Algorithmen und Datenstrukturen 2
SS 98

7. Programmieren in Java
8. Datenstrukturen
9. Algorithmenkonstruktion 2

Die AuDII-Mannschaft

- Dozentin
  - Prof. Dr. Martina Zitterbart, Sprechstunde: Mi. 14.30 - 15.30 Uhr
- Assis
  - Axel Böger, Sprechstunde: Di. 15.00 - 16.00 Uhr
  - Oliver Brand, Sprechstunde: Do.10.00 - 11.00 Uhr
- Hiwis
  - Oliver Kamphenkel
  - Andreas Schoolmann
  - Anke Speer
- Klausur
  - voraussichtlich ???

Unterlagen

- Folien
  - wie immer: im Web
    - http://www.ibr.cs.tu-bs.de/lehre/ss98
  - Bemerkungen
    - Sie werden auf den Folien an der ein oder anderen Stelle Lücken finden. Das ist Absicht. Die fehlenden Teile werden in der Vorlesung erarbeitet.
- Bücher
  - Goos; *Vorlesungen über Informatik - Band 2: Objektorientiertes Programmieren und Algorithmen*, Springer-Verlag
  - Bemerkungen
    - Die Vorlesung orientiert sich in wesentlichen Teilen an diesem Buch. Allerdings sind die Programmierbeispiele in Java gehalten und nicht in Modula 2.
    - Über Algorithmen gibt es jede Menge weiterer Bücher, in die man einen Blick riskieren kann und sollte.

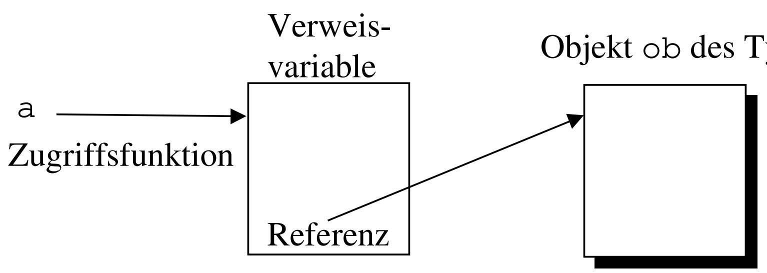
7. Programmieren in Java

7.1 Programmieren mit Objekten

- Variable oder Konstante (Größen) bereits folgendermaßen eingeführt
  - Referenz
    - unveränderliches, eindeutiges Kennzeichen einer Variablen.
    - In imperativen Sprachen i.A. durch Namen, in Assemblersprachen durch Adresse.
  - Behälter
    - eigentliche Variable, typischerweise realisiert durch einen Speicherbereich.
  - Wert
    - Inhalt des Behälters zu einem bestimmten Zeitpunkt, z.B. eine Zahl
- Zugriffsfunktionen
  - Funktion zur Berechnung der Referenz und zum Schreiben oder Lesen des Wertes einer Variablen (bei Konstanten nur Lesen)
- Eine Zuweisung `x = y` schreibt eine Kopie des Wertes `y` in den Behälter `x`
- Hierbei handelt es sich um *Wertsemantik*
  - Größen werden als Wertobjekte bezeichnet
  - Typen werden als Werttyp bezeichnet
  - Referenzen treten nicht explizit in Erscheinung
- Problem: bei vielen Aufgaben ist die Anzahl der Objekte aus der Aufgabenstellung nicht bekannt
  - Reihungen fester Längen und Zugriff durch Indizieren nicht angemessen

Referenzsemantik

- Referenzobjekte
  - Objekte mit Referenzsemantik
  - Typ dieser Objekte: Referenztyp
  - Beispiel
    - `T a`
    - der Wert von `a` ist eine Referenz für das (Referenz-)Objekt `ob` des Typs `T`



- Objekt `ob`
  - kann nicht unmittelbar mit einem Namen versehen werden
  - Referenzobjekte sind anonym
- Objektorientierte Sprachen
  - verwenden im allgemeinen Referenzsemantik
  - Wertsemantik nur für relativ einfache Objekte
- Referenzobjekte sind für den Aufbau dynamischer Datenstrukturen geeignet

Dynamische Variable

- Dynamische Variable
  - werden mit der Anweisung **`new`** erzeugt
  - haben keinen Namen
  - der Bezug auf den Wert geschieht mit Referenzen

- Beispiel
  - Arrays werden mit **`new`** erzeugt
  - Grenzen können zur Laufzeit festgelegt werden
    - ➩ `new x();`

      ----> *Konstruktor*
      - ➩ es wird ein Objekt der Klasse `x` erzeugt und eine Referenz darauf zurückgeliefert
      - ➩ Referenzen sind implementierungstechnisch Adressen
      - ➩ **`null`** ist einzige Konstante; sie gehört zum Wertebereich aller Referenzen

- Dynamische Klasseninstanzen ("Objekte")
  - können in Java erzeugt werden
  - hiermit lassen sich komplexe Datenstrukturen aufbauen, die sich in Struktur und Größe während des Programmablaufs ändern können

Datenverbunde und Referenzen

- ```
  class Person {
    String name;
    int geburtsjahr;
    int groesse;
    Person vater, mutter;
  }
  ```
  - ➩ `vater`, `mutter` sind hier Platzhalter für Referenzen auf andere Personen (genauer: auf Objekte des Typs `Person`)

- ```
  class Test {
    public static void main (String args[]) {
          Person p1, p2, p3;
          p1.name = "Hugo";
          System.out.println(p1.name);
    }
  }
  ```
  - ➩ was passiert hier?

- Was sind hier Referenzvariablen?

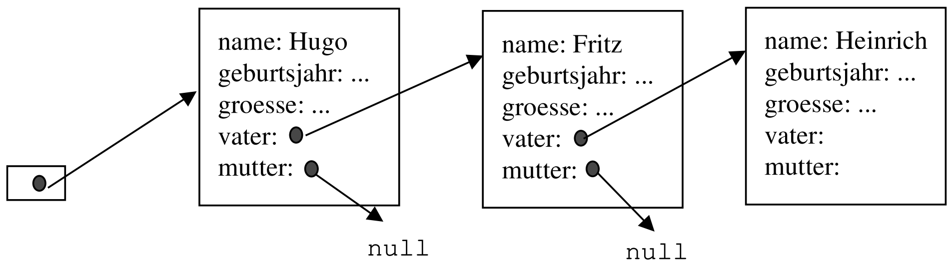
Rekursive Datenverbunde

- Zweck könnte etwa die Speicherung eines Stammbaumes sein ...

```
p1.name = "Hugo";
p1.vater.name = "Fritz";
p1.vater.vater.name = "Heinrich";
p1.vater.vater.vater.name = ...
```

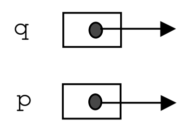
- Ist das ok? Wie sieht es mit dem zu reservierenden Speicherplatz aus?

➩ Es entsteht die folgende verkettete Struktur:

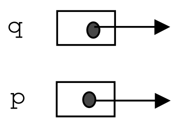


Verändern von Referenzvariablen

```
Person p, q;
p = new Person();
```




```
p = null;
```



und was ist hier das Ergebnis?

```
q = p;
p = null;
```

Zuweisung von Referenzen

- Einzige Operation auf Referenzvariablen ist die Zuweisung
  - ```
    p = new Person(); q = new Person();
    p.groesse = 186; p.name = "Mike";
    q.groesse = 170; q.name = "Sandra";
    ```
  - `p = q;`
    - ➩ wie sieht die Situation nun aus?
  - Was sind jeweils die Ausgaben?
    - ➩ `System.out.println(p.name);`
    - ➩ `System.out.println(q.name);`
    - ➩ `p.groesse = 200;`
    - ➩ `System.out.println(q.groesse);`

Vergleich von Referenzen

- Referenzvariablen können auf Gleichheit / Ungleichheit getestet werden
  - Wie?
  - `p == q` ist genau dann `true`, wenn die beiden Referenzvariablen auf das gleiche Objekt zeigen (z.B. nach `p = q`) oder wenn beide gleich `null` sind.
  - `p == q` ist `false`, wenn zwei verschiedene Objekte, auf die `p` bzw. `q` verweisen, die gleichen Werte haben.
  - ```
    p = new Person(); q = new Person();
    p.groesse = 186; p.name = "Mike";
    q.groesse = 170; q.name = "Sandra";
    if (p == q)           ... liefert welches Ergebnis?

    p = q;
    if (p == q)           ... liefert welches Ergebnis?

    p = new Person(); q = new Person();
    p.groesse = 186; p.name = "Mike";
    q.groesse = 186; q.name = "Mike";
    if (p == q)           ... liefert welches Ergebnis?
    ```

Parameterübergabe von Referenzen

- Referenzen werden by value übergeben
  - wo werden Änderungen am Objekt bzw. an der Referenz vorgenommen?

```
class elem {
  int i=5;
}
static void f(elem x) {
  x.i = 7;
  x = null;
}
static elem g(elem x) {
  x = null;
  return (x);
}
public static void main (String args[]) {
  elem e=new elem();
  f(e);
  if (e==null) Systen.out.println("null");
          else System.out.println("nicht null");
  System.out.println(e.i);
  e = new elem();
  e = g(e);
  if (e==null) System.out.println("null");
          else System.out.println("nicht null");
}
```

Ergebnis?

Ergebnis?

Ergebnis?

Strukturierungskonzepte für Software

- Strukturierung ist wesentlich für die Beherrschung eines komplexen Vorgangs oder Systems
  - also auch für Entwurf, Implementierung, Weiterentwicklung, Wartung ... großer Softwaresysteme
- Strukturierung ist auch in anderen Disziplinen wesentlich, z.B. Bauingeneurwesen
  - Strukturierung des Entwicklungsprozesses ("Hausbau")
  - Strukturierung des Produktes ("Haus")
- Auch die Softwaretechnik strukturiert beides

"Klassische" Softwarestrukturierungskonzepte

- Typisierung von Daten und Variablen
  - Variable eines bestimmten Typs darf nur Werte dieses Typs enthalten
- Kontrollstrukturen
  - Strukturierung der Reihenfolge von Teilaufgaben
  - Typische Muster durch while, for, if, ... adäquat realisieren
- Parametrisierte Prozeduren und Funktionen (bzw. Methoden in Java)
  - Wiederverwendung von Lösungen für Teilaufgaben
  - "Prozedurale Abstraktion"
  - Unterstützt Top-Down bzw. Bottom-Up-Entwurf

Weitere Softwarestrukturierungskonzepte

- Abstrakte Datentypen
  - gekapselte Einheiten mit Zugriffsmethoden
  - Integritätsschutz durch Information-Hiding
- Modularisierung

7.2 Modularisierung

- Systeme bestehen aus Teilsystemen (Moduln); Module haben eine klare Grenze (Schnittstelle)
  - Module gibt es in vielen Bereichen, z.B. in der Natur, in der Gesellschaft, ...
- Ziele bei der Modularisierung
  - Erhöhung der Übersichtlichkeit
    - Zusammensetzung von Systemen aus geringer Anzahl von Teilsystemen.
    - Menschen haben Schwierigkeiten, Systeme schnell zu erfassen, die aus mehr als 7 Teilsystemen bestehen
    - Komplexität befindet sich in den Teilsystemen
  - Arbeitsteilung
    - Modulschnittstellen sind Kooperationsabsprachen
  - Unterscheidung Funktion/Implementierung
    - Wir unterscheiden die Rolle von demjenigen, der sie ausführt, z.B. im Theater
    - Die Rolle entspricht einer Schnittstelle, der Mensch implementiert sie
  - Lokalisierung der Entscheidung
    - Jedes Teilsystem besitzt eigenen Zustand. Wechsel des Teilsystems ändert lokal den Zustandsraum, hat aber global keinen Einfluß
  - Wiederverwendung
    - Erfolgreiche Module können in anderen Systemen wieder eingesetzt werden

Anforderungen an den Modulbegriff

- Rekursivität
  - Ein Teilsystem kann sich wiederum aus Teilsystemen zusammensetzen
- Unterscheidung Schnittstelle/Implementierung
  - Schnittstelle ist die Außenansicht eines Moduls. Sie kann auf verschiedene Arten implementiert werden
- Geheimnisprinzip
  - information hiding
  - Implementierung einer Schnittstelle ist von außen unsichtbar
  - von außen kann die korrekte Arbeitsweise eines Moduls nicht gestört werden
    - Seiteneffekte vermeiden
    - globale Variable vermeiden
- Zusammengehörigkeit
  - in einem Modul sollen logisch/funktional zusammengehörige Funktionen gekapselt werden
  - Standardbeispiel: ADT-Implementierung
- Schwache Kopplung
  - zwischen Modulen sollte so wenig Information wie möglich ausgetauscht werden

Typische Eigenschaften eines Moduls

- ❑ Baustein einer größeren Einheit
- ❑ Besitzt einen sinnvollen Namen
- ❑ Vereinigt eine Menge zusammengehöriger Komponenten
- ❑ Realisiert eine klar abgegrenzte Teilaufgabe
- ❑ Enthält lokale Anweisungen / Routinen / Daten
- ❑ Information Hiding
- ❑ Hat möglichst explizite Import-/Exportschnittstellen
  - kann damit Komponenten anderer Module benutzen
  - stellt damit Komponenten anderer zur Verfügung
  - Benutzt-Relation
- ❑ Ist unabhängig entwickelbar und übersetzbar
- ❑ Kann in einer Modulbibliothek gespeichert werden
- ❑ Kann ggf. instanziiert werden

Information Hiding

- ❑ Modul besitzt
  - private Komponenten
  - öffentliche Komponenten
- ❑ Nur öffentliche Komponenten sind von außen sichtbar. Sie bilden damit die Exportschnittstelle
  - Prinzip:

Modularisierung: Sprachunterstützung

- ❑ Idealerweise sollte der Modulbegriff durch die Programmiersprache unterstützt werden
  - Erleichtert die Programmierung
  - Compiler ermöglicht
    - ➩ getrennte Übersetzung
    - ➩ Test auf Schnittstellenverletzung

- ❑ Beispiele
  - Pascal
    - ➩ nur funktionale / prozedurale Aspekte
  - Modula
    - ➩ "module" kommt Idealvorstellung nahe
  - Objektorientierte Sprachen
    - ➩ Klassen und Methoden
      - – Beispiele: Simula, C++, Java
      - – statische Klassen (mit klassenbasierten Variablen und Methoden) im Gegensatz zu dynamisch instanziierbaren Objekten
      - – Klassenkonzept geht über die Modularisierung hinaus

Zufallszahlen-Modul

- ❑ Zufallszahlen werden oft benötigt. In der Informatik verwendet man Pseudo-Zufallszahlengeneratoren. Diese erzeugen beispielsweise alle Integers in einem gewissen Intervall nach einem irregulären Muster und simulieren so gleichverteilte Zufallszahlen.
  - Tatsächlich ergibt sich die nächste Zufallszahl deterministisch aus der letzten. Die erste Zufallszahl wird "echt" zufällig gewählt, etwa aus der Uhrzeit (Sekundenzeiger).
  - Die bekanntesten Pseudo-Zufallszahlengeneratoren arbeiten nach dem sogenannten linearen Kongruenzverfahren.
    - ➩ Dabei ergibt sich die nächste Zufallszahl folgendermaßen aus der letzten

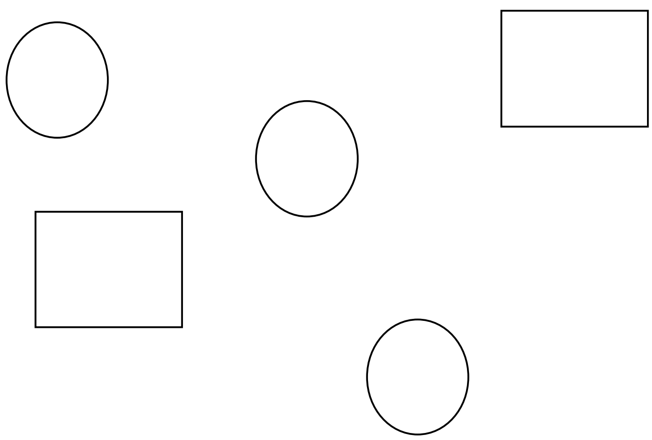
      mit Primzahl m und geeigneter natürlicher Zahl a.
    - ➩ Dies erzeugt Zufallszahlen im Intervall [0, m-1]
    - ➩ Für ein anderes Intervall [r, s-1] transformiert man
    - ➩ m heißt Periode des Generators. Die Wahl von a ist kritisch, um die volle Periode zu erreichen und zahlentheoretisch nicht trivial.
    - ➩ Beispiele:
      - ➩ a=9, m=17: 7, 12, 6, 3, 10, 5, 11, 14, 7
      - ➩ a=10, m=17: 7, 2, 3, 13, 11, 8, 12, 1, 10, 15, 14, 4, 6, 9, 5, 16, 7

Zufallszahlen-Modul II

- Standard-Zufallszahlengenerator von Park and Miller
  - verwendet $m = 2^{31} - 1$ (Mersennsche Primzahl)
  - erzeugt alle positiven Zahlen einer Länge von 32 Bit
  - $a = 16807$
  - erzeugt für $x_0 \,!= 0$ alle Zahlen $0 < x < 2^{31} - 1$ genau einmal (volle Periode)
- Problem
  - Zwischenergebnis $a\, x_n$ ist länger als 32 Bit
- Trick
  - $a\, x \bmod m = a\, (x \bmod q) - r\, (x \bmod q) + c(x)\, m$
    - ⇨ wobei $m = a * q + r$, wobei $q = m$ div a, $r = m$ mod a und
    - ⇨ $c(x)n = \begin{cases} 1 & \text{falls } \mathrm{a(x \ \ mod \ \ q) - r(x \ \ div \ \ q) < 0} \\ 0 & \text{sonst} \end{cases}$
- Wie sieht ein entsprechendes Java-Programm aus?

7.3 Objektorientiertes Programmieren

- Weltsicht
  - Die Welt besteht aus verschiedenen interagierenden "Dingen", die sich klassifizieren lassen

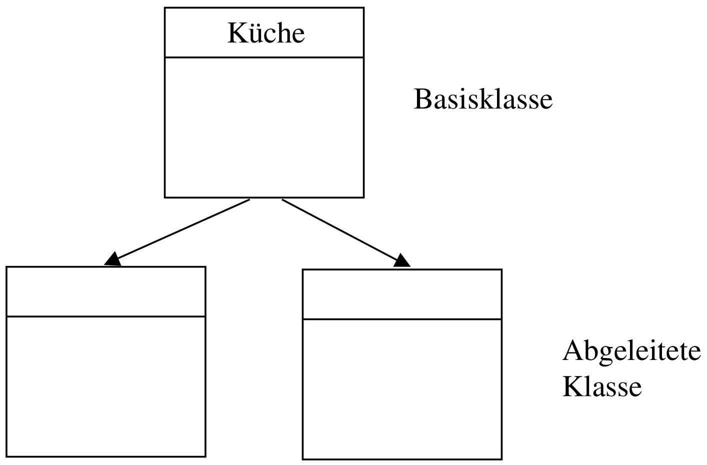


- Ziel
  - Betrachteten Weltausschnitt strukturkonsistent mit kommunizierenden Objekten abbilden und modellieren
- Objekte sind:

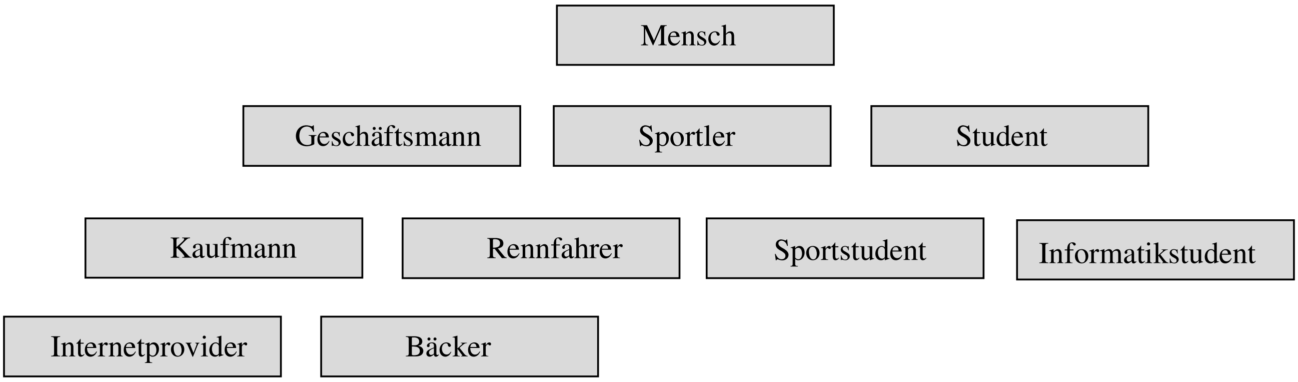
Objektorientierter Softwareentwurf

- Strukturierung der Problemlösung in eine Menge kooperierender Objekte als Ausprägungen von ADTs
- Entwurf der Objekttypen, dabei ähnliche Objekte zu Klassen zusammenfassen
- Herausfaktorisierung gemeinsamer Aspekte verschiedener Klassen ==> Hierarchie festlegen, Klassenbibliothek
- Festlegung einzelner Dienstleistungen (--> Methoden)
- Entwurf der Objektbeziehungen
- Feinplanung der einzelnen Methoden, Festlegung der Klassenattribute etc.
- Strukturierung und Implementierung der Methoden
  - klassische Programmierung im Kleinen

Vererbung



Vererbung ("Inheritance")



- Spezialisierung, Verallgemeinerung
- Alle Merkmale, Eigenschaften des umfassenderen Begriffs werden auf den Unterbegriff vererbt
- Vererbungsrelation ist transitiv
- Erstellen des Klassenbaums ist wichtiger Bestandteil der objektorientierten Programmierung

7.4 Einige Details zu Java

- Gemeinsame Basisklasse: `java.lang.Object`
  - wird bei der Deklaration keine `extends`-Klausel angegeben, so ergänzt der Java-Compiler automatisch `extends java.lang.Object`
  - Von Object werden folgende Methoden angeboten
    - ⇨ `boolean equals(Object o), String toString(), Object clone(), Class getClass(), int hashCode(), finalize(), notify(), notifyAll(), wait(long ms), wait(long ms, int ns), wait()`
    - ⇨ Die Methoden sollten in den Unterklassen geeignet überschrieben werden
- Containerklassen
  - realisieren bestimmte grundlegende Daenstrukturen (z.B. Stack oder Liste). Jedes Element dieser Datenstrukturen enthält eine Referenz auf ein `Object`. In die Containerklasse können beliebige Java-Objekte eingetragen werden.
- Wrapper-Klassen
  - In `Object` können keine elementaren Datentypen eingetragen werden (z.B. `double`). Hierfür sind sogenannte Wrapper-Klassen erforderlich.
  - Beispiel: Wrapper-Klasse `Double`
    - ⇨
```
Double d_obj = new Double(3.1415);   // einpacken
double d = d_obj.doubleValue();  // auspacken
```

Schnittstellen

- Schnittstelle
  - Definiert Methoden und Konstanten
  - Kann andere Schnittstellen erweitern (beliebige Anzahl)
  - Eine Implementierung muß alle enthaltenen Methoden realisieren
- Schnittstelle als Pseudocode
- Beispiel einer Schnittstelle
- ersetzen Zeiger auf Funktionen