eingabedaten gemessen

Name	Kürzel	Umgangssprachliche Übersetztung
Konstant	O(c)	Immer gleiche Arbeit
Logarithmisch	O(log(n))	10fache Arbeit
Linear	O(n)	1000fach
N log n	O(n log(n))	10000fach
Quadratisch	$O(n^2)$	Millionenfach
Exponentiell	O(2^n)	Hoffnunglos

- *istEnthalten* erfordert einen Komplexität von O(n), da das zu durchsuchende Datenkonstrukt sequentiell durchlaufen wird

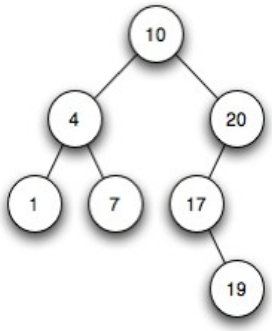
Graphen und Bäume

Graph: besteht aus einer Menge von Knoten und einer Menge von Kanten, wobei jede Kante zwei Knoten verbindet



- **gerichteter Graph:** Kante geht von V nach W (V,W)
- **ungerichteter Graph:** Kante ist zwischen V und W ohne angegebene Richtung {V, W}
- **gewichteter Graph:** Kanten wird eine Zahl (Gewicht) zugewiesen
- zwei Knoten heißen **benachbart**, wenn zwischen ihnen eine Kante exisitiert
- ein Knoten ist von einem anderen **erreichbar**, wenn es einen Pfad (Sequenz von benachbarten Knoten) zwischen Ihnen gibt
- ein Graph ist **zusammenhängend**, wenn jeder seiner Knoten von jedem anderen Knoten aus erreichbar ist

Bäume:



- Struktur, in der Knoten miteinander durch gerichtete Kanten verbunden sind
- jeder Knoten hat maximal einen Vorgängerknoten aber beliebig viele Kindknoten
- Knoten ohne Vorgänger: **Wurzel**
- Knoten ohne Nachfolger: **Blatt**
- rekursiv (Kindnoten = Wurzelknoten eines Unterbaumes)
- **binärer Baum:** Knoten hat maximal zwei Kindknoten

Binärer Suchbaum:

- **Elemente müssen sortierbar sein**
- im linken Unterbaum sind alle kleineren und im rechten Unterbaum alle größeren Elemente
- **voller Binärbaum** hat bei einer Höhe h 2^h Blätter und 2^(h+1)-1 Knoten
- **balancierter binärer Baum:** Höhe der beiden Unterknoten eines Knotens unterscheiden sich maximal um eins. Höhe = Logarithmus Dualis(n), Sortieraufwand von O(log(n))

Die Inhalte dieser Zusammenfassung beruhen auf der Veranstaltung SE1 von Axel Schmolitzky. Für die Korrektheit des Inhalts wird keine Garantie übernommen.

In der Version 3.0 sind die Korrekturen von Axel Schmolitzky und die Anmerkungen aus dem letzten Jahr eingepflegt.