Algorithmen

Algorithmen und Merkmale (ALG 1)

Ein Algorithmus…

- ist ein schrittweises Verfahren
- ist endlich
- ist ausführbar
- hat einen eindeutigen nächsten Schritt

Komplexität eines Algorithmus (ALG 1)

Unterscheidung:
– Zeitkomplexität ⇨ Anzahl benötigte Rechenschritte ⇨ Zeitabhängigkeit
– Speicherkomplexität ⇨ benötigte Speicherressourcen

Ein Algorithmus stellt also einen funktionalen Zusammenhang dar
⇨ Ordnung eines Algorithmus: f(n) = O(g(n)) O ⇨ «Big Oh», landausches Symbol

Beispiele einiger Algorithmen:	Ordnung	«Rangliste»
$f_1(n) = \ln(n) + 10000 \cdot n + n^3 + 0.1 \cdot 2^n$	$O(2^n)$	6
$f_2(n) = \log(n) + 50000 \cdot n + n^5 + n^3 + n$	$O(n^5)$	5
$f_3(n) = 3n \cdot ld(n) + 110$	$O(n \cdot \log(n))$	4
$f_4(n) = 8 \cdot ld(n+4) + 500$	$O(\log n)$	2
$f_5(n) = (8 \cdot n^5 + 2) + 1/500 \cdot n!$	$O(n!)$	7
$f_6(n) = 8 \cdot n + 2 \cdot n + 500/n$	$O(n)$	3
$f_7(n) = 500 + 4!$	$O(1)$ [= konstant]	1

Rekursive Algorithmen (ALG 2/12)

Eine Funktion/Methode heisst rekursiv, wenn sie sich (direkt oder indirekt) selber aufruft:

Rekursionsbasis wie kann ein einfaches Problem der angegebenen Problemklasse gelöst werden?
Rekursionsvorschrift wie kann ein Problem auf ein einfacheres Problem zurückgeführt werden?

```
public long fak(int n)
{
    if((n == 1)||(n == 0)) {        ⇦ REKURSIONSBASIS
        return 1;
} else {
        return (n * fak(n-1));      ⇦ REKURSIONSVORSCHRIFT
    }
}
```

Speicherverwaltung (ALG 2/23)

Instanzvariable	⇨ Heap
Lokale Variablen / Parameter	⇨ Stack («Call Stack»)
Methoden	⇨ Stack («Call Stack») *ruft die eine Methode eine andere Methode auf, so werden zusätzlich die Informationen der zweiten Methode auf dem Call Stack abgelegt!*

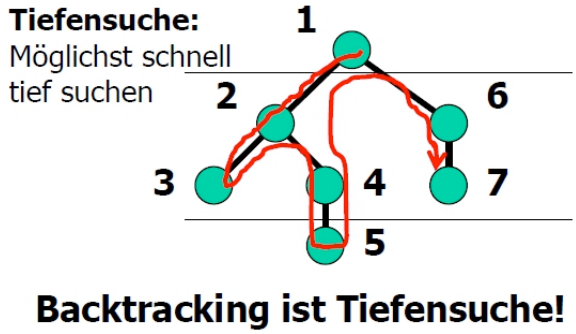
Rekursive Methodenaufrufe sind in der Regel speicherintensiver als iterative Verarbeitungsroutinen, in gewissen Fällen aber unumgänglich (z.B. Auslesen einer Verzeichnisstruktur, Backtracking-Algorithmen, …)

Backtracking (ALG 3/)

Backtracking ist – eine systematische Art der Suche
– in einem vorgegeben Lösungsraum

Wenn eine (Teil-) Lösung in eine Sackgasse führt, dann wird der letzte Schritt wieder rückgängig gemacht.

Merkmale:
- **Entspricht dem Konzept der TIEFENSUCHE**
- **gute Speicherkomplexität, da nur der aktuelle «Suchast» gespeichert werden muss**
- **wird mittels rekursivem Ansatz gelöst…**
- **… und hat daher meist eine nicht ideale Zeitkomplexität [exponentielle Laufzeit $O(k^n)$]**
- **benutzt HEURISTIK (Problemlösestrategien)**



Aufwand:

k Möglichkeiten in
n Stufen
⇨ $O(k^n)$

1, 2, 3, …, 7 ⇨ Zählschritte (kontinuierliche Inkrementierung)

Sortieren (ALG 4/)

Beim Sortieren wird zwischen stabilen und instabilen Sortier-Algorithmen unterschieden:

stabil ⇨ **Insertion Sort / Bubble Sort / Merge Sort**
Reihenfolge mehrerer Obj. mit gleichem Schlüssel vor und nach Sortieren bleibt **gleich**
8 5 **17_1** 23 **17_2** 10 21 ⇨ 5 8 10 **17_1 17_2** 21 23

instabil ⇨ **Shell Sort / Selection Sort / Quick Sort**
Reihenfolge mehrerer Obj. mit gleichem Schlüssel vor und nach Sortieren **ungleich**
8 5 **17_1** 23 **17_2** 10 21 ⇨ 5 8 10 **17_2 17_1** 21 23

Beispiel (anhand Selection Sort):
1 **17_1 17_2** 8 ⇨ 1 8 **17_2 17_1**

Sortieralgorithmen (ALG 4-6)

⇨ **de.wikipedia.org/wiki/Sortieralgorithmen** (⇨ Komplexitätsübersichten, etc.)

Einfache Sortieralgorithmen *(für eine kleine Anzahl zu sortierender Elemente)*:

Insertion Sort

von allen einfachen Alg. der effizienteste

«direktes Einfügen» (ist bei bereits [nahezu] sortierten Listen sehr effizient)
- zu sortierenden Array unterteilen («sortiert» / «unsortiert»)
- erster Schlüssel im unsortierten Teil wird durch Vergleich mit seinem «links sortierten Nachbarn» sortiert (⇨ Elemente mit grösserem Schlüssel werden nach rechts verschoben)

⇨ Sortieren endet nach n-1 Durchgängen

⇨ **Aufwand: best case O(n)** [für sort. Listen] **/ worst case $O(n^2)$ / average case $O(n^2)$**

Selection Sort

sortiert das kleinste Element immer an die 1. Stelle (in gewissen Anwendungsfällen wichtig)

«direktes Auswählen»
- zu sortierenden Array unterteilen («sortiert» / «unsortiert»)
- kleinsten Schlüssel im unsortierten Teil suchen und an die erste Stelle setzen
- nächsten Schlüssel im unsortierten Teil markieren (i_2)
- kleinsten Schlüssel im unsortierten Teil suchen und mit i_2 austauschen
- nächsten Schlüssel im unsortierten Teil markieren (i_3), Vorgehen wiederholen
- …

⇨ Sortieren endet nach n-1 Durchgängen

⇨ **Aufwand: O(n^2)** **Algorithmus ist effizient über grössere Distanzen!**

Bubble Sort **«direktes Austauschen»**

sortiert das grösste Element immer ganz nach rechts (in gewissen Anwendungsfällen wichtig)

- zu sortierenden Array unterteilen («sortiert» / «unsortiert»)
- im Array ganz links beginnen
- zwei benachbarte Schlüssel werden ausgetauscht, wenn nicht schon aufsteigend geordnet
- der grössere der beiden Schlüssel «blubbert» so immer weiter nach rechts
- der sortierte Teil des Arrays befindet sich so ganz rechts ⇨ der grösste Schlüssel steht im Array ganz rechts aussen

⇨ Sortieren endet nach n-1 Durchgängen

⇨ **Aufwand: eher ineffizient («Sesselrücken»), im allgemeinen Fall O(n^2)**

Shell Sort **«Verfeinerung des direkten Einfügens» (⇨ Spezielles Insertion Sort-Verfahren)**

- zu sortierenden Array unterteilen («sortiert» / «unsortiert»)
- Sortieren über mehrere Stufen: Schrittweiten (z.B. 1, 3, 7, 15, 31, «2n + 1») wählen
- zusammengehörende Schlüssel (⇨ Schrittweite!) mittels Insertion Sort sortieren

⇨ **Aufwand: im allgemeinen Fall O(n^2), bei Schrittweite 2n + 1 sogar O($n^{1,2}$)**
liegt damit zwischen einfachen [O(n^2)] und höheren [O(n·log(n))] Sortieralgorithmen

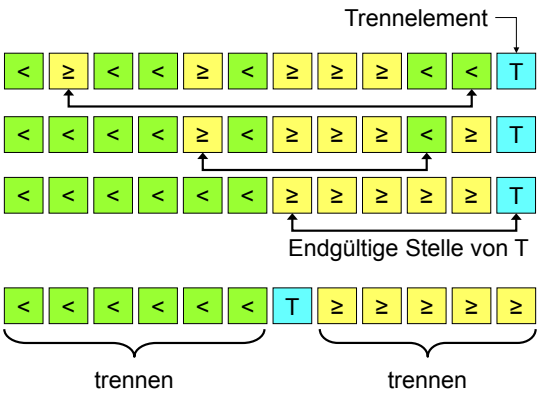
Höhere Sortieralgorithmen *(für eine grössere Anzahl zu sortierender Elemente)*:

Quick Sort **fortschreitende Zerlegung der Folge in kleinere Teilfolgen nach Bestimmen des Trennelem.**
(rekursiv)

für grosse unsortierte Listen sehr effizient. Braucht mehr Speicher als einfache Suchalgorithmen, da rekursiv umgesetzt.

- Bestimmung eines Trennelementes (bzw. Wert des Trennelementes) ⇨ «teile und herrsche»
- Alle Elemente rechts vom Trennelement sollen grösser oder gleich diesem sein.
- Alle Elemente links vom Trennelement sollen kleiner dem Trennelement sein.

⇨ 2 Möglichkeiten zur Bestimmung des Trennelementes:
- ⇨ letztes Element der Folge
- ⇨ «median-of-three»: Nimm das Element mit dem mittleren Wert von drei Elementen und tausche es mit dem letzten Element der Folge



ca. 10 – 50% schneller als Mergesort

⇨ **Aufwand:** **durchschnittlicher Zeitkomplexität von O(n · log(n))**
worst case O(n^2) (*)

⇨ **Speicherkomplexität: O(n)** (im worst-case Szenario)

(*) bei kleinen Teilfolgen oder bereits sortierten Listen ist der Quick Sort Algorithmus überhaupt nicht mehr effizient!

⇨ **Optimierung: Definition von M (= Minimum an zu sortierenden Elementen)**
Benchmark-Tests liefern beste Ergebnisse mit M = 25

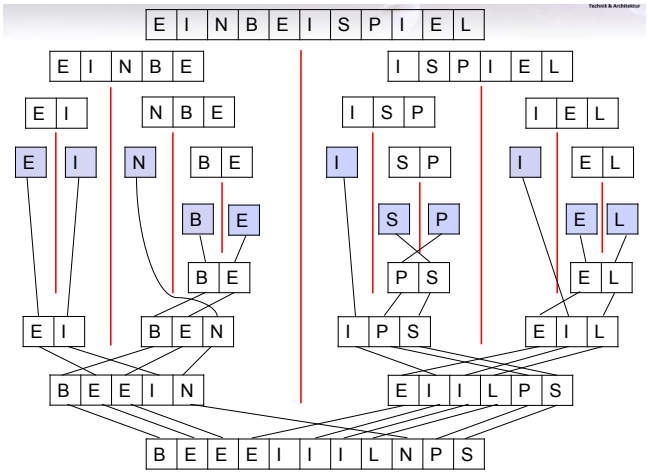
Quick-Insertion Sort **wenn Teilfelder kleiner als M sind ⇨ normale Sortierung mit Insertion Sort**
Quick-Insertion Sort ist also eine Kombination der beiden Sortieralgorithmen
⇨ damit lassen sich Laufzeitverbesserungen erzielen (> 20%!)

Merge Sort (rekursiv)

für Datensortierung in externem Speicher. Da garantierte Zeitkomplexität macht der Alg. Sinn, wenn nur eine begrenzte Zeit z.Vf. steht (z.B. Webanwendungen).

Zerlegen einer Folge in zwei «gleich grosse» Teilfolgen, wieder zerlegen, wieder zerlegen, …, Teilfolgen sortieren und zusammenfügen am Schluss

- jede Hälfte lässt sich ($\log_2 n$)-mal teilen (⇨ «Binärbaum»)
- Grundidee: teile und teile und teile und …, sortiere am Schluss und füge zusammen
  ⇨ «teile und herrsche»
- bei der Aufteilung in kleinere Teilfolgen ist die Grösse der Teilfolge jeweils ungef. gleich
- «Reissverschluss» am Schluss: letztes Sortieren zweier bereits sortierter Arrays



⇨ **Aufwand: garantierte Zeitkomplexität immer bei $O(n \cdot \log_2(n))$**
⇨ n Schlüsselvergleiche auf $\log_2 n$ Ebenen!

⇨ **Speicherkomplexität: $O(n)$**
⇨ weil Teilungen immer wieder gesp., verglichen und zusammengefügt werden müssen

Sortieren in Java (ALG 5)

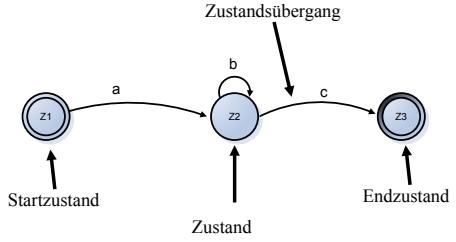
zwei Klassen enthalten Sortieralgorithmen:

- `java.util.Collections`
  ```
  import java.util.ArrayList;
  import java.util.Collections;
  ArrayList<String> strings = new ArrayList<String>();
  ```
  ⇨ **`Methode: Collections.sort(strings)`**
- `java.util.Arrays`
  ```
  import java.util.Arrays;
  int[] ints = new int[10];
  ```
  ⇨ **`Methode: Arrays.sort(ints)`**

Endliche Automaten (ALG 6)

Sprache:	besteht aus einer Menge von ⇨ Worten
Wort:	Element einer Sprache, bestehend aus einem oder mehreren Symbolen eines ⇨ Alphabets
Alphabet:	Menge aller Symbole, die zu einer ⇨ Sprache gehören.

Beispiel: endlicher Automat (akzeptiert folgende Worte: abc, ac, abbbbbbc, …):



Verwendung in der Informatik:
- zum Entscheiden, ob bestimmte Folgen von Zeichen Worte einer Sprache sind
- Spezifikation von Protokollen
- zum Beschreiben der Zustände und –übergänge von Maschinen und SW

Einfache Mustersuche in einem Text (ALG6)

Prinzip: das Muster wird Zeichen für Zeichen den Zeichen des Wortes entlang verschoben und überprüft, ob die Zeichenfolge des Musters im Wort vorkommt:

Wort: `abbabca`
Muster: `abc`

Vorgehen:
```
abbabca
abc
 abc
  abc
   abc
```

⇨ **Aufwand: Laufzeitkomplexität O(n · m)** (im schlechtesten Fall ist das Muster nicht vorhanden!)
«n-mal Schieben, m-mal Vergleichen»

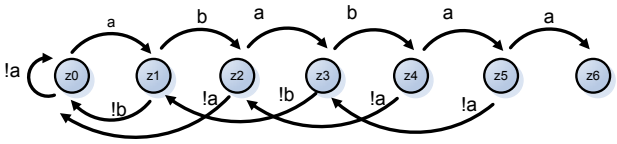
Verbesserte Suche mittels Musterautomaten (ALG6)

Entscheidend bei der Suche ist hierbei der Rand eines Wortes:
⇨ längste Buchstabenfolge, die kürzer ist als das Wort
⇨ und die sowohl am Anfang und Ende des Wortes auftritt
⇨ Anfang & Ende dürfen sich überschneiden!

Beispiel: Muster «ababaa»

Teilwort	Rand	Länge des Randes
a	Ø	0
ab	Ø	0
aba	a	1
abab	ab	2
ababa	aba	3

Musterautomat für "ababaa"



Text: **a b a a b b a b a b a b a a b a b a a**

Der Rand des Wortes gibt an, in welchen Zustand zurückgesprungen werden kann bei fehlender Übereinstimmung (z.B. von z4 zurück zu z2 falls im Muster kein «a» kommt ⇨ man braucht also mit der Musterüberprüfung nicht wieder «von vorne» zu beginnen)

⇨ **Aufwand: Laufzeitkomplexität O(n)** (abgeleitet aus O(n+m) mit folgender Begründung:)
falls m im Vergleich zu n sehr viel kleiner ist (vgl. Suche nach «Otello» in Shakespeare), so resultiert **O(n)**

KMP-Algorithmus [Knuth-Morris-Pratt] (ALG7)

⇨ **Aufwand: Laufzeitkomplexität O(n + m)**

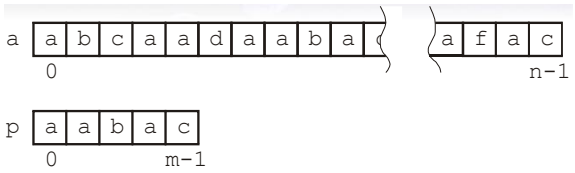
- Gemäss Länge des Suchpatterns (Länge m) ein «Sucharray» der Länge m generieren: Array `next`
- Danach Teilworte des Suchpatterns sowie deren Länge bestimmen
- `next`-Array mit der Länge der Teilworte füllen { -1 (markiert z0 bzw. Beginn), …, …, …, … }
  ⇨ dieses gibt an, an welcher (Such-)Stelle zu welchem Zustand gesprungen werden muss

Beispiel: Pattern „101011“ – Teilworte: „1“ ⇨ **0**; „10“ ⇨ **0**; „101“ ⇨ **1**; „1010“ ⇨ **2**; „10101“ ⇨ **3**
Bildung des **`next`-Arrays: { -1, 0, 0, 1, 2, 3 }**

Quicksearch (ALG7)

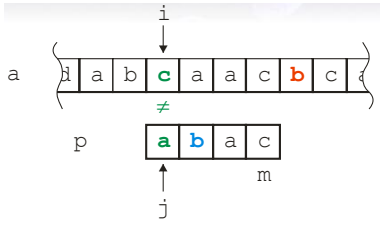
⇨ Aufwand: **best case Laufzeitkomplexität O(n / m)**
worst case Laufzeitkomplexität O(n · m)

Ermöglicht ein schnelles Durchsuchen einer Zeichenkette `a` nach einem Text-Pattern `p`:



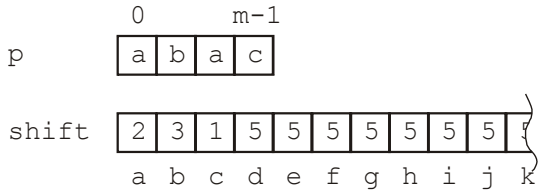
⇨ Vorteil: kommt ein Buchstabe von `a` nicht in `p` vor, kann mit dem Suchpattern `p` um maximal eine Musterlänge gesprungen werden (= Zeitersparnis!)

Das Pattern `p` wird zeichenweise mit der zu durchsuchenden Zeichenkette `a` verglichen:



bei Unstimmigkeit ⇨ Suche abbrechen und sofort das Zeichen unmittelbar nach dem Suchpattern vergleichen: kommt es in `p` vor?
⇨ ja: `p` an die Stelle schieben, wo Übereinstimmung ok («`shift`») und Index für Index vergleichen
⇨ nein: `p` um eine ganze Musterlänge verschieben («springen»)

im Voraus kann aufgrund des Suchpatterns das `shift`-Array gebildet werden
⇨ Frage für jeden Buchstaben: «wieviele Stellen muss `p` verschoben werden, bis der Bst. wieder vorkommt?»)




```
shift['a'] = 4;
shift['b'] = 3;
shift['a'] = 2;
shift['c'] = 1;
```

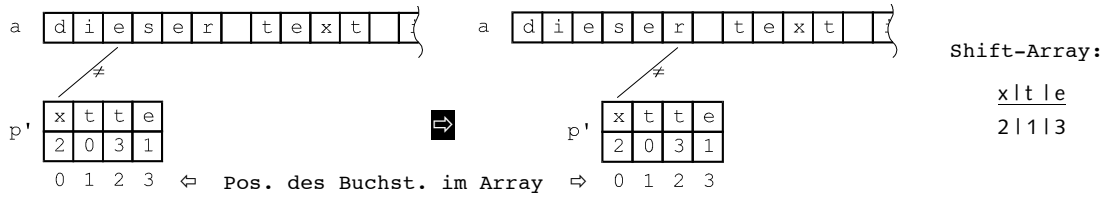
«a» überschreibt sich, da es 2x in p vorkommt // z.B. «d» kommt nicht vor ⇨ «Springen um 5»

Optimal Mismatch (ALG7)

⇨ Aufwand: **vergleichbar mit Quicksearch (i.d.R aber schneller, da «optimierte Suchfolge»)**

Optimal-Mismatch funktioniert von der Art und Weise her wie Quicksearch, besitzt aber die Eigenschaft, dass Zeichenvergleiche grundsätzlich in beliebiger Reihenfolge stattfinden können («stochasische Eigenschaft»)
⇨ Zeichen-Häufigkeit ist in den meisten Texten nicht gleich verteilt!
(z.B. dt. Sprache: Buchstabe e an erster Stelle mit 12%; Buchstabe x an letzter Stelle mit 0.002%)

⇨ selten auftretende Buchstaben vor den häufig auftretenden überprüfen, damit die Wahrscheinlichkeit für einen Mismatch bzw. für das Weiterspringen gross ist: Optimal-Mismatch!



Zusammenfassungen

Information Hiding (OOP 7/26-32)

Information hiding (= Datenkapselung) ⇨ Verbergen von Daten oder Infos vor dem Zugriff von aussen.

Kapselung ist ein **wichtiges Konzept der objektorientierten Programmierung**: **kontrollierter Zugriff auf Methoden bzw. Attribute von Klassen**.
Eine Klasse hat eine Schnittstelle, die darüber bestimmt, auf welche Weise mit der Klasse interagiert werden kann. Dies verhindert das Umgehen von Invarianten des Programms.

Vom Innenleben einer Klasse soll der Verwender – gemeint sind sowohl die Algorithmen, mit der die Klasse arbeitet, sowie der Programmierer, der diese entwickelt – möglichst wenig wissen müssen (Geheimnisprinzip).

Informationen über das „Was" (Funktionsweise) einer Klasse nach aussen sichtbar, nicht aber das „Wie" (die interne Repräsentation). **Dadurch wird eine Schnittstelle nach außen definiert und dokumentiert.**

Java kennt dafür die **vier** folgenden **Zugriffsmodifizierer** mit entsprechender Verwendung:

- public ⇨ möglichst wenig Methoden mit public (ausser mit Verhalten nach «aussen»)
- private ⇨ Attribute sollten IMMER private sein. Methoden immer auf «private» setzen, wenn sie nur von Methoden innerhalb der gleichen Klasse aufgerufen werden sollen!
- protected ⇨ analog private, zusätzlich aber auch für abgeleitete Methoden (Vererbung)
- keine Angabe ⇨ innerhalb des Pakets wie public, ausserhalb wie private

```
public class Person {
      private String vorname;            // gut; nicht sichtbar „von aussen"
      private String nachname;
      public void setVorname(String name){...}       // Mutator-Methode
      public void setNachname(String name){...}
      public String getVorname(){...}    // Accessor-Methode
      public String getNachname(){...}
      public void setName(String name)
      {
            String[] namen = name.split(" ");
            vorname = namen[0];
            nachname = namen[1];
      }
      public String getName()            // einfacher Zugriff für ext. Meth.
      {
            return (vorname + " " + nachname);
      }
}
```

Klassenvariablen und –methoden (OOP 7/34-35)

Definition der **Klassenvariable** (auch «Klassenattribut» genannt) mittels **Schlüsselwort `static`**:

```
static type class_variable;
private static float mwst = 7.6f;
```

Analog dazu verläuft die Definition der **Klassenmethode** mittels **Schlüsselwort `static`**:

```
public static long round(double a);
```

Konstanten (OOP 7/38)

Konstanten sind **Felder, deren Werte nicht verändert werden** dürfen / können. Per Konvention werden sie **GROSS GESCHRIEBEN** und mittels Schlüsselwort **`final`** definiert:

```
public final int MEINE_KONSTANTE = 4711;              // Konstante
public static final double PI = 3.141592653589793;    // Klassenkonstante
```

Klassenkonstanten werden auch hier bekannterweise mit dem **Schlüsselwort `static`** definiert.

Javadoc (OOP 8/21-25)

verarbeitet die Deklarationen und javadoc Kommentare von Java Source Code Dateien und erzeugt HTML:

```
/**
 *  Dieser Kommentar dient gleichzeitig auch dem Information Hiding-Konzept.
 *  WICHTIG: Klassenbeschreibungen sollten über dem Klassenkopf stehen.
 *
 *  @author (nicht nötig) Wer ist der Autor dieser Klasse
 *  @param myVar Beschreibung von myVar. IMMER ANGEBEN!
 *  @return Welchen Rückgabewert hat die Methode? IMMER ANGEBEN!
*/
```

Testen (OOP 8)

Testmethoden

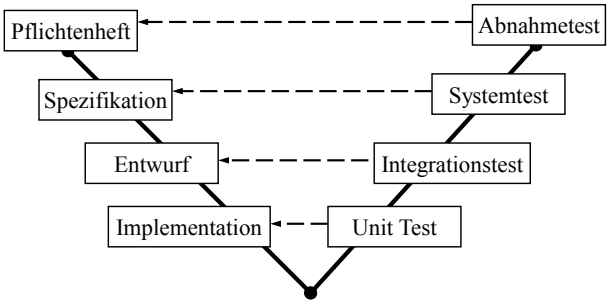
UNIT TESTING	1 Objekt (Klasse) / Einheit testen	
INTEGRATIONSTEST	mehrere Objekte (Klassen) / Einheiten im Zusammenspiel testen	
	2 Arten: a) TOP-DOWN	zuerst Test, dann Code entwickeln
	b) BOTTOM-UP	zuerst Code, dann Test entwickeln
SYSTEM TESTING	testet das ganze System bestehend aus versch. Obj./Kl. (Blackbox)	

Testen basiert auf Anforderungen und gehört zum Implemetationsprozess:
⇨ Welche Qualitätskriterien? Welche Werkzeuge? In welchen Projektphasen? Hard- und Software? Rollen und Verantwortlichkeiten? Meilensteine?

Anwendungen

Blackbox-Testing	Output-Test basierend auf dem Input; keine Beurteilung des Codes Frage nach: «erfüllt das Programm die Programmspezifikationen?» ⇨ versch. Testfälle: a) Normalfall b) zulässige Spezialfälle c) unzulässige Spezialfälle
Whitebox-Testing	Output-Test unter Analyse des Programmcodes – z.B. alle Variablen durchtesten – Wertebereiche der Variablen testen (overflow?) – sämtliche Programmpfade (d.h. alle Anweisungen, Zweige, Bedingungen, Schleifen) mind. 1x durchlaufen
Code Review	auch «Code Inspection» durch fremde Person (alle Methoden)
Walkthrough	fremde Person studiert und testet den Programmcode unter Anleitung des Programmieres (d.h. dieser definiert, was zu testen ist)
Debugger	«Step-by-Step»-Testen (Breakpoints!), Variableninhalte checken
Formale Verifikation	formale Korrektheit des Programmes beweisen (⇨ mathematisch)

Wichtig: Testanleitung und -vorgehen planen, verschriftlichen und die Testdokumentation im Abnahmeprotokoll IMMER festhalten.



Kopplung, Kohäsion und gutes Klassendesign (OOP9/13; 15)

Kopplung: gibt Auskunft über das Mass der Verknüpfung der Klasse mit anderen Klassen. Idealerweise wird eine verknüpfte Klasse so nicht verändert, sobald Änderungen an dieser Klasse gemacht werden.
⇨ Kopplung = Mass des **Zusammenhangs zwischen Klassen**

Kohäsion: bezieht sich auf die Anzahl und Unterscheidung der Aufgaben, welche in dieser Einheit enthalten sind (d.h. für welche Methoden die Klasse verantwortlich ist).
⇨ Kohäsion = Mass des **inneren Zusammenhalts einer Klasse**

Code-Duplikation: ist Anzeichen von schlechtem Code (schlechte Kohäsion) und kann Inkonsistenzen hervorrufen. Dies vergrössert den Unterhaltsaufwand für einen Programmierer und erhöht die Fehleranfälligkeit.

localizing change: **Änderungen am Code der einen Klasse soll so wenig Überarbeitungen wie möglich im Code einer anderen Klasse hervorrufen.**

Ziel des Programmierens soll sein, ein ideales Mass zwischen möglichst grosser Kohäsion und loser Kopplung umzusetzen (resultierende Vorteile: READABILITY und REUSE).
schlechtes Beispiel: implizite Kopplung. Massgebend sind nicht offensichtliche Abhängigkeiten zwischen Klassen, worauf der Quelltext keine Hinweise gibt. «Fehlverhalten» von Klassen bzw. Methoden werden nicht direkt sichtbar; z.B. Kommandos verändern sich in der einen Klasse, trotzdem funktioniert die andere…

Folgen von gutem Klassendesign:

- Bessere Verständlichkeit/Lesbarkeit
- Bessere Testbarkeit
- Bessere Wiederverwendbarkeit
- Auswirkungen von Fehlern halten sich in Grenzen
- Änderungen sind einfacher möglich

Folgen von schlechtem Klassendesign

- Langsame Entwicklung
- Schlechte Entwicklung
- Fehlerhafte Entwicklung
- Abgebrochene Entwicklung
- Nachträgliche Änderungen schwer umsetzbar

Hauptdisziplinen bei der Softwareentwicklung (OOP9/6)

1. Kundenanforderungen erfassen und beschreiben
2. System-Spezifikation: Anforderungen an die Software bestimmen / Skizzen / Datenmodelle erstellen
3. Realisierung: Klassen-Design (Entwurf) ⇨ Programmierung ⇨ Test ⇨ Dokumentation des Codes
4. Test: Software schrittweise zusammenführen / integrieren und testen
5. Deployment: Software verteilen und einsetzen

Refactoring (OOP9/34-36)

Refactoring beschreibt das Anpassen des Designs/Codes an neue Anforderungen und Gegebenheiten. Dies bedingt häufig ein Aufspalten von Klassen und Methoden in weitere Klassen bzw. Methoden. (Auch das Umgekehrte ist möglich, allerdings eher selten der Fall). Vorsicht: «Fehleranfälligkeit des Codes» durch Um- bzw. Neuschreiben/Ergänzen nimmt zu!

Vorgehensweise:

1. Testspezifikation schreiben und Code testen
2. Code ohne funktionale Erweiterungen re-designen und erneut testen
3. Code mit zusätzlichen funktionalen Erweiterungen versehen und erneut testen

Vererbung und Polymorphismus (OOP10/6-19)

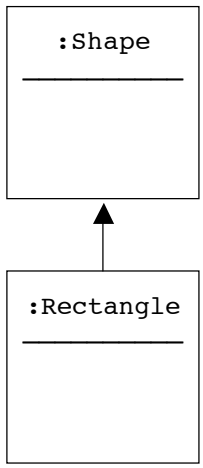
Vererbung (engl. inheritance) und Polymorphismus (Vielgestaltigkeit) sind zentrale Konzepte in der objektorientierten Programmierung:

- eine Klasse als Erweiterung ihrer übergeordneten Klasse (Vererbung)
  abgeleitete Klasse besitzt («erbt») Methoden und Attribute der Basisklasse; sie kann auf diese zugreifen (falls nicht durch Zugriffsmodifizierer eingeschränkt), d.h. dessen Methoden ansprechen und Attribute lesen ⇨ Vergrösserung des «Funktionsspektrums»
- Polymorphismus
  Variablen eines Klassentyps können sowohl Objekte des deklarierten Typs aufnehmen wie auch Objekte aller abgeleiteten Subtypen

Beispiel:

```
/** Erweitert die Klasse «Rectangle» als Unterklasse der Klasse «Shape»
 *  und zeichnet ein Rechteck */
public class Rectangle extends Shape
{
      private int width, height;
      /** Rechteck an Position (0/0)
       * mit 2 Pixel Länge/Breite */
      public Rectangle()
      {
            super(0, 0);
            width = 2; height = 2;
      }

      ...
}
```



Schlüsselwörter

– **`extends`**
⇨ erweitert die Basisklasse (Oberklasse) um die folgende Unterklasse
dabei erbt die Unterklasse alle Attribute und Methoden der Basisklasse
⇨ **`class Unterklasse extends Basisklasse { ... }`**

– **`super(...)`**
⇨ wird in der Unterklasse verwendet zur Erzeugung des Objekts der Basisklasse (= «Konstruktoraufruf» der Basisklasse im Konstruktor der Unterklasse)
⇨ **erste Anweisung im Konstruktor der Unterklasse immer: `super(...)`!**

Vorteile der Vererbung:

- Vermeidung von Code-Duplikationen
- Vereinfachung des Code-Unterhalts
- unterstüzt einfache Erweiterungen bestehender Klassen

⇨ **Verstärkung der Kohäsion**

Nachteile der Vererbung:

- schafft Abhängigkeiten zwischen Basis- und Unterklassen

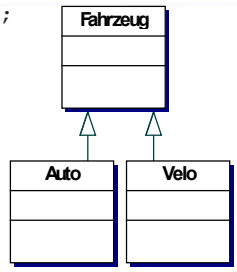
⇨ **Erhöhung der Kopplung**

Klassenhierarchie und Subtyping (OOP10/22-…):

Eine Basisklasse kann mehrere Unterklassen haben.
Jede Unterklasse kann wiederum mehrere Unterklassen haben.
Dabei können Variablen eines Klassentyps sowohl Objekte des deklarierten Typs aufnehmen wie auch Objekte aller abgeleiteten Subtypen (Polymorphismus).

Variablen referenzieren ein Objekt des deklarierten Typs (❶) oder auch Objekte eines Subtyps dessen (❷).

```
❶ Fahrzeug f1 = new Fahrzeug();
❷ Fahrzeug f2 = new Auto();
  Fahrzeug f3 = new Velo();
```



Vergleiche:
❷ Ein Auto/Velo ist ein Fahrzeug.
ABER: Ein Fahrzeug ist kein Auto/Velo!

falsch:

```
Auto a2 = new Fahrzeug();
Auto a3 = new Velo();
```

Casting-Möglichkeiten:

```
Fahrzeug fahrzeug1, fahrzeug2;  Auto auto;  Velo velo;

auto = new Auto();
fahrzeug1 = auto;          // korrekt (implizites Casting)
auto = (Auto)fahrzeug1;    // korrekt (explizites Casting)

velo = (Velo)auto;         // Compiler-Fehler! (Subklasse → Subklasse)

fahrzeug2 = auto;
velo = (Velo)fahrzeug2;    // Laufzeit-Fehler!
```

OOP5

Alle Schleifen-Anweisungen sind gleich mächtig.
`break` beendet die Schleifenausführung sofort, `continue` springt an das Ende des Schleifenrumpfs.

for-Schleife

⇨ wenn **evtl. 0 Durchgänge** erwartet werden **oder die Anzahl bereits bekannt** ist **oder der Index** zur Verarbeitung **gebraucht wird**:

```
for(initialization; condition; action)
{
     doSomething;
}
```

```
for(int i = 0; i < myArray.length; i++ )
{
     System.out.println(myArray[i]);
}

for(int i = 0; i < myArrayList.size(); i++ )
{
     System.out.println(myArrayList.get(i));
}
```

for-each-Schleife

⇨ wenn in einer Collection **alle Elemente traversiert** werden sollen:

```
for(Type element : collection)
{
     doSomething;
}
```

```
for(Type myElement : myCollectionItem)
{
     System.out.println(myElement);
}
```

While-Schleife

⇨ **mind. 0 Durchgänge oder** ∞ (⇦ dann setze condition *true*):

```
while(loop condition)
{
     doSomething;
}
```

```
while(index < myArray.length)
{
     System.out.println(myArray.get(index));
}
```

Allenfalls muss ein Inkrementer gesetzt werden, wenn die Schritte gezählt werden sollten.

do-while-Schleife

⇨ **Anzahl Schleifendurchläufe unbekannt (aber mindestens 1x)** oder bei **Programmüberprüfung:**

```
do {
     statement(s)
} while (expression);
```

```
do {
     int dice=(int) (Math.random()*6+1);
} while (dice!=6);
```

Wie bei der while-Schleife muss allenfalls ein Inkrementer gesetzt werden, wenn die Schritte gezählt werden sollten.

Array:

Kann **Werte eines bestimmten elementaren Datentyps oder Objekte (Referenzen) eines bestimmten Klassentyps aufnehmen**. Seine **Länge (Funktionsaufruf: `.length`) wird unveränderlich festgelegt**!

```
int[] myArr = new int[3];            // Länge von 3 Elementen bleibt fix!
```

Grundsätzliche Deklarationsmöglichkeiten von Arrays:

```
// Deklaration inklusive Initialisierung:
int[] myArr = {0, 2, 4};

// mit Wertzuweisungen:
myArr[0] = 0;                        // Erstes Array-Element mit Index 0!
myArr[1] = 2;
myArr[2] = 4;
```

Arrays können auch Werte mit **Datentypen eines Objekts** beinhalten:

```
ObjectType[] myArr = new ObjectType[3];
myArr[0] = new ObjectType("...");

int arrLength = myArr.length;        // liefert die Anzahl Elemente des Arrays
```

ArrayList:

```
import java.util.ArrayList;
```

Jedes Exemplar der Klasse ArrayList **vertritt ein Array mit variabler Länge**. Der Zugriff auf die Elemente erfolgt über **Indizes**. Da **in einer ArrayList jedes Exemplar einer von Object abgeleiteten Klasse Platz findet**, ist eine ArrayList nicht auf bestimmte Datentypen fixiert, doch Generics spezifizieren diese Typen genauer.

http://openbook.galileodesign.de/javainsel5/javainsel11_002.htm#Rxx747java110020400038E1F0321CB

```
ArrayList<myObjectType> myList = new ArrayList<myObjectType>();
myList.add(AddSomethingOfMyObjectType);
myList.add(AddSomethingOtherOfMyObjectType);
```

Die ArrayList kann mit einem Iterator oder über Schleifen ausgelesen werden.
Zudem stehen folgende Methoden zur Verfügung:

- `add()` `// myList.add(new Balloon("red"));`
- `size()` `// int sizeOfMyList = myList.size();`

OOP6

Vergleichen von Strings:

Achtung **Unterschied: Vergleich von Stringreferenzen**…

```
String input = reader.get();
if (input == "bye")                  // Referenzvergleich (gl. Speicheradr.?)
{ ...
```

… **oder Stringinhalten**

```
String input = reader.get();
if (input.equals("bye"))             // immer mit .equals(...) vergleichen!!!
{ ...
```

Java Application Programmer's Interface (API)

beschreibt nur das **Interface einer Klasse** («WAS») und enthält die Beschreibung aller Bibliotheks-Klassen:

- Name der Klasse
- generelle Beschreibung der Klasse
- Liste aller Konstruktoren, Methoden und öffentlichen Felder (sofern vorhanden)
- Beschreibung des Zwecks jeden Konstruktors / jeder Methode / jeden öffentlichen Feldes (sofern vorhanden)

Method Summary	
`static long`	**`addierenRekursiv`**`(int a, int b)` multiplies two factors by adding them recursively.

...

Method Detail

addierenRekursiv

```
public static long addierenRekursiv(int a,
                                     int b)
```

Multiplies two factors by adding them recursively.

Parameters:

a - first factor (integer value)
b - second factor (integer value)

Returns:

the product as a long value

Code-Fragment:

```
/**
 * Multiplies two factors by adding them recursively.
 * @param a first factor (integer value)
 * @param b second factor (integer value)
 * @return the product as a long value
 */
public static long addierenRekursiv(int a, int b) { ... }
```

Iteratoren:

```
import java.util.Iterator;
```

Jeder Iterator stellt Funktionen namens **`next()`**, **`hasNext()`** sowie eine optionale Funktion namens **`remove()`** zur Verfügung. **Iteratoren werden üblicherweise von einer Funktion namens `iterator()` generiert, welche von der dementsprechenden Containerklasse zur Verfügung gestellt wird.** Ein Iterator gibt als Initialisierungwert einen speziellen Wert zurück, der den Anfang markiert. Aus diesem Grund ist es nötig nach der Initialisierung **`next()`** auszuführen, womit das erste Element zurückgegeben wird. Die **`hasNext()`**-Funktion wird dazu benutzt, um herauszufinden, wann das letzte Element zurückgegeben wird.

Das folgende Beispiel zeigt eine simple Verwendung von Iteratoren in Java:

```
Iterator<myObjectType> itr = myObject.iterator();
while(itr.hasNext()) {
      System.out.println(itr.next());
}
```

Quellenangabe: http://de.wikipedia.org/wiki/Iterator

Listeniteratoren:

```
import java.util.ListIterator;
```

Listeniteratoren ermöglichen – zusätzlich zu den Methoden, die ein Iterator bereits zur Verfügung stellt – zudem einen Rückgriff auf das vorhergehende Listeniteratorobjekt mittels der Funktionen **`previous()`** und **`hasPrevious()`** (kann also auch rückwärts ein Objekt traversieren), siehe Beispiel:

```
ListIterator<myListType> listItr = myList.listIterator(myList.size());
while(itr.hasPrevious()) {
      System.out.println(itr.Previous());      // beginnt so beim letzten El.
}
```

StringTokenizer:

```
import java.util.StringTokenizer;
```

Die StringTokenizer class ermöglicht, eine Zeichenkette bequem in einzelne Teile («Tokens») zu zerlegen:

```
ArrayList<String> myString = new ArrayList();        // only for demo purpose
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens()) {
      myString.add(st.nextToken());
}
```

OOP7

Map ⇨ Anwendung HashMap:

```
import java.util.(Hash)Map;
```

In einer Map werden Objektpaare bestehend aus einem Schlüssel-Objekt (ein-eindeutig!) und einem Wert-Objekt abgespeichert. Ein Wert-Objekt kann mit verschiedenen Schlüssel-Objekten mehrfach in einer Map abgespeichert werden. **Schlüssel-Objekte in einer Map sollen nach dem Einfügen nicht verändert werden.**

Eine `HashMap` ist eine Implementation einer «Map»:

```
private HashMap<myObjectTypeKey, myObjectTypeValue> myMap;
myMap = new HashMap<myObjectTypeKey, myObjectTypeValue>();
```

Die `Map` kann nicht direkt traversiert werden (zusätzlich sind Methoden `keySet()` und `values()` nötig). Zudem stehen folgende Methoden zur Verfügung:

- `put(key, value)` — `key und value müssen dem korrekten Objekttyp entsprechen!`
- `get(key)` — `key muss dem Wertepaar entsprechen!`
- `size()`
- `containsKey()`
- `containsValue()`
- `isEmpty()`
- `remove()`
- **`keySet()`** — **erzeugt eine Set aller enthaltenen Schlüssel**
- **`values()`** — **erzeugt eine Collection aller enthaltenen Werte**

```
private HashMap<String, String> myMap = new HashMap<String, String>();
myMap.put("012 345 67 89", "Hubert Burri");
int sizeOfMyMap = myMap.size();
```

Um eine Map zu traversieren muss entweder eine Set der Schlüssel-Objekte oder eine Collection der Wert-Objekte erstellt werden (diese können dann traversiert werden).

Set ⇨ Anwendung HashSet:

```
import java.util.Set;
```

Eine Menge ist eine (erst einmal) **ungeordnete Sammlung von Elementen**. **Jedes Element darf nur einmal vorkommen. Objekte, die in einem Set enthalten sind, sollten nicht mehr manuell mutiert werden!**

Beispiel für die Verwendung von `Set`:

```
private Set<myObjectType> mySet;
mySet = myHashMap.keySet();   // liest alle key-Elemente aus einer HashMap)
```

Typische Verwendung: Neu-Erstellung einer `HashSet` (Implementation einer «Set»):

```
private HashSet<myObjectType> mySet;
mySet = new HashSet<myObjectType>();
```

Die `Set` kann mit den bekannten Mitteln traversiert werden. **Sie enthält keine «get»-Methode (muss durchiteriert werden, falls Werte ausgelesen werden sollen)!** Zudem stehen folgende Methoden zur Verfügung:

- `add()`
- `size()`
- `iterator()`
- `contains()`
- `isEmpty()`
- `remove()`

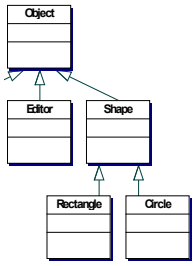
Externer Funktionsaufruf mit der main-Methode

Beim Start der Klasse wird die `main`-Methode zuerst aufgerufen. Dazu muss sie aber in der Klasse definiertsein:

```
public static void main(String[] args)
{
      char[] feld = {'A', 'B', 'C'};
      permute(feld, feld.length-1);       // Methode «permute» wird aufgerufen
}
```

OOP10

Die Klasse `Object`



⇨ **alle (durch den Programmierer festgelegten) Klassen sind Unterklassen der Basisklasse `Object`.**

Wichtige Methoden:

- `equals(Object obj)` — `Testet, ob zwei Objekte gleich sind`
- `toString()` — `erzeugt eine String-Repräsentation des Obj.`

Wrapper-Klassen

Zu jedem primitiven Datentyp in Java gibt es eine korrespondierende Wrapper-Klasse, welche die primitive Variable in einer objektorientierten Hülle kapselt («Behandlung wie ein Objekt») und eine Reihe von Methoden zum Zugriff auf die Variable zur Verfügung stellt.

Dies wird z.B. bei ArrayLists gebraucht (vgl. Speicherung von float-Werten in ArrayLists):

```
float f = 4.23;                          // float Variable f
Float floatObject;                       // Klasse Float
floatObject = new Float(f);              // in Float Objekt einpacken
```

```
ArrayList floats = new ArrayList();       // ArrayList ohne Generics erstellen
floats.add(floatObject);                  // einfügen

floatObject = (Float)floats.get(0);       // auslesen
f = floatObject.floatValue();             // konvertieren des Object in Zahl


/**
 * ODER mittels ArrayList MIT Generics:
*/

ArrayList<Float> floats = new ArrayList<Float>();
floats.add(f);                            // einfügen

f = floats.get(0);                        // auslesen
```