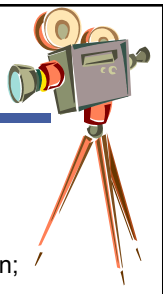
Teil 1: Abstraktion, Vertragsmodell, Fehlerbehandlung, Polymorphie

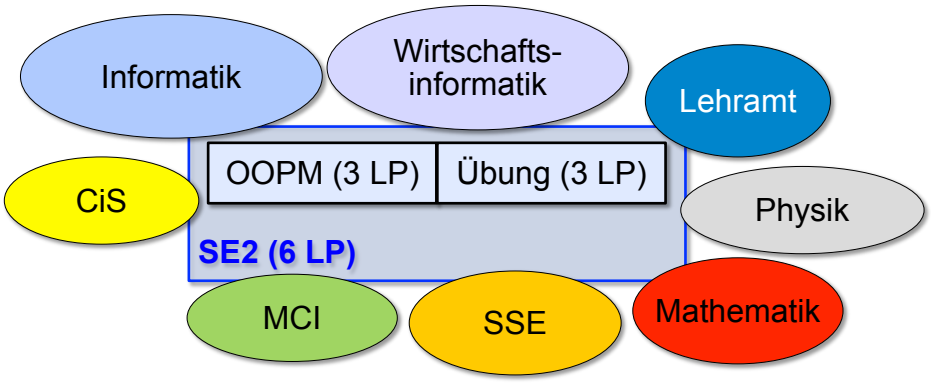
Videoaufzeichnung



- Es findet eine Videoaufzeichnung **aller SE2-Vorlesungstermine** statt. Hierfür möchten wir auf einige Punkte aufmerksam machen:
- Gefilmt werden die **Vortragenden**, nicht das Plenum!
- Nur der **Ton des Vortragenden** wird über ein Funkmikrofon aufgenommen; aber **Hintergrundgeräusche** könnten trotzdem hörbar sein!
- **Fragen aus dem Plenum werden nicht gesondert aufgenommen.** Der Vortragende wiederholt die Fragen sowohl für das Plenum als auch für das Video.
- Die **Folien** werden ebenfalls zeitlich synchronisiert dem Video hinzugefügt.
- Die Vorlesung wird nach der Vorlesung **auf Lecture2Go veröffentlicht**.

- **Verlasst euch nicht darauf**, dass ihr innerhalb einer bestimmten Frist alle Vorlesungen zum Download vorfinden werdet!

Auch diesmal wieder: viele Studiengänge



…und weitere

Ihr Einsatz

- Wir erwarten über die 3 Präsenzstunden hinaus, dass ihr euch wöchentlich auf die Aufgaben im Team vorbereitet:
  - 2 Stunden Vorlesung
  - 3 Stunden Übung
  - 3 Stunden Vor- und Nachbereitung
  - = **8 Stunden pro Woche für SE 2**

Der Organisator der SE2-Übung

Christian Späh

SE2 CommSy SoSe 2015

se-orga@informatik.uni-hamburg.de

Übersicht über die SE2-Übungen (1)

- Beginn der Übungswoche jeweils **donnerstags**.
- Karfreitag und Ostermontag fallen aus. TeilnehmerInnen müssen nach Möglichkeit einen anderen Termin in dieser Übungswoche besuchen.
- Wochen 1 bis 5: **Laborphase 1** wie in SE1
  - 3-stündiger Präsenzbetrieb in Rechnerräumen des RZ
  - 5 wöchentliche Übungsblätter
  - Arbeit in Paaren
  - **Programmierdiktat**
- Wochen 6 bis 13: **Laborphase 2**
  - Erfolgreicher Abschluss der Laborphase 1 ist Voraussetzung
  - 3-stündiger Präsenzbetrieb, auch in Rechnerräumen des RZ
  - Vier 14-tägige Übungsblätter
  - Arbeit in Teams (4 Studierende)

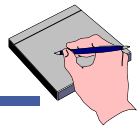
Die SE2-Übungswoche

Stand: 31.3.2015

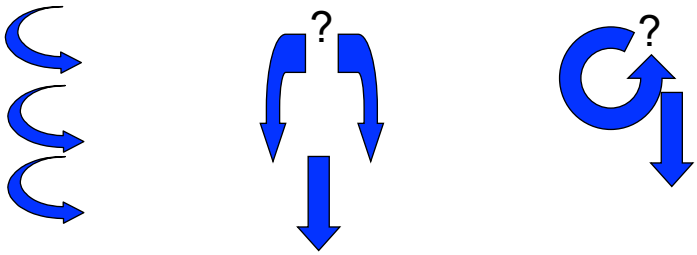
Vorausgesetzte Inhalte aus SE1

- Objekte, Klassen, Operationen
- Datenfelder, Methoden, Konstruktoren, Parameterübergabe
- Strukturierte Programmierung (Sequenz, Auswahl, Wiederholung)
- Variablen und Typen, Ausdrücke, Zuweisungen
- Basistypen und Referenztypen
- Grundelemente der Modellierung mit UML
- Klasse als Typ
- Funktionale Dekomposition, Rekursion
- Umgang mit APIs, Schnittstellen, Interfaces
- Sammlungen benutzen:
  - Listen, Mengen, Abbildungen
  - Iterieren, Gleichheit und Identität
- Sammlungen implementieren:
  - Arrays, verkettete Strukturen
- Fehlersuche/Debugging
- Testen: Modultests, Regressionstests

Ablaufsteuerung durch Kontrollstrukturen



- Ablaufsteuerung in **Programmiersprachen** durch **Kontrollstrukturen**:
  - Die Ausführungsreihenfolge der Anweisungen einer Methode entspricht zunächst der textlichen Anordnung (**Sequenz**). Davon kann aber abgewichen werden. Dazu gibt es spezielle Mechanismen der Ablaufsteuerung:
    - **Fallunterscheidung**
    - **Wiederholung**

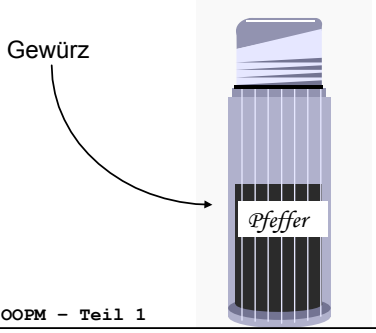


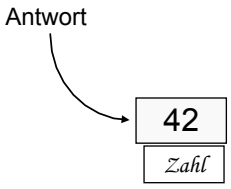
SE2 – OOPM – Teil 1 9

Imperative Variablen

Der Begriff Variable ist grundlegend für das Verständnis imperativer Sprachen:

- Eine Variable ist eine Abstraktion eines physischen Speicherplatzes.
- Sie hat einen Namen (häufig auch: Bezeichner), über den sie angesprochen werden kann.
- Eine Variable hat den Charakter eines Behälters:
  - Sie hat eine Belegung (ihren aktuellen Inhalt), die sich ändern kann;
  - und einen Typ, der Wertemenge sowie zulässige Operationen und weitere Eigenschaften festlegt.

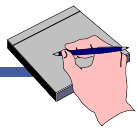




Die Typen sind hier *Pfeffer* und *Zahl*.

SE2 – OOPM – Teil 1 10

Der Typbegriff (1. Definition)



Im Zusammenhang mit Programmiersprachen hat der Begriff Typ oder (oft auch) Datentyp eine zentrale Bedeutung:

- „Unter einem Datentyp versteht man die Zusammenfassung von Wertebereichen und Operationen zu einer Einheit."

  [Informatik-Duden]

Dies bedeutet:

- Für jeden Typ ist nicht nur die Wertemenge definiert, sondern auch die Operationen, die auf diesen Werten zulässig sind.

Java-Beispiele:

Datentyp: `int`
Wertemenge: { -2^{31} … $2^{31}-1$ }
Operationen: ganzzahlig Addieren, ganzzahlig Multiplizieren, …

Datentyp: `boolean`
Wertemenge: { wahr, falsch }
Operationen: Und, Oder, …

SE2 – OOPM – Teil 1 11

Typprüfung

- Wenn jeder Variablen (und Konstanten), jedem Literal und jedem Ausdruck in einem Programm ein fester, nicht änderbarer Typ zugeordnet ist, nennt man dies **statische Typisierung**.
- Als Folge der Typisierung kann für programmiersprachliche Ausdrücke geprüft werden, ob sie „korrekt typisiert" sind, d.h. ob die einzelnen Komponenten einen passenden Typ besitzen, und ob dem Ausdruck insgesamt ein definierter Typ zugeordnet werden kann. Diese Prüfung nennt man **Typprüfung**.
- In **statisch typisierten Sprachen** (wie Java, C#, C++, Pascal, Eiffel) prüft der Compiler dies zur Übersetzungszeit.

Smalltalk ist eine dynamisch typisierte Programmiersprache, in der Variablen nicht mit einem Typ deklariert werden.

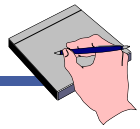
Dynamisch typisierte Sprachen gestatten nur eine Laufzeitprüfung.

Beispiel: Die Addition ist als binäre Operation auf zwei `int` Zahlen definiert, nicht aber für eine `int` Zahl und einen Wahrheitswert.

```
int sum = 12 + 6;
int result = 12 + false; // Typfehler!
```

SE2 – OOPM – Teil 1 12

Merkmale unserer ersten Klasse



```
class Girokonto
{
  private int _saldo;

  public void einzahlen( int betrag )
  {
    _saldo = _saldo + betrag;
  }
}
```

- Java-Programme bestehen aus Klassen (hier: `Girokonto`).
- Die Klasse definiert eine Methode (hier: `einzahlen`).
- Die Methode erhält einen Parameter (hier: `betrag` vom Typ `int`) und hat keinen Rückgabewert (hier: Schlüsselwort `void`).
- Im Rumpf der Methode wird ein Wert einem Zustandsfeld zugewiesen (hier: `_saldo`).
- Das Feld muss deklariert sein (hier vom Typ `int`).
- Alternativ nennen wir die Felder in einer Klassendefinition auch Exemplarvariablen.

SE2 – OOPM – Teil 1 13

Abgleich mit den Prinzipien der Objektorientierung

```
class Girokonto
{
  private int _saldo;

  public void einzahlen( in
  {
    _saldo = _saldo + betra
  }
}
```

- Das Verhalten eines Objekts ist durch seine angebotenen Dienstleistungen (Methoden) bestimmt.
  - ✓ `einzahlen` ist durch `public` für Klienten aufrufbar.
- Die Realisierung dieser (zusammengehörigen) Dienstleistungen (als Methoden) ist verborgen.
  - ✓ Kein Zugriff durch Klienten auf die Implementierung von `einzahlen`
- Ebenso sind die Zustandsfelder als interne Strukturen eines Objekts gekapselt.
  - ✓ Das Feld `_saldo` ist durch `private` vor externem Zugriff geschützt.
- Auf den Zustand eines Objektes kann nur über seine Dienstleistungen zugegriffen werden.
  - ✓ Hier durch `einzahlen`

SE2 – OOPM – Teil 1 14

Die Doppelrolle einer Klasse

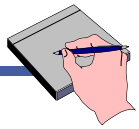
- Aus Sicht der **Klienten** einer Klasse ist interessant:
  - Welche **Operationen** können an Exemplaren der Klasse aufgerufen werden?
  - Welchen Typ haben die **Parameter** einer Operation und welches **Ergebnis** liefert sie?
  - Was sagt die **Dokumentation** (Kommentare, javadoc) über die **Benutzung**?

Außensicht, öffentliche Eigenschaften, Dienstleistungen, Schnittstelle

- Für die **Implementation** der Methoden einer Klasse ist relevant:
  - Wie sind die Operationen in den **Methodenrümpfen** umgesetzt?
  - Welche **Exemplarvariablen/Felder** definiert die Klasse?
  - Welche **privaten Hilfsmethoden** stehen in der Klasse zur Verfügung?

Innensicht, private Eigenschaften, Implementation

Zentrale Eigenschaften von Interfaces



- Interfaces ...
  - sind Sammlungen von **Methodenköpfen**. Alle Methoden in einem Interface sind implizit `public`;
  - enthalten **keine Methodenrümpfe** (also keine Anweisungen);
  - definieren **keine Felder**;
  - sind **nicht instanziierbar** - es können keine Exemplare von Interfaces erzeugt werden;
  - werden von Klassen implementiert.

Randnotiz: In Java können in einem Interface auch Konstanten definiert werden (mit den Modifikatoren `static final`).

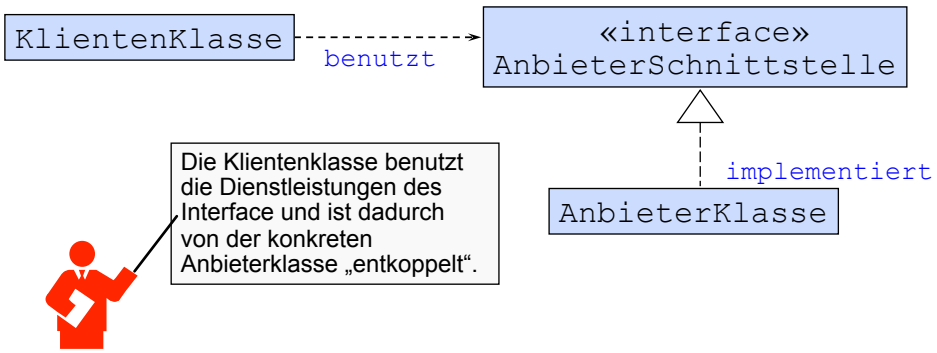
Interfaces werden durch Klassen implementiert

- Eine Klasse kann deklarieren, dass sie ein Interface implementiert.
- Die Klasse muss dann für jede Operation des Interfaces eine entsprechende Methode anbieten. Sie „erfüllt“ dann das Interface.

```
class KontoImpl implements Konto
{
    private int _saldo;

    public void einzahlen(int betrag) {...}
    public void auszahlen(int betrag) {...}
    public int gibSaldo() {...}
}
```

- An allen Stellen, an denen ein Objekt mit einem bestimmten Interface erwartet wird, kann eine Referenz auf ein Exemplar einer implementierenden Klasse verwendet werden.

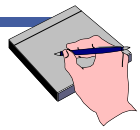
Trennung von Schnittstelle und Implementation mit Interfaces



Klassen und Interfaces definieren Typen

- Jede **Klasse** definiert in Java einen **Typ**:
  - durch ihre Schnittstelle (Operationen)
  - durch die Menge ihrer Exemplare (Wertemenge)
- Ein **Interface** definiert in Java ebenfalls einen **Typ**:
  - durch seine Schnittstelle
  - durch die Menge der Exemplare aller Klassen, die dieses Interface erfüllen, d.h. die die Schnittstelle des Interface implementieren.
- Für einen **Typ im objektorientierten Sinne** ist also wichtig:
  - welche Objekte gehören zur Wertemenge des Typs,
  - welche Operationen sind auf diesen Objekten zulässig
  - und **nicht**, wie die Operationen implementiert sind.

Der erweiterte objektorientierte Typbegriff (1. Fassung)



- Der **klassische Typbegriff**:
  - Ein Typ definiert eine Menge an Werten, die eine Variable oder ein Ausdruck annehmen kann.
  - Jeder Wert gehört zu genau einem Typ.
  - Die Typinformation ist statisch aus dem Quelltext ermittelbar.
  - Ein Typ definiert die zulässigen Operationen.
- Der **erweiterte objektorientierte Typbegriff**:
  - Ein Typ definiert das Verhalten von Objekten durch eine Schnittstelle, ohne die Implementation der Operationen und des inneren Zustands festzulegen.
- Folge:
  - Ein Objekt wird von **genau einer** Klasse erzeugt.
  - Da eine Klasse auch mehrere Interfaces erfüllen kann, kann ein Objekt **zu mehr als einem** Typ gehören.

Statischer und dynamischer Typ (I)

- Durch den erweiterten objektorientierten Typbegriff muss der statische vom dynamischen Typ einer **Referenzvariablen** unterschieden werden.
- Der **statische Typ einer Variablen** wird durch ihren deklarierten Typ definiert. Er heißt statisch, weil er zur Übersetzungszeit feststeht.

```
Konto k;  // Konto ist hier der statische Typ von k
```

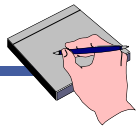
- Der statische Typ legt die **Operationen** fest, die über die Variable aufrufbar sind.

```
k.einzahlen(200);  // einzahlen ist hier eine Operation
```

- Ein Compiler kann bei der Übersetzung prüfen, ob die genannte Operation tatsächlich im statischen Typ definiert ist.

Hinweis: Dies alles gilt sinngemäß auch für Ausdrücke, deren Ergebnis ein Referenztyp ist.

Statischer und dynamischer Typ (II)



- Der **dynamische Typ einer Referenzvariablen** hängt von der Klasse des Objektes ab, auf das die Referenzvariable zur Laufzeit verweist.

```
k = new KontoSimpel();  // dynamischer Typ von k: KontoSimpel
```

- Er bestimmt die Implementation und ist dynamisch in zweierlei Hinsicht:
  - Er kann erst zur Laufzeit ermittelt werden.
  - Er kann sich während der Laufzeit ändern.

```
k = new KontoAnders();  // neuer dynamischer Typ von k: KontoAnders
```

- Ein **Objekt** hingegen **ändert seinen Typ nicht**; es bleibt sein Leben lang ein Exemplar seiner Klasse.
- Der dynamische Typ einer Variablen (bzw. der Typ des referenzierten Objektes) entscheidet darüber, welche **konkrete Methode** bei einem **Operationsaufruf** ausgeführt wird. Da diese Entscheidung **erst zur Laufzeit** getroffen werden kann, wird dieser Prozess **dynamisches Binden** (einer Methode) genannt.

Umgang mit einer Liste

- Aus Sicht des Klienten einer Liste sind relevant:
  - Eine Liste kann **beliebig viele Elemente** enthalten.
  - Über den Index kann direkt auf **beliebige Positionen** in der Liste zugegriffen werden.
  - Das **Einfügen** eines Elements **erhöht** den Index der nachfolgenden Elemente.
  - Das **Entfernen** eines Elements **verringert** den Index der nachfolgenden Elemente.
  - Häufig wird die Information benötigt, ob ein gegebenes Element bereits in der Liste enthalten ist („**Test auf Enthaltensein**“).

Das Interface `List` (relevanter Ausschnitt)

Indexbasierter Zugriff auf die Elemente

Indexbasierte Modifikatoren

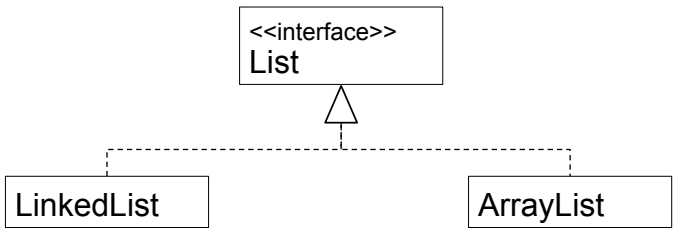
Bestimmung eines Element-Index

Bildung von Teillisten

```
public interface List<E>
extends Collection<E>
{
  // Alle Operationen wie in Set
  ...

  // zusätzlich: indexbasierte Operationen
  E get(int index);
  E set(int index, E element);
  void add(int index, E element);
  E remove(int index);

  int indexOf(Object o);
  int lastIndexOf(Object o);

  List<E> subList(int from, int to);
}
```

Listen-Implementationen im Java Collections Framework

- Der Umgang mit einer Liste ist im JCF mit dem Interface **`List`** modelliert.
- Das JCF bietet zwei Implementationen für dieses Interface:
  - **`LinkedList`**
  - **`ArrayList`**
- **`LinkedList`** basiert auf dem Konzept **verkettete Liste**.
- **`ArrayList`** basiert auf dem Konzept **wachsender Arrays**.

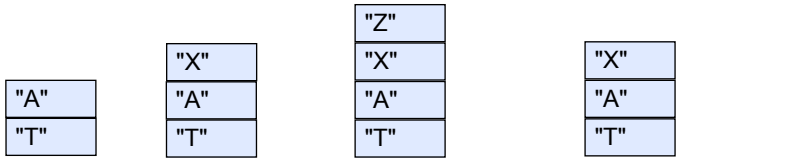


Ein Stack-Interface

```
interface Stack<E>
{
  void push(E element);
  E pop();
  boolean isEmpty();
}
```

Das Interface Stack spezifiziert zwar die Signaturen der Stack-Operationen, legt aber nicht die gewünschte Organisationsform der Elemente (LIFO) fest.

gegeben: `Stack<String> s` mit zwei Elementen.



`s.push("X");` `s.push("Z");`

```
String t = s.pop();
// t == "Z"
```


Teil 1: Abstraktion, Vertragsmodell, Fehlerbehandlung, Polymorphie

SE2 2015: OOPM-Vorlesung und Übung

	W	Vorlesung	Übung
L5	1	VL01: **01.04.** Einführung, **Wrap-UP von SE I**	**Blatt 01 02.04.** Eclipse, Umstieg von BlueJ, Interfaces
L5	2	VL02: **08.04.** Objektorientierte Tests, Vertragsmodell	**Blatt 02 09.04.** Debugger, Vertragsmodell, Test First
L5	3	VL03: **15.04.** Polymorphie und Typhierarchien	**Blatt 03 16.04.** Subtyp-Polymorphie, Typhierarchien
L6	4	VL04: **23.04.** Implementationsvererbung	**Blatt 04 23.04.** Implementationsver., Schablonenm.
	5	VL05: **29.04.** Fehlerbehandlung, Exceptions; Module (Pakete)	**Blatt 05 30.05.** Exceptions, Pakete
L6	6	VL06: **06.05.** OO Analyse und Modellierung (WAM)	**Blatt 06 07.05.** **Entwurf im Kleinen, Service u. Material (Mediathek)**
L7	7	VL07: **13.05.** Strukturierung von Anwendungssystemen (WAM light)	**18.05.**
L7	8	VL08: **20.05.** Entwurfsmuster insb. Beobachter	**Blatt 07 01.06.** **Beobachtermuster (Kinosystem)**
L7	9	VL09: **03.06.** GUI-Programmierung	**08.06.**
L7	10	VL10: **10.06.** Objektorientierter Entwurf	**Blatt 08 15.06.** **Werkzeuge, GUI mit Swing (Kinosystem)**
L8	11	VL11: **17.06.** Werte und Objekte, Fachwerte	**22.06.**
L8	12	VL12: **24.06.** Refactoring Modellierung & Abstraktion (hz)	**Blatt 09 29.06.** **Fachwerte (Kinosystem)**
L8	13	VL13: **01.07.** Korrektheit, abstrakte Datentypen	**06.07.**
L8	14	VL14: **09.07.** Rückblick auf Laborphase 2, Lehreevaluation	

SE2 – OOPM – Teil 1 27