

OBJEKTORIENTIERTE PROGRAMMIERUNG

Kurzbeschreibung

Das der objektorientierten Programmierung zugrunde liegende Klassenkonzept sowie wichtige Klassen dynamischer Datenstrukturen und das Vererbungsprinzip werden behandelt.

Schlüsselwörter

Klasse, Geheimnisprinzip, Konstruktor, dynamischer Datentyp, Liste, Vererbung, Polymorphie, dynamische Bindung, abstrakte Klasse, *Framework*, *Interface*, Schnittstellenklasse

Lernziele

1. Das Klassenkonzept als Basis der Objektorientierung wird konzeptionell und praktisch durchdrungen.
2. Das auf dem Klassenkonzept aufsetzende Vererbungsprinzip und die dynamische Bindung werden verstanden.
3. Programme, die das Klassenkonzept und weiterführende objektorientierte Prinzipien nutzen, können geschrieben werden.

Hauptquellen

- Hanspeter Mössenböck: Sprechen Sie Java? – Eine Einführung in das systematische Programmieren, dpunkt.verlag 2003.

Inhaltsverzeichnis

1	KLASSEN.....	3
1.1	Beispielklasse Date	4
1.2	Typkompatibilität und Vergleich von Objekten	4
1.3	Objekte als Rückgabewerte	5
1.4	Klassen und Arrays.....	6
2	OBJEKTORIENTIERUNG.....	8
2.1	Beispiel: Klasse Fraction.....	8
2.2	Konstruktoren	11
2.3	Klassenattribute und Klassenmethoden versus Objektattribute und Objektmethoden .	12
2.4	Stapel als Beispiel einer Klasse	13
3	DYNAMISCHE DATENSTRUKTUREN.....	15
3.1	Listen	17
3.2	Suchen in einer unsortierten Liste	18
3.3	Einfügen in eine sortierte Liste.....	19
3.4	Stapel als verkettete Liste	20
4	VERERBUNG	22
4.1	Modellieren und Programmieren von Vererbungsbeziehungen	23
4.2	Polymorphie und Kompatibilität	25
4.3	Dynamische Bindung.....	26
4.4	Abstrakte Klassen und Interfaces	28
	VERZEICHNISSE	34
	Abkürzungen und Glossar	34
	Index	35

Informationen und Interaktionen	35
Literatur	36

- **KLASSEN**
 - Deklaration und Verwendung, Klassen und Arrays, Array-Stack
- **OBJEKTORIENTIERUNG**
 - Konstruktoren, Klasselemente und Objektelemente
- **DYNAMISCHE DATENSTRUKTUREN**
 - Listen (unsortiert und sortiert), Listen-Stack
- **VERERBUNG**
 - Polymorphie, Dynamische Bindung, abstrakte Klassen, Interfaces, Frameworks

Information 1: OBJEKTORIENTIERTE PROGRAMMIERUNG

1 KLASSEN

Die objektorientierte Programmierung bildet die Basis der heute in der Praxis verwendeten Softwaretechnik [Mö03].

Die der Objektorientierung zugrunde liegenden Techniken setzen auf den in der Kurseinheit IMPERATIVE PROGRAMMIERUNG [C&M-IP] beschriebenen Konzepten auf. Zu den zentralen Konzepten der Objektorientierung gehören die Klassen und die daraus hervorgehenden Objekte. Die Objekte sind dabei Gegenstände der realen oder der künstlichen Welt, wie in der Kurseinheit GRUNDBEGRIFFE DER INFORMATIK [C&M-GI] im Zusammenhang mit der Modellierung dargestellt wurde.

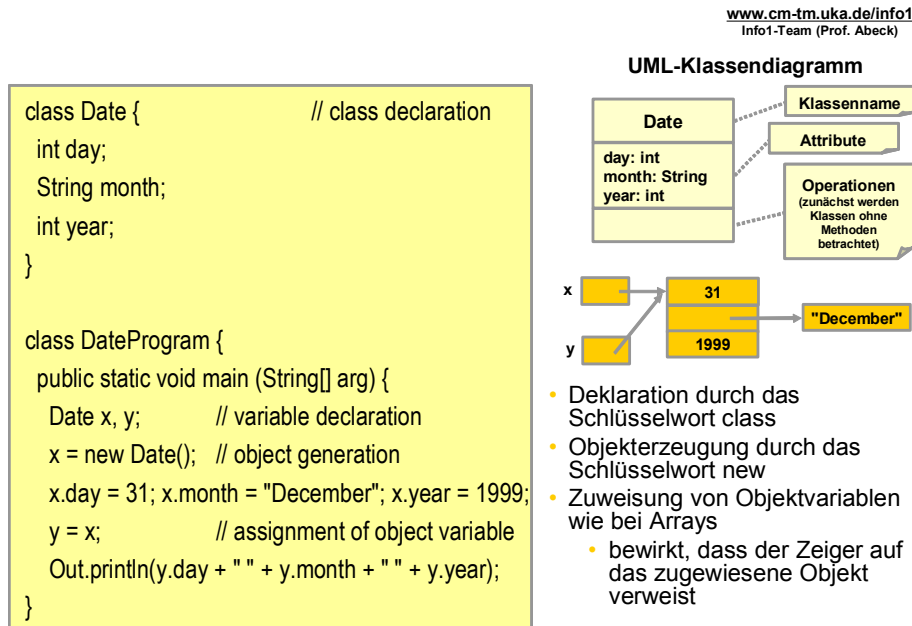
- Klassen sind selbst definierte Datentypen, durch die mehrere Elemente zusammengefasst werden können
- Die Elemente können aus verschiedenartigen Elementen aufgebaut sein
 - wesentlicher Unterschied zu Arrays
- Beispiel
 - es soll eine Klasse aufgebaut werden, die ein Datum beschreibt
 - Elemente sollen sein
 - Tag beschrieben als ganzzahliger Wert
 - Monat beschrieben als Zeichenkette
 - Jahr beschrieben als ganzzahliger Wert

Information 2: KLASSEN - Wesentliche Eigenschaften und Beispiel

Durch Klassen werden verschiedene bestehende Datentypen zu einem neuen, übergeordneten Datentyp zusammengefasst. Hinsichtlich dieses Aspekts haben Klassen eine Ähnlichkeit mit den in [C&M-IP] eingeführten Arrays. Ein wesentlicher Unterschied zu Arrays ist, dass Klassen aus verschiedenartigen Elementen aufgebaut sein können.

1.1 Beispielklasse Date

Wie im Beispiel in Information 2 ausgeführt ist, kann eine Datums-Klasse beispielsweise aus zwei ganzzahligen Werten, die den Tag und das Jahr beschreiben, sowie einer den Monat angehenden Zeichenkette zusammengesetzt sein.



Information 3: Deklaration und Verwendung von Klassen

Das Schlüsselwort class wurde bereits früher im Zusammenhang mit der Grundstruktur eines Java-Programms eingeführt (siehe Kurseinheit PROGRAMMIERGRUNDLAGEN [C&M-PG]).

Wie das UML-Klassendiagramm in Information 3 verdeutlicht, werden zunächst Klassen betrachtet, die nur Attribute und keine Methoden besitzen. Die deklarierte Klasse Date besitzt die drei zuvor eingeführten Attribute day, month und year.

Wie bereits im Zusammenhang mit den Arrays kennen gelernt, werden durch die Deklaration

```
Date x, y;
```

lediglich Objektvariablen angelegt, die auf kein Objekt zeigen, also den Nullpointer null als Wert beinhalten. Die Erzeugung eines Date-Objekts und dessen Zuweisung zur Objektvariablen x erfolgt durch

```
x = new Date();
```

Nach der Belegung dieses Objekts mit einem konkreten Datum (31 December 1999) erfolgt mittels

```
y = x;
```

die Zuweisung des Wertes von x zur Objektvariablen y und der Zugriff über y auf die Attribute des Objekts im Rahmen des Methodenaufrufs

```
Out.println(y.day + " " + y.month + " " + y.year);
```

Hinsichtlich der Zuweisung von Objektvariablen stellt sich unmittelbar die Frage, welche Eigenschaften die Objektvariablen erfüllen müssen, um typkompatibel zu sein.

1.2 Typkompatibilität und Vergleich von Objekten

Im Folgenden werden Zuweisungen und Vergleiche von Objekten näher betrachtet.

- In Java ist eine Zuweisung von Objektvariablen nur zulässig, wenn die Objektvariablen den gleichen Typnamen haben
 - diese Art der Typkompatibilität heißt Namensäquivalenz
- Eine andere Art von Typkompatibilität ist die Strukturäquivalenz
 - zwei Typen sind gleich, wenn sie die selbe Struktur haben
 - ist in Java wegen geforderter Namensäquivalenz nicht zulässig
- Namensäquivalenz lässt sich durch einen Compiler einfacher überprüfen als Strukturäquivalenz

Information 4: Typkompatibilität

Die Sprache Java unterstützt hierbei mit der in Information 4 beschriebenen Namensäquivalenz eine im Vergleich zur Strukturäquivalenz restriktivere Typkompatibilität.

- Folgende zwei Arten von Vergleichen lassen sich bei Objekten unterscheiden
 - (1) Vergleich auf der Ebene der Objektvariablen (Zeigervergleich)

```
Date x, y; if (x==y) ....; if (x!=y) ...
```

- (2) Vergleich auf der Ebene der Attribute

```
static boolean isDateEqual (Date x, Date y) {
    return x.day == y.day && x.month.equals(y.month) && x.year == y.year;
}
...
if (isDateEqual(x,y)) ...
```

- Richtig oder falsch?

	richtig	falsch
Aus der Objektegleichheit folgt die Attributgleichheit	<input type="checkbox"/>	<input type="checkbox"/>
Aus der Attributgleichheit folgt die Objektegleichheit	<input type="checkbox"/>	<input type="checkbox"/>

Interaktion 1: Objektegleichheit und Attributgleichheit

Wie Interaktion 1 verdeutlicht, können Objekte auf der Objektvariablen-Ebene und damit auf der Zeigerebene oder auch auf der Attribut-Ebene miteinander verglichen werden. Offensichtlich haben zwei Objektvariablen, die auf dasselbe Objekt zeigen (also $x=y$) auch übereinstimmende Attributwerte (also $\text{isDateEqual}(x, y)$). Übereinstimmende Attributwerte lassen aber keine Aussage darüber zu, ob es sich um dasselbe oder zwei verschiedene Objekte handelt.

1.3 Objekte als Rückgabewerte

Bislang wurden ausschließlich als Methoden realisierte Funktionen betrachtet, die nur einen einfachen Wert zurückgeliefert haben. Da der Rückgabewert auch ein einzelnes Objekt sein kann, lassen sich hierüber beliebige, in dem Objekt zusammengefasste Ergebniswerte durch eine Methode zurückliefern.

- Eine Methode (Funktion) kann ein Objekt als Ergebnis zurückliefern
 - hierdurch kann eine Funktion mehrere Werte zurückgeben

```

class Time { int h, m, s;}           // hours, minutes, seconds

class ConvertTimeProgram {
  static Time convert (int seconds) { // convert seconds to time
    Time t = new Time();           // generate time object
    t.h = seconds / 3600;
    t.m = (seconds % 3600) / 60;    // rest from (seconds / 3600) divided by seconds per minute
    t.s = seconds % 60;
    return t;
  }

  public static void main (String[] arg) {
    int seconds; Time t;
    Out.print("enter seconds: "); seconds = In.readInt();
    while (In.done()) {
      t = convert(seconds); Out.println(t.h + ":" + t.m + ":" + t.s);
      Out.print("enter seconds: "); seconds = In.readInt();
    } /* while */ } /* main */ } /* class */

```

Information 5: Objekt als Rückgabewert einer Methode

Am Beispiel der im Java-Programm von Information 5 enthaltenen Methode `convert()` zu der Klasse `ConvertTimeProgram` wird deutlich, dass innerhalb dieser Methode drei `int`-Werte (`t.h`, `t.m`, `t.s`) berechnet werden, die mittels `return t` in einem `Time`-Objekt zusammengefasst zurückgeliefert werden.

1.4 Klassen und Arrays

Die vorhergehenden Ausführungen haben gezeigt, dass zwischen Klassen und Arrays gewisse Zusammenhänge bestehen, die im Folgenden näher untersucht werden.

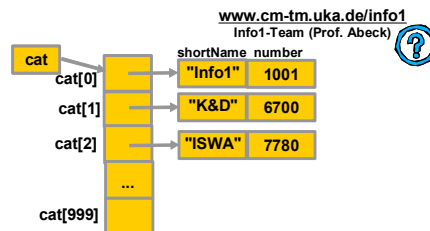
- Gemeinsamkeiten von Klassen und Arrays
 - bestehen aus mehreren Elementen
 - werden durch Zeiger referenziert
- Unterschiede
 - Arrays bestehen aus gleichartigen Elementen, während Klassen aus verschiedenartigen Elementen bestehen
 - Array-Elemente haben keinen Namen, Klassen-Elemente (Attribute) haben einen Namen
 - Anzahl der Elemente wird zu unterschiedlichen Zeitpunkten festgelegt
 - Array: bei der Erzeugung des Array-Objekts
 - Klasse: bei der Deklaration

Information 6: Klassen und Arrays

Information 6 stellt die Gemeinsamkeiten und Unterschiede der beiden Datenstrukturen im Überblick dar. Die Unterschiedlichkeit der Elemente von Arrays und Klassen im Hinblick auf Art, Name und Festlegung der Anzahl verdeutlicht, dass die beiden Datenstrukturen trotz der offensichtlichen Gemeinsamkeiten ganz unterschiedliche Zielsetzungen verfolgen.

Häufig werden beide Datenstrukturen kombiniert in einem Programm verwendet, wie am Beispiel der Programmierung eines Kurskatalogs in Interaktion 2 verdeutlicht wird.

- Beispiel Kurskatalog: Array von Objekten einer Kurs-Klasse, die hier durch folgende Attribute beschrieben sind:
 - Kurzbezeichnung vom Typ String
 - eindeutige Nummer vom Typ int



```
class Course {String shortName, int number;} // declaration of class course

class CourseCatProgram {
    static Course[] cat; // catalog as array of courses
    static int courseCount = 0; // counts the number of courses in the catalog

    public static void main (String[] arg) {
        int i; cat = new Course[1000]; // generation of array of 1000 pointers to course objects
        cat[courseCount] = new Course(); // generation of one new course object
        cat[courseCount].shortName = "Info1"; cat[courseCount].number = 1001; courseCount++;
        // add courses describing K&D and ISWA
        for (i = 0; i < courseCount; i++) Out.println(cat[i].shortName + " " + cat[i].number);
    } /* main */ } /* class */
```

Interaktion 2: Kombination von Klassen und Arrays

Unter einem Kurs wird hierbei eine Lehrveranstaltung, wie z.B. die INFORMATIK-I-Veranstaltung verstanden. Im Beispiel sind zwei weitere Kurse K&D (KOMMUNIKATION UND DATENHALTUNG [C&M-KuD]) und ISWA (INTERNET-SYSTEME UND WEB-APPLIKATIONEN [C&M-ISWA]) genannt.

Ein Kurs wird im Beispiel durch die Klasse Course mit den zwei Attributen shortName (Kurzbezeichnung) und number (eindeutige Kursnummer) beschrieben. Es wären zahlreiche weitere Attribute denkbar, wie z.B. der den Kurs anbietende Dozent, die Art der Veranstaltung (z.B. Vorlesung oder Praktikum) oder die Anzahl der Semesterwochenstunden, die der Kurs umfasst.

Der Kurskatalog cat wird durch ein Array von Kursen Course[] implementiert, wobei insgesamt maximal 1000 Kurseinträge (new Course [1000]) im Katalog aufgenommen werden können.

Durch die nachfolgende Anweisungsfolge in der main()-Methode werden die drei oben beschriebenen Kurse erfasst und in einer for-Anweisung ausgegeben.

Im obigen Java-Programm wurden Informationen nur für den ersten Kurs zugewiesen. Das Programm ist entsprechend zu ergänzen, damit der Katalog den in der oberen Abbildung in Interaktion 2 gezeigten Zustand besitzt.

Offensichtlich bieten sich für die Erfassung und die Ausgabe, aber auch für andere Katalog-Operationen (z.B. Suche oder das Löschen) separate Methoden an. Diese Überlegungen führen zu der im folgenden Kapitel näher ausgeführten Objektorientierung.

2 OBJEKTORIENTIERUNG

Bislang lag der Schwerpunkt auf dem Datenaspekt von Klassen. Dabei wurde deutlich, dass Klassen dazu geeignet sind, semantisch zusammengehörige Daten in eine Datenstruktur zusammen zu führen.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Durch Klassen lassen sich nicht nur die Daten, sondern auch die dazugehörigen Methoden zu einer Einheit zusammenfassen
 - diese Einheit von Daten und darauf operierende Methoden schaffen eine geeignete Ordnung im Programm
- Beispiel: Klasse zur Beschreibung eines Kurses
 - Daten: Kurzbezeichnung, Nummer
 - Methoden
 - Erfassen der Kursdaten
 - Ändern der Kursdaten
 - Abfrage der Kursdaten

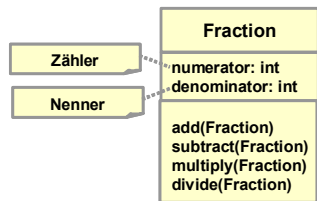
Information 7: OBJEKTORIENTIERUNG - Daten und Methoden als eine Einheit

Wie in Information 7 ausgeführt ist, lassen sich mittels Klassen nicht nur Daten, sondern insbesondere Daten und die darauf arbeitenden Operationen, also die Methoden zusammenfassen. Die Einheit aus Daten und Methoden stellt eine geeignete Form zur Gliederung komplexer Software-Systeme dar.

Programme, die beispielsweise die im letzten Abschnitt betrachtete Klasse `Course` nutzen möchten, stehen damit neben den Daten gleichzeitig auch alle auf diesen Daten sinnvoll anwendbaren Methoden zur Verfügung.

2.1 Beispiel: Klasse `Fraction`

Am Beispiel des Bruchrechnens und der Klasse `Fraction` wird die Zusammenfassung von Daten und Methoden als zentrales Konzept der Objektorientierung verdeutlicht.



- Methoden sind lokal zu Fraction
 - greifen auf die Daten (Attribute n und d) des Fraction-Objekts zu
- this bezeichnet dasjenige Objekt, auf das eine Methode gerade angewendet wird
- Die Hilfsvariable n0 wird in divide() für den Fall $f == this$ benötigt
 - $n = n * f.d$ würde n und damit in diesem Fall auch f.n verändern
 - $d = d * f.n$ würde dann zu einem falschen Resultat führen

```

class Fraction {
    int n; // numerator
    int d; // denominator

    void add (Fraction f) {           // n1/d1 + n2/d2
        n = n * f.d + f.n * d;       // = (n1*d2 + n2*d1)
        d = d * f.d;                 // / d1 * d2
    }
    // subtract() and multiply() to be added

    void divide (Fraction f) {       // n1/d1 / n2/d2 =
        int n0 = n * f.d;            // = n1 * d2
        d = d * f.n;                 // / d1 * n2
        n = n0;
    }
}
  
```

Interaktion 3: Beispiel-Klasse Fraction

Interaktion 3 zeigt die Klasse Fraction mit den Daten numerator und denominator sowie den vier Methoden add(), subtract(), multiply() und divide() als graphische UML-Beschreibung und als unvollständige Java-Klasse, die um die fehlenden Methoden zu ergänzen ist.

Wie am Beispiel der Methode add() zu erkennen ist, kann im Rumpf einer Methode ein Zugriff auf die in der Klasse lokal deklarierten Daten erfolgen. Diese Daten sind dem Objekt zugeordnet, dessen Methoden aufgerufen werden. Java bietet mit dem Schlüsselwort this die Möglichkeit, dieses Objekt explizit anzusprechen.

Am Beispiel der Methode add() soll der Umgang mit dem Objekt this verdeutlicht werden. In der ersten Zeile

$$n = n * f.d + f.n * d;$$

bedeuten n und d den Zugriff auf die Daten des Objekts, auf das die Methode angewendet wird, weshalb die Zuweisung auch gleichbedeutend lauten könnte:

$$this.n = this.n * f.d + f.n * this.d;$$

Es ist nicht ausgeschlossen, dass eine Methode mit dem Objekt als formalen Parameter aufgerufen wird, auf das die Methode angewendet wird. Aus diesem Grund muss in der Methode divide() eine Methoden-lokale Variable zur Aufnahme des Zwischenresultats eingeführt werden.

```

Fraction a = new Fraction(); a.n = 1; a.d = 2;
Fraction b = new Fraction(); b.n = 3; b.d = 5;

a.add(b);           // a = 1/2 + 3/5 = 11/10
b.divide(b);       // b = 3/5 * 3/5 = 15/15

```

- Objektorientierte Sprechweise zum Aufruf der Methode a.add(b):
 - Objekt a bekommt die Meldung (oder den Auftrag) add
 - hierdurch Aufruf der Methode add()
 - Objekt a, das die Meldung erhält, wird der Empfänger der Meldung genannt
- Die Fraction-Methoden haben neben dem formalen Parameter f noch einen zweiten (versteckten) formalen Parameter this
 - beim Aufruf a.add(b) wird this der aktuelle Parameter a und f der aktuelle Parameter b zugewiesen
 - zu beachten: dem formalen Parameter f kann als aktueller Parameter auch das Empfänger-Objekt zugeordnet werden

Information 8: Methodenaufruf

In Information 7 werden zwei Objekte a und b der Klasse Fraction erzeugt und die Klassenvariablen werden so initialisiert, dass a den Bruch 1/2 und b den Bruch 3/5 beinhaltet.

In den nachfolgenden Anweisungen erhält der Empfänger a die Meldung add mit b als aktuellen Parameter und b erhält die Meldung divide mit dem aktuellen Parameter b. In der erstgenannten Anweisung wird der Standardvariablen this beim Aufruf der Methode der Wert a und in der zweiten Anweisung der Wert b zugewiesen.



Aufruf b.divide(b)

```

void divide (* Fraction this, */ Fraction f) {
  int n0 = this.n * f.d; // int n0 = b.n * b.d;
  this.d = this.d * f.n; // b.d = b.d * b.n;
  this.n = n0;           // b.n = n0;
}

```

- Warum wird die Variable n0 benötigt?
-
- Könnte man die Variable statt n0 auch n nennen?
-

Interaktion 4: Versteckter Parameter this im Methodenaufruf

Interaktion 4 führt den versteckten Parameter this beim Methodenaufruf durch entsprechende Kommentare im Methodenkopf explizit auf. Am Beispiel des konkreten Aufrufs b.divide(b) kann man sich klarmachen, warum die Einführung einer lokalen Hilfsvariablen n0 erforderlich ist.

Die zweite Frage zur Benennung dieser Variablen zielt darauf ab, ob hierdurch im Falle einer Umbenennung zu `n` ggf. ein Konflikt mit dem gleichnamigen Attribut `n` entsteht.

2.2 Konstruktoren

Konstruktoren sind spezielle Methoden, die bei der Erzeugung eines Objekts automatisch aufgerufen werden und die dazu genutzt werden können, dieses Objekt in der gewünschten Form zu initialisieren.

```
class Fraction {
    int n = 0; int d = 1;

    Fraction(int n) {
        this.n = n;
    }

    Fraction (int n, int d) {
        this.n = n; this.d = d;
    }
}
```

- Anweisung: `Fraction a = new Fraction();`
 - es wird ein Objekt `a` mit der Standardinitialisierung (hier: `n = 0, d = 1`) erzeugt
 - diese Standardinitialisierung erbringt die Konstruktormethode `Fraction()`
 - hat den gleichen Namen wie die Klasse
 - wird ohne Funktionstyp und ohne das Schlüsselwort `void` deklariert
- Die Konstruktormethode kann durch selbst geschriebene Konstruktoren überladen werden

// constructor is called when new object is created

```
Fraction x = new Fraction();           // standard constructor
Fraction y = new Fraction(3, 5);       // overloaded constructor
Fraction z = new Fraction(6);          // overloaded constructor

z.divide(x.add(y)); // result? _____
```

Interaktion 5: Konstruktormethoden

Konstruktormethoden haben im Vergleich zu den bislang kennen gelernten Methoden einige besondere Eigenschaften. Neben den in Interaktion 5 genannten Namens- und Typ-Charakteristika ist zu beachten, dass ein Konstruktor niemals explizit, sondern immer nur implizit während der Erzeugung eines Objekts (durch das Schlüsselwort `new`) aufgerufen wird.

Die bereits beschriebene Eigenschaft des Methodenüberladens kann dazu genutzt werden, eigene Konstruktoren einzuführen. Die in Konstruktoren erlaubten formalen Parameter werden dabei üblicherweise dazu genutzt, die Initialisierungswerte der Klassenvariablen bei der Objekterzeugung mittels `new` zu übergeben, wie das Beispiel in Interaktion 5 verdeutlicht. Es wird zudem ersichtlich, dass mehrere Konstruktoren bestehen können.

Die Angabe eines oder mehrerer Konstruktoren zu einer Klasse ist optional. Java fügt automatisch einen parameterlosen Standard-Konstruktor hinzu, der die Attribute mit den gemäß ihres Typs bestehenden Standardwerten (`0`, `null`, `false`, ...) belegt.

Eine andere Form der Initialisierung besteht darin, den zu initialisierenden Wert bei der Deklaration der Klassenattribute anzugeben. Falls keine eigenen Konstruktoren existieren, treten diese Initialisierungswerte an die Stelle der Standard-Initialisierungswerte. Im Falle selbst geschriebener Konstruktoren werden die Initialisierungen ggf. nochmals verändert. Bezogen auf das Beispiel in Interaktion 5 bedeutet diese Festlegung, dass beispielsweise das Objekt `y` nach dessen Erzeugung nicht die Initialisierungswerte der Deklaration, sondern die durch den Konstruktor-Aufruf zugewiesenen Werte beinhaltet.

2.3 Klassenattribute und Klassenmethoden versus Objektattribute und Objektmethoden

In der Beispiel-Klasse `Fraction` wurden erstmals Methoden verwendet, die nicht mit dem Schlüsselwort `static` deklariert wurden. Dieser Sachverhalt betrifft die in der Objektorientierung wichtige Unterscheidung zwischen klassenbezogenen (statischen) und objektbezogenen (dynamischen) Attributen und Methoden.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Sowohl die Klasse selbst als auch die von dieser Klasse erzeugten Objekte können Attribute und Methoden haben
 - eine Klasse ist selbst ein Objekt, das wie eine Schablone das Aussehen aller Objekte dieser Klasse festlegt
- Syntaktische Unterscheidung
 - Klassenattribute und -methoden werden mit dem Schlüsselwort "static" gekennzeichnet
- Klassenattribute existieren nur einmal pro Klasse, Objektattribute werden getrennt für jedes erzeugte Objekt angelegt

Information 9: Klassenbezogene und objektbezogene Attribute und Methoden

Objektattribute werden erst zur Zeit der Objekterzeugung angelegt und sind diesem speziellen Objekt zugeordnet. Eine Änderung eines Objektattributwertes durch eine Objektmethode hat keine Auswirkung auf den Wert des gleichnamigen Objektattributs eines anderen Objekts. Im Gegensatz dazu bestehen Klassenattribute nur einmal pro Klasse, weshalb eine Änderung eines Klassenattributs für alle Objekte wirksam wird.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

```
class Fraction {
    static int fractionCounter           // class attribute
    int n = 1; int d = 1;               // object attributes

    static { fractionCounter = 0; }     // class constructor; only one is allowed
    static void reset FractionCounter () { // class method
        fractionCounter = 0;
    }

    Fraction () { ... }                 // constructor method(s)
    Fraction (int n, int d) { ... }     // more than one constructor is allowed

    void add (Fraction f) { ... }      // object methods
    ...
}
```

Information 10: Klassen- und objektbezogene Elemente in der Klasse `Fraction`

Der in Information 10 gezeigte Programmausschnitt zur Klasse `Fraction` macht deutlich, wie die verschiedenen klassen- und objektbezogenen Elemente in einer Klassendefinition genutzt

werden können. Wie das Beispiel zeigt, besteht nicht nur die Möglichkeit, objektbezogene Konstruktormethoden einzuführen. Es kann auch ein Klassenkonstruktor benutzt werden, um die Klassenattribute (hier FractionCounter) zu initialisieren.

Der Zugriff auf Klassenattribute und Klassenmethoden kann im Falle von Mehrdeutigkeiten durch die Angabe des Klassennamens qualifiziert werden.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- In jedem Java-Programm kommt eine spezielle (öffentliche, also public) Klassenmethode main() vor
- Konvention: diese Klassenmethode kann von der Kommandozeile aus wie ein Programm aufgerufen werden
- Die main()-Methode kann beim Aufruf mit Parametern versehen werden
 - Beispiel: java MyProg test 12
 - hierdurch werden zwei Strings "test" und "12" als Parameter im Stringarray arg übergeben

Information 11: Spezielle Klassenmethode main()

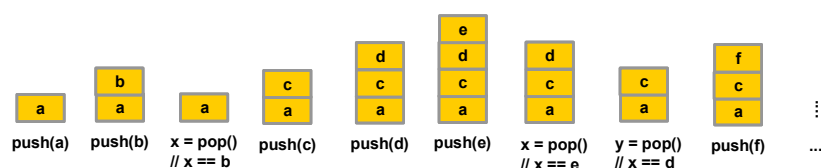
Die Klassenmethode main() spielt in einem Java-Programm eine besondere Rolle, da hierdurch die Einsprungstelle in das Programm vorgegeben wird. Erreicht wird dieses Verhalten durch die in Information 11 angegebene Konvention, die auf der Bezeichnung der Methode basiert.

2.4 Stapel als Beispiel einer Klasse

Der in diesem Abschnitt als eine Klasse beschriebene Stapel – auch Keller bzw. *Stack* genannt – ist eine der elementarsten und wichtigsten Datenstrukturen der Informatik.

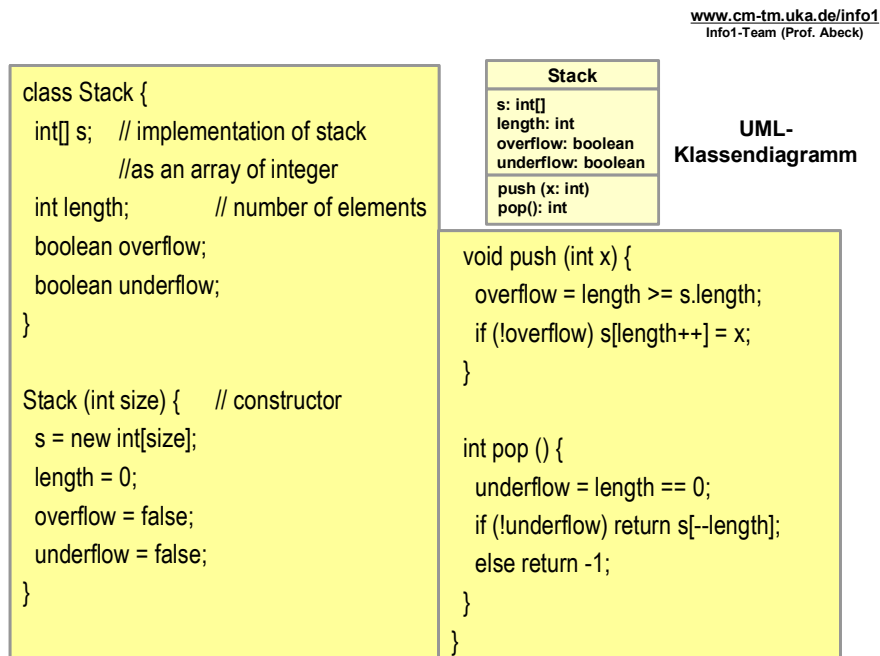
www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Wichtige Datenstruktur der Informatik, durch die ein bestimmtes Zugriffsprinzip auf eine Elementemenge vorgegeben wird
- Zugriffsprinzip: Das zuletzt auf den Stapel gelegte Element wird als erstes wieder entfernt
 - wird als Kellerprinzip oder LIFO-Prinzip bezeichnet
 - Last In First Out
- Stapeloperationen sind
 - push(x) ein Element x auf den Stapel legen
 - x = pop() das oberste Element x vom Stapel nehmen



Information 12: Stapel (Keller, *Stack*) als Beispiel einer Klasse

Durch den Stapel wird ein ganz bestimmtes, in Information 12 beschriebenes Zugriffsprinzip vorgegeben, durch das die Wirkungsweise der Schreiboperation `push()` und der Leseoperation `pop()` festgelegt ist. Anhand des beispielhaften Verlaufs eines Stapelinhalts lässt sich diese auch als Kellerprinzip oder LIFO-Prinzip (*Last In First Out*) bezeichnete Charakteristik der Datenstruktur einfach nachvollziehen.



Information 13: UML-Klassendiagramm und Java-Programm zu Stack

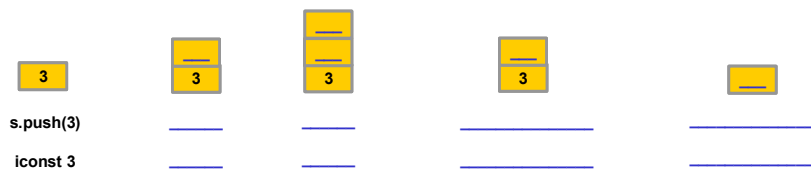
Bei der Umsetzung der Datenstruktur muss u.a. entschieden werden, welche Elemente auf dem Stapel gehalten und wie die Daten gespeichert werden sollen. In der exemplarischen Implementierung eines Stapels in Information 13 sind die Stapелеlemente ganze Zahlen, die in einem `int`-Array gehalten werden.

In beiden Methoden `push()` und `pop()` muss vor dem Schreiben bzw. Lesen des Elements zunächst überprüft werden, ob der Stapel voll (`overflow`) bzw. leer (`underflow`) ist. In diesem Fall kann kein Element auf den Stapel geschrieben bzw. vom Stapel gelesen werden.



- Mittels Stapeln lassen sich beliebige geklammerte Ausdrücke berechnen
 - wird z.B. in der Java Virtual Machine ausgenutzt

<pre>Stack s = new Stack(50); s.push(3); s.push(7); s.push(5); s.push(s.pop() * s.pop()); s.push(s.pop() + s.pop());</pre>	<p>Entsprechende Befehlsfolge (sog. Bytecode) der Java Virtual Machine</p> <pre>iconst 3 legt eine Konstante auf den Stapel iconst 7 iconst 5 imul Multiplikation auf dem Stapel iadd Addition auf dem Stapel</pre>
--	--



- Berechneter Ausdruck: _____

Interaktion 6: Verwendung von Stapeln

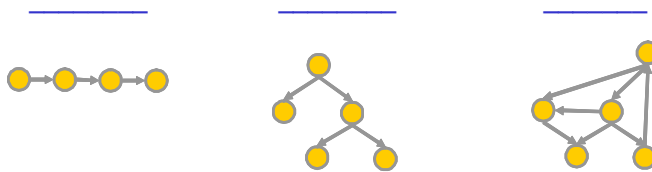
An einem einfachen Anwendungsbeispiel der Klasse Stack kann man sich die grundsätzliche Bedeutung dieser Datenstruktur für die Informatik klarmachen. Stapel lassen sich dazu nutzen, beliebige Ausdrücke zu berechnen, indem die Operanden in gewisser Reihenfolge auf den Stapel geschrieben (push) werden und auf die jeweils obersten Elemente die im Ausdruck auftretenden Operationen ausgeführt werden. Im Rahmen der Ausführung der Operation wird das Operationsergebnis anstelle der Operanden auf den Stapel geschrieben.

Genau in dieser Art und Weise arbeitet z.B. auch die zum Java-Laufzeitsystem gehörende *Java Virtual Machine* (JVM), wie anhand des Beispiels im Rahmen der Interaktion 6 skizziert werden soll.

3 DYNAMISCHE DATENSTRUKTUREN

Zahlreiche in der Informatik relevante Datenstrukturen sind dadurch gekennzeichnet, dass sie sich dynamisch zur Laufzeit aufbauen und manipulieren lassen. Aufgrund dieser Eigenschaft werden sie als dynamische Datenstrukturen bezeichnet.

- Dynamische Datenstrukturen bestehen aus verketteten Knoten
 - Knoten können dynamisch zur Laufzeit erzeugt und anschließend verkettet werden
 - Knoten können dynamisch zur Laufzeit wieder entfernt oder umgehängt werden
- Typische dynamische Datenstrukturen



Interaktion 7: DYNAMISCHE DATENSTRUKTUREN - Die wichtigsten Arten

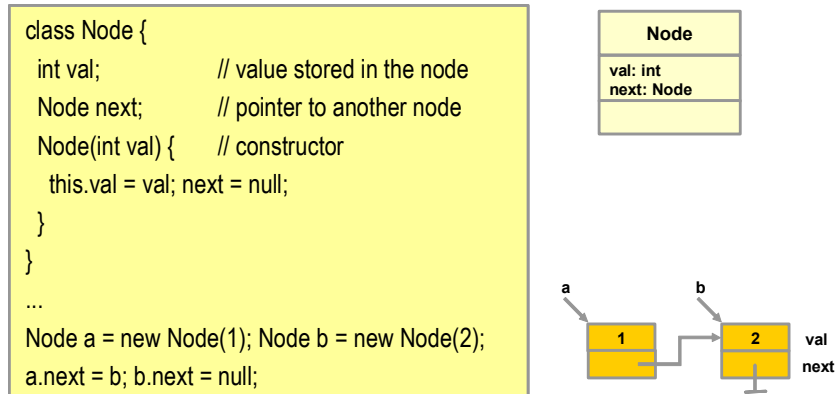
Interaktion 7 zeigt drei typische dynamische Datenstrukturen:

1. **Listen:** mit Ausnahme des letzten Knotens hat jeder Knoten genau einen Zeiger auf den Nachfolger-Knoten.
2. **Baum:** jeder Knoten kann mehrere Zeiger auf Nachfolger-Knoten haben und wird nur durch maximal einen Vorgänger-Knoten referenziert.
3. **Graph:** jeder Knoten kann mehrere Zeiger auf Nachfolger-Knoten und kann durch mehrere Vorgänger-Knoten referenziert werden.

Die drei genannten Datenstrukturen sind nicht zuletzt deshalb für die Software-Entwicklung so bedeutsam, weil hiermit viele Gegenstände der realen Welt gut modelliert werden können. So treten Listen z.B. in einer Lagerhaltung in Form von Einkaufslisten auf, mittels Bäumen lassen sich Hierarchien, wie z.B. die Struktur eines Unternehmens abbilden und Graphen sind zur Darstellung beliebiger Arten von Netzwerken, z.B. einem Verkehrs- oder Kommunikationsnetz, geeignet.

Die Grundlage zur Erstellung und zur Bearbeitung von dynamischen Datenstrukturen bilden die Zeiger, mit denen auf beliebige andere Objekte referenziert werden kann. Dabei ist auch zulässig, dass ein Objekt einer Klasse auf ein Objekt derselben Klasse verweist, was bei allen oben eingeführten Datenstrukturen der Fall ist, da diese nur Objekte einer Klasse beinhalten.

- Der Aufbau und die Manipulation von dynamischen Datenstrukturen erfolgt durch das Verketteten von Knoten
 - Verbindung zu einem entfernten Knoten wird durch einen Zeiger (Pointer) auf diesen Knoten realisiert



Information 14: Verketteten von Knoten

Ein Knoten besteht im einfachsten Fall aus einem Wert (hier eine ganze Zahl `int val`, siehe Information 14) und einem Verweis auf einen anderen Knoten (hier `Node next`). Durch Deklaration, Erzeugung und Zeigermanipulationen (z.B. `a.next = b`) lässt sich eine einfache dynamische Struktur aufbauen.

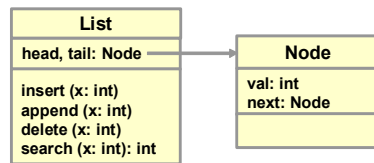
Die Abbildung in Information 14 zeigt die erstellte dynamische Datenstruktur.

3.1 Listen

Die eingeführte Klasse `Node` kann dazu verwendet werden, die in einer Liste zu pflegenden Daten zu halten. Die Liste selbst und die mit dieser Datenstruktur verbundenen typischen Operationen wie

- Einfügen eines neuen Elements am Anfang (*insert*),
- Anhängen eines neuen Elements am Ende (*append*),
- Löschen eines bestehenden Elements (*delete*) oder
- Suchen eines Elements (*search*)

stellen eine eigenständige Abstraktion dar, die in einer separaten Klasse `List` zusammengefasst werden sollte.



- Listen stellen eine eigenständige Abstraktion dar, die von den Listenknoten als reine Datenklassen getrennt werden sollte
- Methode insert(x)
 - Einfügen eines Knotens am Anfang der Liste

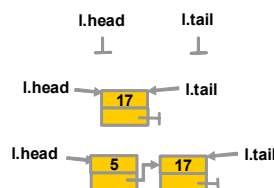
```

class List {
    Node head = null; Node tail = null;
    // methods
}
  
```

```

void insert (int val) {
    Node p = new Node(val);
    if (tail == null) tail = p;
    p.next = head;
    head = p;
}
  
```

- Beispiel:



```

List l = new List();

l.insert(17);
l.insert(5);
  
```

Information 15: Listen

Information 15 zeigt das Ergebnis der Modellierung einer Liste in einem UML-Klassendiagramm. Als Attribute werden in List zwei auf Objekte der Klasse Node verweisende Zeiger head und tail vorgesehen, die auf das erste Element bzw. das letzte Element der Liste zeigen sollen.

Den beiden Zeigern werden bei der Erzeugung einer neuen Liste durch entsprechende Initialisierung bei der Deklaration der beiden Attribute die null-Pointer zugewiesen.

Die Methode void insert(int val), die einen Knoten mit dem int-Wert val am Anfang der Liste einfügt, zeigt exemplarisch, wie mit den Zeigern head und tail zu verfahren ist, um das gewünschte Ergebnis zu erhalten.

Das Anhängen eines neuen Elements am Ende (void append(int val)) erfolgt in ganz ähnlicher Form und wird daher an dieser Stelle nicht näher ausgeführt.

3.2 Suchen in einer unsortierten Liste

Bei den Listen gehört das Suchen wie bei den Arrays zu den wichtigsten Operationen. Da beim Einfügen mittels der Methode insert() nicht auf eine bestimmte Reihenfolge geachtet wurde – die Liste also unsortiert ist – muss die Liste sequentiell durchlaufen werden, bis das gesuchte Element gefunden wurde. Falls das Ende der Liste erreicht wird, war die Suche erfolglos.



- Eingaben
 - Liste mit ganzzahligen Werten
 - ganzzahliger Such-Wert
- Ausgaben
 - Zeiger auf das entsprechende Listenelement, falls der Such-Wert in der Liste gefunden wurde
 - null, falls kein Listenelement mit dem Such-Wert in der Liste vorkommt

```
Node search (int val) {  
    Node p = head;           // set p to first element  
    _____             // traverse list by using p  
    _____  
    // end loop assertion: p == null || p.val == val  
    return p;  
}
```

Interaktion 8: Suchen eines Listenelements

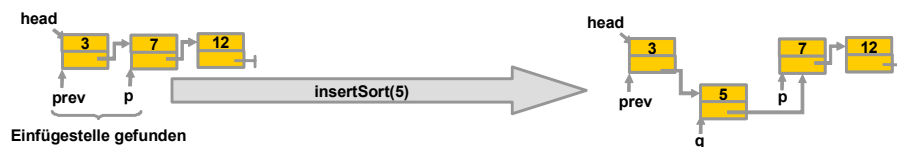
Dieses Vorgehen der sequentiellen Suche ist im Programm in Interaktion 8 durch eine geeignete Schleifenanweisung zu ergänzen.

Das Suchen wird beispielsweise in der Methode `void delete(int val)` zum Löschen eines Elements benötigt, das den angegebenen Wert besitzt.

3.3 Einfügen in eine sortierte Liste

Das Suchen könnte effizienter durchgeführt werden, falls die Liste sortiert wäre. Diese Anforderung wird durch die bisherigen Operationen (`insert()` bzw. `append()`) zum Aufbau einer Liste nicht erfüllt, da das Element unabhängig von dessen Wert eingefügt wird.

- Eine sortierte Liste liefert den Vorteil des schnelleren Suchens
 - neue Anforderung: beim Einfügen eines Elements muss die Sortiert-Eigenschaft bestehen bleiben
- Algorithmus-Überlegung anhand eines Beispiels



```
void insertSort (int val) {
    Node p = head; Node prev = null;    // initialize both traversal pointers
    while (p != null && val > p.val) {    // search insert position
        prev = p; p = p.next; }
    Node q = new Node(val);             // generate node to be inserted
    q.next = p;                          // insert node at the correct position
    if (p == head) head = q; else prev.next = q;
}
```

Information 16: Einfügen in eine sortierte Liste

Die Sortiert-Eigenschaft wird nur dann nach dem Einfügen eines neuen Elements erhalten bleiben, wenn zunächst die durch die Sortierung vorgegebene Einfügeposition gesucht wird. An einem einfachen Beispiel kann man sich den Algorithmus zu der Methode `insertSort()` erarbeiten (siehe Information 16). Offensichtlich ist zum Einhängen des neuen Elements in die Liste nicht nur der Zeiger auf das aktuell überprüfte Element erforderlich (hier `p`), sondern es wird zusätzlich noch ein weiterer Zeiger benötigt, der das unmittelbar vor dem aktuellen Element liegende Element (`prev`) in der Liste referenziert. Das Problem in diesem Zusammenhang ist, dass zu einem gegebenen Element immer nur das Nachfolger-Element ermittelt werden kann. In diesem Fall wird aber das Vorgänger-Element benötigt, was nur effizient durch den zusätzlichen Hilfszeiger ermittelt werden kann.

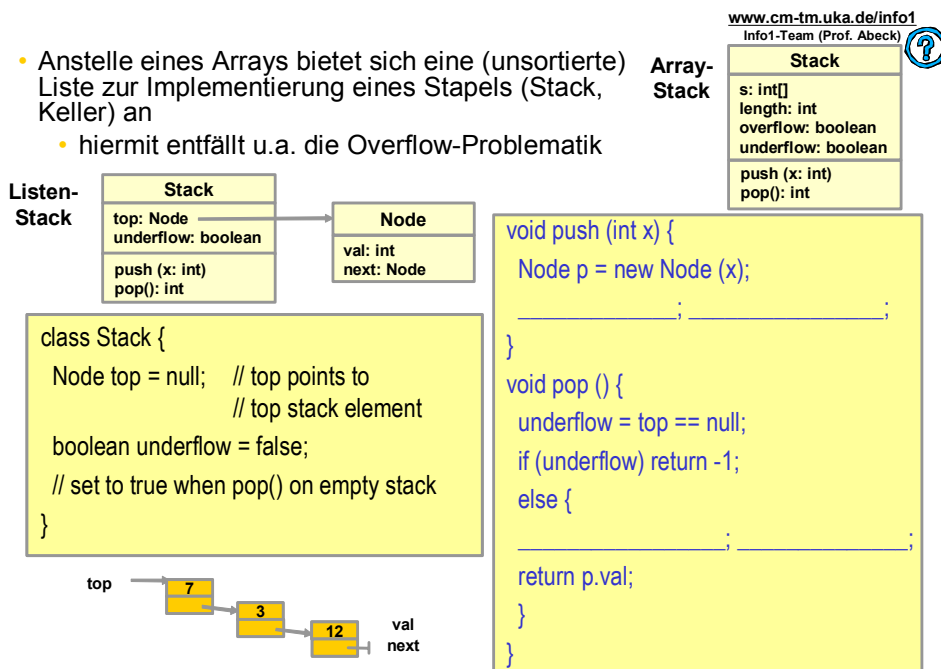
Listen, die nur einen Zeiger auf den Nachfolger vorsehen, heißen einfach verkettet. Werden zu jedem Listenelement sowohl der Nachfolger als auch der Vorgänger in der Datenstruktur gehalten, so heißt die Liste doppelt verkettet.

Die Suche in einer sortierten Liste lässt sich offensichtlich effizienter gestalten, da das sequentielle Durchlaufen im Fall, dass das Element nicht in der Liste enthalten ist, früher abbrechen kann, wie am Beispiel der Arrays in der Kurseinheit IMPERATIVE PROGRAMMIERUNG [C&M-IP] näher ausgeführt ist.

3.4 Stapel als verkettete Liste

Die Datenstruktur der verketteten Liste bietet eine interessante Alternative zur Realisierung der kennen gelernten Datenstruktur des Stapels (*Stack*, Keller).

- Anstelle eines Arrays bietet sich eine (unsortierte) Liste zur Implementierung eines Stapels (Stack, Keller) an
 - hiermit entfällt u.a. die Overflow-Problematik



Interaktion 9: Stapel als verkettete Liste

Interaktion 9 stellt die *Stack*-Varianten "Array-Stack" und "Listen-Stack" in Form der UML-Klassendiagramme gegenüber. Beide Datenstrukturen stimmen in ihrem Verhalten insoweit überein, dass sie das gleiche Zugriffsprinzip "Last In First Out" (LIFO) realisieren. Der wichtigste Unterschied bzgl. des Verhaltens betrifft das *Overflow*-Problem.

Die *push()*-Operation wird beim *Listen-Stack* (ganz ähnlich wie die *insert()*-Operation) in Form einer unsortierten Liste realisiert, da das Stapелеlement am Anfang der den Stapel implementierenden Liste eingefügt werden muss. Die Methode *pop()* liefert das erste Listenelement für den Fall, dass die Liste nicht leer ist, d.h. *top* != null. Das vom Stapel genommene Element ist nach Ausführung der Methode dann nicht mehr zugänglich.

In Interaktion 9 sind die in den beiden Zugriffsmethoden durchzuführenden Zeiger-Zuweisungen entsprechend zu ergänzen.

- Die beiden Datenstrukturen Array-Stack und Listen-Stack haben die gleiche Klassen-Schnittstelle
 - (Klassen-) Schnittstelle = Namen der Methoden und deren Parameter
- Eine Klasse mit einer bestimmten Schnittstelle kann auf verschiedene Arten implementiert werden
- Ziel: Austausch der Implementierung einer Klasse (z.B. aus Effizienzgründen) ohne Änderung der Schnittstelle
 - Programme, die diese Klasse nutzen, müssen nicht verändert werden
 - Verstecken der internen Datenstrukturen einer Klasse gegenüber der Außenwelt
 - Information Hiding

Information 17: Schnittstelle und *Information Hiding*

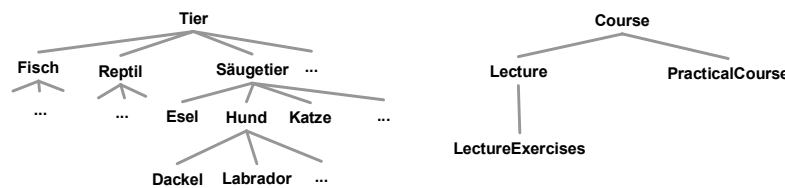
Am Beispiel der beiden Datenstrukturen wird deutlich, dass eine Klassen-Schnittstelle, die durch die Namen der Methoden und deren Parameter bestimmt ist, durch unterschiedliche Implementierungen realisiert werden kann. Die Implementierungen unterscheiden sich dabei üblicherweise bezüglich ihrer Effizienz oder gewisser anderer Merkmale, wie anhand des *Overflow*-Aspekts im obigen Beispiel aufgezeigt wurde.

Mit den Schnittstellen eng verknüpft ist das *Information Hiding*, das ein zentrales Ziel der Objektorientierung darstellt, wie in Information 17 ausgeführt wird

4 VERERBUNG

Zur Objektorientierung gehört das im Folgenden näher beschriebenen Prinzip der Vererbung. Vererbung ist eng mit dem Klassenkonzept verknüpft und stellt eine spezielle Art einer Klassenbeziehung dar.

- Die Vererbung ist eine Art von Beziehung zwischen Klassen
 - eine andere Art von Beziehung ist z.B. das Enthaltensein (Containment)
- Vererbungsbeziehungen sind in der objektorientierten Programmierung mit Umsicht zu nutzen
 - können das Programm komplex und damit schwer durchschaubar machen
- Die Vererbungsbeziehung tritt an vielen Stellen in der Wirklichkeit auf
 - Klassen dienen dazu, Dinge der realen Welt zu modellieren
- Beispiele für Vererbungshierarchien



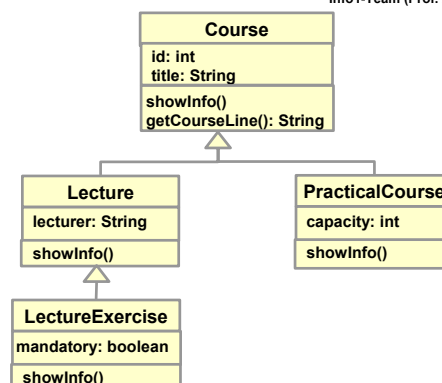
Information 18: VERERBUNG - Einführung und Beispiele

Vererbungsbeziehungen treten an vielen Stellen in der Realität auf, wie an den zwei unterschiedlichen Beispielen von Vererbungshierarchien in Information 18 verdeutlicht wird.

4.1 Modellieren und Programmieren von Vererbungsbeziehungen

Die *Unified Modeling Language* (UML) ermöglicht die Modellierung von Vererbung durch die Kennzeichnung einer Klassenbeziehung mittels einer hohlen Pfeilspitze. Dabei ist die Klasse, auf die die Pfeilspitze zeigt, die so genannte Oberklasse. Die andere mit der Oberklasse in der Vererbungsbeziehung stehende Klasse heißt Unterklasse.

- Modellierung einer Vererbungsbeziehung in einem UML-Klassendiagramm durch eine hohle Pfeilspitze
 - Attribute und Methoden der Oberklasse, auf die die Pfeilspitze zeigt, werden der Unterklasse vererbt
- Programmierung einer Vererbungsbeziehung in Java durch das Schlüsselwort `extends`



```
class Course { ... }
class Lecture extends Course { ... }
class LectureExercise extends Lecture { ... }
class PracticalCourse extends Course { ... }
```

Interaktion 10: Modellieren und Programmieren von Vererbungsbeziehungen

Im Beispiel in Interaktion 10 übernimmt die Unterklasse `Lecture` die Attribute `id` und `title` sowie die Methoden `showInfo()` und `getCourseLine()` von deren Oberklasse `Course`. Während die Methode `showInfo()` die Attributwerte auf dem Bildschirm ausgibt, erzeugt `getCourseLine()` einen String, der als Eintrag für ein Verzeichnisverzeichnis dient.

Gegenüber der Klasse `LectureExercise` ist `Lecture` ihrerseits eine Oberklasse. Das bedeutet, dass `LectureExercise` alle Attribute und Methoden von `Lecture` erbt – das sind in diesem Fall sowohl die in `Lecture` definierten Attribute (`lecturer`) und Methoden (`showInfo()`) als auch die von `Course` geerbten Attribute und Methoden. Außerdem wird in `LectureExercise` ein boolesches Attribut `mandatory` definiert, das `true` ist, falls die zu der Vorlesung angebotene Übung verpflichtend (*mandatory*) ist (weil beispielsweise von den Teilnehmern ein Pflichtenchein erworben werden muss).

Die Vererbungsbeziehung zwischen zwei Klassen lässt sich in Java einfach durch Verwendung des Schlüsselworts `extends` ausdrücken, wie der in Interaktion 10 zu ergänzende Programmausschnitt verdeutlicht.

Einen praktischen Nutzen der Vererbungsbeziehung kann man sich anhand der vererbten Methode `getCourseLine()` klarmachen: Diese Methode wird nur einmal in der Oberklasse `Course` definiert und kann aufgrund der Vererbungsbeziehung in allen Unterklassen (`Lecture`, `LectureExercises`, `PracticalCourse`) verwendet werden, um zu jeder dieser verschiedenen Veranstaltungen den Verzeichniseintrag bestehend aus `id` und `title` zu erzeugen.

Anders verhält es sich mit der Methode `showInfo()`, die in den Unterklassen überschrieben wird. Es lässt sich leicht nachvollziehen, dass die Methode `showInfo()` zu einem Objekt der Klasse `Lecture` nicht nur die Information zu seiner Oberklasse `Course`, sondern auch den Namen des Dozenten ausgeben möchte, der im Attribut `lecturer` gespeichert ist.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Von einer Oberklasse geerbte Methoden können in der Unterklasse überschrieben werden
 - Überschreiben heißt, dass die Methode mit der gleichen Schnittstelle neu deklariert wird
 - Beispiel: Methode `showInfo()`
- In der überschreibenden Methode einer Unterklasse kann mittels eines speziellen Aufrufs `super.<Methodennamen>` auf die (überschriebene) Methode der Oberklasse zugegriffen werden

```
class Lecture extends Course {
    ...
    void showInfo() {           // Lecture method which is overwriting Course method
        super.showInfo();      // calls showInfo() from class Course
        Out.println("Lecturer: " + lecturer);
        ....
    }
}
```

Information 19: Methoden und Vererbungsbeziehung

Wie der Programmausschnitt in Information 19 zeigt, kann in der überschreibenden Methode zum Objekt der Unterklasse (*Sub Class*) die überschriebene Methode zu dem Objekt der Oberklasse (*Super Class*) durch einen so genannten `super`-Aufruf in Anspruch genommen

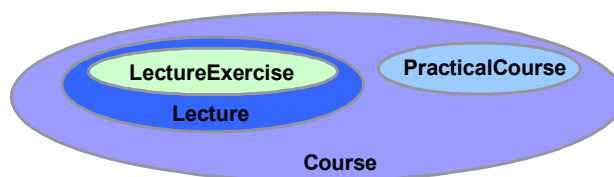
werden. Hierdurch lässt sich in der Beispiel-Methode `showInfo()` zu einem `Lecture`-Objekt zunächst die Information zum `Course`-Objekt, dessen Eigenschaften geerbt wurden, anzeigen (`super.showInfo()`) und um die zusätzlich anzuzeigende Information (hier `lecturer`) ergänzen.

4.2 Polymorphie und Kompatibilität

Durch die Vererbung wird bewirkt, dass Attribute und Methoden der Oberklasse automatisch auch zur Unterklasse gehören. Hieraus resultiert, dass eine Objektvariable der Unterklasse so auftreten kann, als wäre sie ein Objekt der Oberklasse. Die Objektvariable kann also in Gestalt verschiedener Klassen auftreten. Diese mit der Objektorientierung verbundene Eigenschaft der Vielgestaltigkeit wird auch als Polymorphie bezeichnet.

www.cm-tm.uka.de/info1
Info1-Team (Prof. Abeck)

- Der Nutzen der Vererbung liegt darin, dass eine Unterklasse mit einer Oberklasse kompatibel ist
 - jedes Programm, das in der Lage ist, mit Objekten der Oberklasse zu arbeiten, kann auch mit Objekten der Unterklasse arbeiten
 - Objekte können polymorph (vielgestaltig) sein
- Die Kompatibilität gilt, weil jedes Objekt einer Unterklasse zugleich auch ein Objekt der Oberklasse ist
 - "ist ein"-Beziehung (englisch "is a")
 - Objekt der Unterklasse "ist ein" Objekt der Oberklasse
 - in umgekehrter Richtung gilt die "ist ein"-Beziehung nicht



Information 20: Kompatibilität zwischen Oberklasse und Unterklasse

Die Eigenschaft der Polymorphie ist aufgrund der Kompatibilität zwischen Oberklasse und Unterklasse gegeben. Die Abbildung in Information 20 zeigt den Zusammenhang der Objekte der oben eingeführten Klassen in Form eines Mengendiagramms. Das Diagramm besagt, dass z.B. die Menge der `LectureExercise`-Objekte in der Menge der `Lecture`-Objekte enthalten ist, die ihrerseits Untermenge der `Course`-Objekte ist. Diese Mengenbeziehung zwischen Objekten der Oberklasse und Unterklasse wird durch die "ist ein"-Beziehung ausgedrückt.

- Einer Objektvariablen der Oberklasse kann ein Objekt der Unterklasse zugewiesen werden
 - diese Typflexibilität gilt auch für Parameterübergaben
 - Typkonversion zur Unterklasse möglich

```
Course a = new Lecture();
...
if (a instanceof Lecture)
  Lecture b = (Lecture) a;
```

- Zwei Arten von Typen bei Variablen zu unterscheiden

(1) statischer Typ

- Typ, in der die Variable deklariert wurde
- im Beispiel: statischer Typ von a ist Course

(2) dynamischer Typ

- Typ des Objekts, auf das die Variable zur Laufzeit zeigt
- im Beispiel: dynamischer Typ von a ist nach der Zuweisung Lecture

Information 21: Statischer und dynamischer Typ

Die Kompatibilitäts-Eigenschaft bewirkt, dass einer Objektvariablen, die als Typ der Oberklasse deklariert wurde, ein Objekt der Unterklasse zugewiesen werden kann, wie der Programmausschnitt in Information 21 verdeutlicht.

Hieraus resultiert, dass bzgl. der Frage, von welchem Typ diese Variable ist, zwischen einem statischen Aspekt, der durch die Deklaration gegeben ist und einem dynamischen Aspekt, der durch die Zuweisung ggf. kompatibler Typen zur Laufzeit festgelegt wird, unterschieden werden muss.

4.3 Dynamische Bindung

Im Zusammenhang mit der Polymorphie stellt sich unmittelbar die Frage, die Methode welcher Klasse aufgerufen wird, wenn eine polymorphe Variable eine entsprechende Meldung zum Aufruf einer Methode erhält.

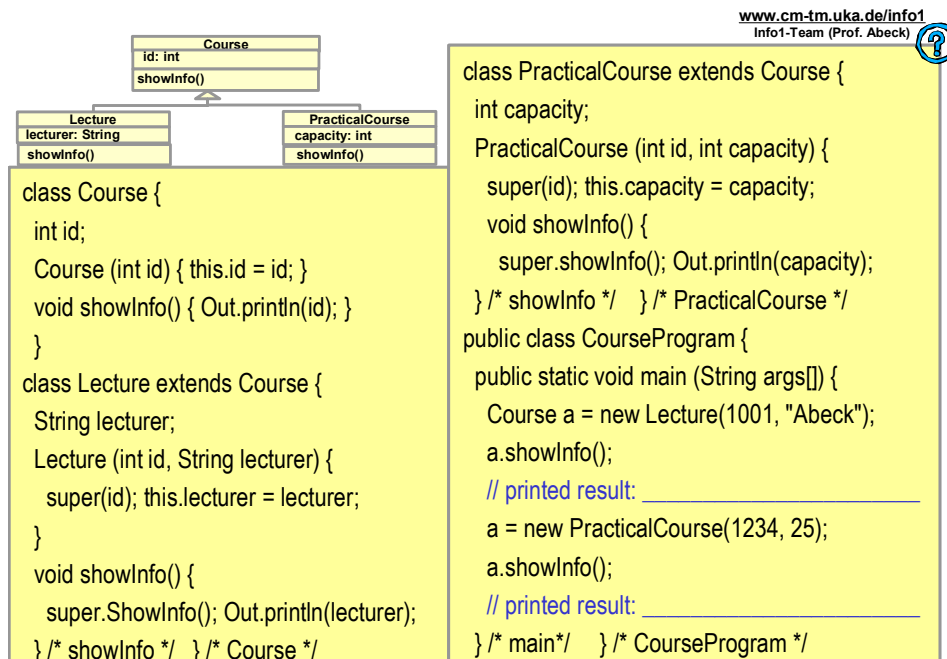
- Eine Meldung `obj.m()` führt immer zum Aufruf der `m()`-Methode, die zum dynamischen Typ von `obj` gehört
 - die Meldung wird also dynamisch (d.h. zur Laufzeit) gebunden
- Vorteile
 - der Aufrufer muss sich nicht darum kümmern, von welchem Typ eine polymorphe Variable gerade ist
 - es können Unterklassen ergänzt werden, ohne den Teil des Programms ändern zu müssen, durch den der Aufrufer realisiert wird

Information 22: Dynamische Bindung

Die Antwort lautet, dass der dynamische Typ der Variablen zugrunde gelegt wird; die Variable wird dynamisch – also zur Laufzeit – an die Klasse gebunden. Im Beispielprogramm von

Information 21 heißt das konkret, dass eine Meldung `a.showInfo()` zum Aufruf der `showInfo()`-Methode der Klasse `Lecture` führt.

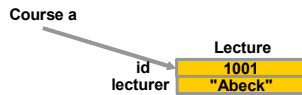
Die dynamische Bindung hat zwei große Vorteile bei der Erstellung von Programmen, die mit polymorphen Objekten umgehen, die in Information 22 beschrieben sind.



Interaktion 11: Beispiel zur dynamischen Bindung von Methodenaufrufen

Das Programm in Interaktion 11 greift das zuvor eingeführte Beispiel zu den Lehrveranstaltungen (Klassen `Course`, `Lecture`, `PracticalCourse`) in reduzierter Form auf. Neben der dynamischen Bindung der Objektvariablen `a` in der Methode `main()` wird in den Klassendefinitionen zu `Lecture` und `PracticalCourse` der Zugriff auf die Konstruktor-Methode mittels `super(id)` und die Methode `showInfo()` mittels `super.showInfo()` verdeutlicht.

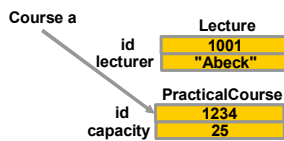
- Wirkungsweise der Befehle der main()-Methode des Programms CourseProgram:



```
Course a = new Lecture (1001, "Abeck");
```

- Dynamische Bindung bewirkt, dass die Methode zur Klasse Lecture aufgerufen wird

```
a.showInfo();
```



```
a = new PracticalCourse(1234, 25);
```

- Dynamische Bindung bewirkt, dass die Methode zur Klasse PracticalCourse aufgerufen wird

```
a.showInfo();
```

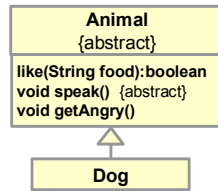
Information 23: Veranschaulichung des Ablaufs

In Information 23 wird die Wirkungsweise der in der main()-Methode enthaltenen Befehle graphisch veranschaulicht. Die Zuweisungen zur Variablen a der Klasse Course zeigen, dass die Variable auf Objekte von Unterklassen zu Course – hier sind das die Klassen Lecture und PracticalCourse – zeigen kann.

Wird die überladene Methode showInfo() aufgerufen, so bewirkt die dynamische Bindung, dass nicht die Methode der (statisch durch Deklaration bestimmten) Klasse Course, sondern die Methode der (dynamisch zur Laufzeit zugewiesenen) Klassen Lecture bzw. PracticalCourse aufgerufen wird.

4.4 Abstrakte Klassen und Interfaces

In Vererbungshierarchien kann der Fall auftreten, dass man zu einer Superklasse die Schnittstelle, aber nicht deren Implementierung vollständig angeben kann. Zu einer solchen Superklasse soll auch kein Objekt erzeugt werden können, d.h., die Klassenbeschreibung soll ausschließlich als Grundlage für die hieraus abgeleiteten Unterklassen dienen.



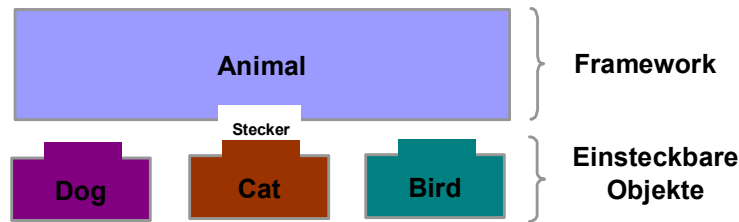
- Eine abstrakte Klasse ist dadurch gekennzeichnet, dass
 - keine Objekte zu dieser Klasse erzeugt werden können
 - diese eine Schnittstelle zu zukünftigen Unterklassen definiert

<pre> abstract class Animal { boolean likes (String food) { return false; } abstract void speak(); void getAngry() { speak(); } } class Dog extends Animal { boolean like (String food) { return food.equals("bones"); } void speak() {Out.println("bark");} void getAngry() {Out.println("growl");} } </pre>	<pre> public class AnimalProgram { public static void main (String args[]) { Animal a = new Dog(); if (a.like("corn")) a.speak(); else a.getAngry(); // printed result: _____ if (a.like("bones")) a.speak(); else a.getAngry(); // printed result: _____ } } </pre>
---	--

Interaktion 12: Abstrakte Klassen

Besitzt eine Klasse die beschriebenen Anforderungen, so ist sie als so genannte abstrakte Klasse zu deklarieren. Im UML-Klassendiagramm werden abstrakte Klassen durch {abstract} unter dem Klassennamen gekennzeichnet. Die Sprache Java sieht das Schlüsselwort `abstract` vor, das vor dem Schlüsselwort `class` anzugeben ist.

Diejenigen Methoden, zu denen in einer abstrakten Klasse keine Implementierungen erfolgen, sind ebenfalls mit diesem Schlüsselwort zu kennzeichnen. Im Beispiel in Interaktion 12 wird die Methode `speak()` der abstrakten Klasse `Animal` als abstrakte Methode definiert. Abstrakte Methoden müssen zwingend in den Unterklassen überschrieben werden. Im Beispiel überschreibt die Unterklasse `Dog` nicht nur diese Methode, sondern auch die beiden anderen (nicht abstrakten) Methoden `like()` und `getAngry()`.



- Eine abstrakte Klasse definiert eine Familie von Klassen, die alle eine bestimmte Menge von Meldungen verstehen
 - eine Variable der abstrakten Klasse wirkt wie ein Steckplatz, in die die Objekte dieser Familie eingesteckt werden können
- Eine Software mit solchen Steckplätzen wird als ein Framework bezeichnet
 - sind Halbfabrikate, die durch Einstecken von Objekten zu verschiedenen Endfabrikaten ausgebaut werden können

Information 24: *Framework-Konzept*

Die Abbildung in Information 24 verdeutlicht den Zusammenhang zwischen der abstrakten Klasse (hier *Animal*) und den daraus abgeleiteten Unterklassen (hier *Dog*, *Cat* und *Bird*). Die von der abstrakten Klasse vorgegebenen Methoden stellen eine Art Stecker (Plug) dar, in den die Objekte der Unterklassen eingesteckt werden können, um die unvollständige abstrakte Klasse zu vollenden.

Eine Software, die gemäß diesem Steckprinzip aufgebaut ist, wird als *Framework* (Rahmenwerk) bezeichnet.

- Abstrakte Klassen können neben abstrakten Methoden auch nicht-abstrakte Methoden enthalten
 - sind alle Methoden abstrakt, entspricht die abstrakte Klasse einem Interface (Klassenschnittstelle)
- Ein Interface kann neben abstrakten Methoden Konstanten-Attribute enthalten, aber keine Variablen-Attribute und keine Konstruktoren

```
interface Writer {
    int eol = '\n';
    void open();
    void close();
    void write(char ch);
}
```

- Alle Methoden eines Interface sind automatisch öffentlich (public) und abstrakt (abstract)
- Alle Attribute sind automatisch öffentlich (public) und konstant (static final)

- Ein Interface kann zu einem "Unter-Interface" erweitert werden


```
interface ReadWriter extends Writer { char read(); }
```

Information 25: *Interfaces*

Eine spezielle Ausprägung von abstrakten Klassen stellen die *Interfaces* (Schnittstellen) dar. Die besonderen Eigenschaften von *Interfaces* sowie ein Beispiel interface *Writer* sind in Information 25 beschrieben.

Eine Klasse kann von einem *Interface* (Schnittstellen-Klasse) erben und muss in diesem Fall Implementierungen zu allen Methoden des Interfaces liefern, da *Interface*-Methoden gemäß Definition abstrakt sind.

Interfaces sind spezielle Klassen und bieten daher auch die Möglichkeit der Erweiterung, also der Bildung von "Unter-*Interfaces*".

www.cm-itm.uka.de/infot
Prof. Abeck & Info1-Team 

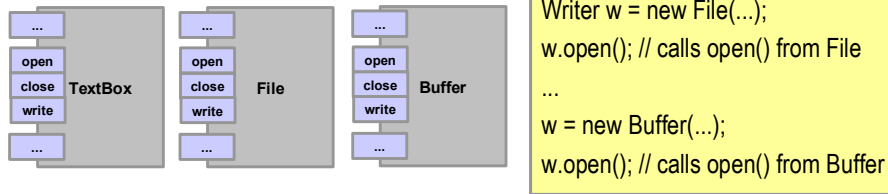
<pre>class GUIObject { int x, y, width, height; public void resize (int w, h) { ... }; ... } class TextBox extends GUIObject implements Writer { public String text; private StringBuffer buffer; ... public void open() { buffer = new StringBuffer(); } public void close() { text = buffer.toString(); } public void write(char ch) {buffer.append(ch); } }</pre>	<pre>... Writer w; w = new TextBox(); w.open() // dynamic binding w.write('o'); w.write('k'); w.close() if (w instanceof TextBox) TextBox t = (TextBox) w; w.resize(10,20); // correct? t.resize(10, 20); // correct? ...</pre>
---	---

Interaktion 13: Beispiel zur Nutzung des *Interface* *Writer*

Im Beispiel in Interaktion 13 erbt (extends) die Klasse *TextBox* von der Klasse *GUIObject* und implementiert (implements) das Interface *Writer*. Alle *Interface*-Methoden von *Writer* müssen in *TextBox* implementiert werden. Die Implementierung muss dabei nicht zwingend direkt in der Klasse erfolgen (wie das im Beispiel gegeben ist), sondern kann auch durch Erben von einer anderen Klasse erfolgen.

Da *Interfaces* Typen sind, kann man auch entsprechende Variablen deklarieren (z.B. *Writer* *w*). Da ein *Interface* eine spezielle Form einer abstrakten Klasse darstellt, verhält sich eine Klasse, die ein *Interface* implementiert, wie dieses *Interface* (Polymorphie). Im Beispiel ist es daher gestattet, ein *TextBox*-Objekt in einer *Writer*-Variablen zu speichern (*w = new TextBox()*). Eine Meldung an dieses *TextBox*-Objekt (z.B. *w.open()* oder *w.write()*) wird entsprechend dynamisch gebunden. Der Zugriff auf Methoden, die in der *TextBox*-Klasse definiert wurden oder die von anderen Klassen geerbt wurden, ist nur nach geeigneter Typ-Konversion möglich. Dieser Aspekt ist bei der Beantwortung der in Interaktion 13 gestellten Fragen zu berücksichtigen.

- Interfaces ermöglichen die Gleichbehandlung von Klassen, die in keiner Vererbungsbeziehung stehen



- Eine Klasse kann beliebig viele Interfaces implementieren

```
class Buffer extends StorageObject implements Writer, Comparable, Serializable { ... }
```

- Im Gegensatz zur Mehrfach-Implementierung von Interfaces ist die Mehrfach-Vererbung kritisch und wird daher in Java nicht unterstützt
 - Grund: Zwei Basisklassen könnten Methoden mit gleichem Namen und gleicher Schnittstelle aufweisen
 - Kein Problem bei Interfaces, weil die Methode in der Unterklasse neu implementiert werden muss

Information 26: Einsatzzweck von Interfaces

Interfaces vererben somit nur den Typ und nicht die eigentliche Implementierung. Hierdurch lassen sich Klassen im Sinne der Polymorphie gleich behandeln, die in keiner direkten Beziehung – insbesondere in keiner Vererbungsbeziehung – stehen. Information 26 verdeutlicht diesen Sachverhalt an einem Beispiel mit drei Klassen `TextBox`, `File` und `Buffer`.

Interfaces bieten außerdem eine Möglichkeit, die Eigenschaft auszudrücken, dass eine Klasse Eigenschaften von mehreren Klassen erbt. Diese als Mehrfach-Vererbung (*Multiple Inheritance*) bezeichnete Eigenschaft ist in Java aufgrund des in Information 26 genannten Problems nicht erlaubt.

- Aufbau und Einsatz von Frameworks
 - die sinnvolle Verwendung von Vererbung ist bis heute noch nicht abschließend geklärt
- Komponentenbasierte Programmierung
 - gezielte Nutzung von Interfaces (Beispiel: JavaBeans)
- Entwurfsmuster (Patterns)
 - schematische Lösung häufig auftretender Probleme in Softwarearchitekturen mittels objektorientierter Techniken
- Weitere Java-Sprachmittel (z.B. geschachtelte Klassen)
- Klassenbibliotheken (z.B. zur Unterstützung von Applets, XML, ...)

Information 27: Ausblick auf weitere Themen der Objektorientierung

Die behandelten Konzepte zur Vererbung liefern eine gute Grundlage, dieses mächtige aber auch nicht ganz unproblematische objektorientierte Konzept in der Programmierung vorteilhaft

zu nutzen. Information 27 macht deutlich, dass die Beschreibung dieses Kapitels über die Vererbung bei weitem nicht vollständig ist.

Es sei abschließend angemerkt, dass das Konzept der Objektorientierung nicht nur in der Programmierung, sondern verstärkt auch in der Modellierung genutzt wird, wie in der Kurseinheit GRUNDBEGRIFFE DER INFORMATIK [C&M-GI] ausgeführt wurde. In zahlreichen weiterführenden Veranstaltungen, die sich mit dem für die Informatik zentralen Themen der Modellierung und der Softwareentwicklung beschäftigen, wird die Objektorientierung aufgegriffen und weiter vertieft.

VERZEICHNISSE

Abkürzungen und Glossar

Abkürzung oder Begriff	Langbezeichnung und/oder Begriffserklärung
Dynamische Datenstruktur	Datenstruktur die sich dynamisch zur Laufzeit aufbauen und manipulieren lässt. Beispiele: Liste, Baum, Graph
einfach verkettet	Typ einer Liste, die einen Zeiger auf den Nachfolger vorsieht (aber keinen Zeiger auf den Vorgänger).
<i>Framework</i>	Eine Software, die so aufgebaut ist, dass Software-Baustein gemäß einem auf dem Vererbungsprinzip basierenden Steckprinzip ergänzt werden können. Deutscher Begriff: Rahmenwerk
Interface	Eine spezielle Ausprägung einer abstrakten Klasse, die abstrakte Methoden und Konstanten-Attribute beinhaltet.
<i>Information Hiding</i>	Zentrales Prinzip der Objektorientierung, durch das die internen Datenstrukturen einer Klasse gegenüber der Außenwelt versteckt wird. Deutscher Begriff: Geheimnisprinzip
ISWA	INTERNET-SYSTEME UND WEB-APPLIKATIONEN Titel einer Lehrveranstaltung.
K&D	KOMMUNIKATION UND DATENHALTUNG Titel einer Lehrveranstaltung.
Klasse	Einheit bestehend aus Daten (Attribute) und darauf arbeitende Operationen (Methoden). Klassen stellen eine geeignete Form zur Gliederung komplexer Software-Systeme dar.
Konstruktor	Spezielle Klassenmethode, die bei der Objekterzeugung dazu genutzt werden kann, ein Objekt in der gewünschten Form zu initialisieren.
LIFO	<i>Last In First Out</i> Ein auch als Kellerprinzip bezeichnetes Zugriffsprinzip, das besagt, dass das zuletzt geschriebene Element (<i>Last In</i>) bei der nächsten Leseoperation ausgegeben wird (<i>First Out</i>).
null	Nullpointer Wert, der besagt, dass auf keinen Inhalt gezeigt wird. Beispiel: Eine bereits deklarierte Klassenvariable, der aber noch kein Objekt zugewiesen wurde, hat den Wert null.
Objektattribut	Einem speziellen Objekt zugeordnetes Attribut, das erst zur Zeit der Objekterzeugung angelegt wird.
Objektorientierte	Philosophie des Programmierens, die von einer Welt ausgeht, die aus

Programmierung	gleichberechtigten und einheitlich erscheinenden Objekten besteht. Oberstes Prinzip des objektorientierten Vorgehens ist es, Objekte stets nur von außen zu betrachten und ihren inneren Aufbau zu ignorieren [CS93].
Polymorphie	Eine im Zusammenhang mit der Objektorientierung auftretende Vielgestaltigkeit in der Form, dass eine Objektvariable der Unterklasse so auftreten kann, als wäre sie ein Objekt der Oberklasse. Die Objektvariable kann also in Gestalt verschiedener Klassen auftreten.
Stapel	Bezeichnung einer Rechenstruktur, der das LIFO-Prinzip (Last In First Out) zugrunde liegt. Synonymer Begriff: Keller Englischer Begriff: <i>Stack</i>
UML	<i>Unified Modeling Language</i> Sprache, die aus graphischen Elementen besteht und zur semi-formalen Beschreibung von beliebigen Gegenständen (z.B. Software-Systeme oder Geschäftsbereiche) genutzt wird.
Vererbung	Objektorientiertes Prinzip, das eng mit dem Klassenbegriff verknüpft ist und eine von verschiedenen Arten von Beziehungen zwischen Klassen darstellt, durch die Attribute und Operationen einer Klasse an eine Unterklasse weitergegeben („vererbt“) werden.
Virtuelle Maschine	Ein in Software realisierter Prozessor.

Index

dynamische Datenstrukturen	15	Konstruktoren	11
einfach verkettet	20	LIFO	14
Framework	30	Nullpointer	null 4
Information Hiding	22	Objektattribute	12
Interfaces	31	objektorientierte Programmierung	3
INTERNET-SYSTEME	UND	Polymorphie	25
APPLIKATIONEN	7	Stapel	13
Java Virtual Machine	15	UML-Klassendiagramm	4
KOMMUNIKATION	UND	Unified Modeling Language	23
DATENHALTUNG	7	Vererbung	22

Informationen und Interaktionen

Information 1: OBJEKTORIENTIERTE PROGRAMMIERUNG	3
Information 2: KLASSEN - Wesentliche Eigenschaften und Beispiel	3
Information 3: Deklaration und Verwendung von Klassen	4
Information 4: Typkompatibilität	5
Information 5: Objekt als Rückgabewert einer Methode	6
Information 6: Klassen und Arrays	6
Information 7: OBJEKTORIENTIERUNG - Daten und Methoden als eine Einheit	8
Information 8: Methodenaufruf	10
Information 9: Klassenbezogene und objektbezogene Attribute und Methoden	12
Information 10: Klassen- und objektbezogene Elemente in der Klasse Fraction	12

Information 11: Spezielle Klassenmethode main()	13
Information 12: Stapel (Keller, <i>Stack</i>) als Beispiel einer Klasse	13
Information 13: UML-Klassendiagramm und Java-Programm zu <i>Stack</i>	14
Information 14: Verketteten von Knoten	17
Information 15: Listen.....	18
Information 16: Einfügen in eine sortierte Liste	20
Information 17: Schnittstelle und <i>Information Hiding</i>	22
Information 18: VERERBUNG - Einführung und Beispiele.....	23
Information 19: Methoden und Vererbungsbeziehung	24
Information 20: Kompatibilität zwischen Oberklasse und Unterklasse	25
Information 21: Statischer und dynamischer Typ	26
Information 22: Dynamische Bindung.....	26
Information 23: Veranschaulichung des Ablaufs.....	28
Information 24: <i>Framework</i> -Konzept	30
Information 25: <i>Interfaces</i>	30
Information 26: Einsatzzweck von Interfaces	32
Information 27: Ausblick auf weitere Themen der Objektorientierung.....	32
Interaktion 1: Objektgleichheit und Attributgleichheit	5
Interaktion 2: Kombination von Klassen und Arrays.....	7
Interaktion 3: Beispiel-Klasse <i>Fraction</i>	9
Interaktion 4: Versteckter Parameter <i>this</i> im Methodenaufruf	10
Interaktion 5: Konstruktormethoden	11
Interaktion 6: Verwendung von Stapeln.....	15
Interaktion 7: DYNAMISCHE DATENSTRUKTUREN - Die wichtigsten Arten.....	16
Interaktion 8: Suchen eines Listenelements	19
Interaktion 9: Stapel als verkettete Liste	21
Interaktion 10: Modellieren und Programmieren von Vererbungsbeziehungen	23
Interaktion 11: Beispiel zur dynamischen Bindung von Methodenaufrufen.....	27
Interaktion 12: Abstrakte Klassen	29
Interaktion 13: Beispiel zur Nutzung des <i>Interface Writer</i>	31

Literatur

- [C&M-GI] Cooperation&Management: GRUNDBEGRIFFE DER INFORMATIK, Kursdokument zur Vorlesung "INFORMATIK I", <http://www.cm-tm.uka.de/info1>, Universität Karlsruhe (TH), C&M (Prof. Abeck).
- [C&M-IP] Cooperation&Management: IMPERATIVE PROGRAMMIERUNG, Kursdokument zur Vorlesung "INFORMATIK I", <http://www.cm-tm.uka.de/info1>, Universität Karlsruhe (TH), C&M (Prof. Abeck).
- [C&M-ISWA] Cooperation&Management, Kursdokumente zur Vorlesung "INTERNET-SYSTEME UND WEB-APPLIKATIONEN (ISWA)", <http://www.cm-tm.uka.de/iswa>, Universität Karlsruhe (TH), C&M (Prof. Abeck)
- [C&M-KuD] Cooperation&Management, Kursdokumente zur Vorlesung "KOMMUNIKATION UND DATENHALTUNG (K&D)", <http://www.cm-tm.uka.de/iswa>, Universität Karlsruhe (TH), C&M (Prof. Abeck)
- [C&M-PG] Cooperation&Management: PROGRAMMIERGRUNDLAGEN, Kursdokument zur Vorlesung "INFORMATIK I", <http://www.cm-tm.uka.de/info1>, Universität Karlsruhe (TH), C&M (Prof. Abeck).

- [CS93] Duden "Informatik" – ein Sachlexikon für Studium und Praxis, Dudenverlag, 1993.
- [Ma89] Udi Manber: Introduction to Algorithms – A Creative Approach, Band 1: Programmierung und Rechstrukturen, Addison Wesley, 1989.
- [Me90] Bertrand Meyer: Objektorientierte Softwareentwicklung, Carl Hanser Verlag, 1990.
- [Mö03] Hanspeter Mössenböck: Sprechen Sie Java? – Eine Einführung in das systematische Programmieren, dpunkt.verlag 2003.
- [Oe01] Bernd Oestereich: Objektorientierte Software-Entwicklung – Analyse und Design mit der Unified Modeling Language, Oldenbourg Verlag, 2001.