#### Informatik II

Vorlesung am D-BAUG der ETH Zürich

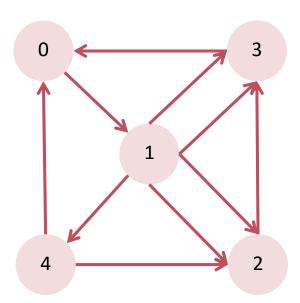
Vorlesung 4, 14.3.2016

Pagerank, Klassen, Datenkapselung

### Zufallssurfer, Pagerank

#### Modell

- Zufallssurfer startet auf einer beliebigen Seite  $s_0 \in \{0..n-1\}$  des www und klickt sich von dort weiter auf andere verlinkte Seiten.
- Bei jedem seiner Klicks wird jeder ausgehende Link mit gleicher Wahrscheinlichkeit angewählt.
- Darüber hinaus gibt es eine Grundwahrscheinlichkeit d, dass mit einer beliebigen Seite neu gestartet wird.
- Gibt es keinen ausgehenden Link, so wird wieder gleichverteilt von einer beliebiger Seite gestartet.



# Fallstudie: Der Zufallssurfer, Pagerank

#### Frage:

Wie gross ist die Wahrscheinlichkeit, dass er nach sehr langer Zeit auf Seite

x landet?

→ Welche Seite wird langfristig oft angeklickt?

# Die Übergangsmatrix des Random Surfers

Für jede Entscheidung des Zufallssurfers gilt

$$p_{ij} \coloneqq \mathbb{P}(s_t = j \mid s_{t-1} = i)$$

$$= \begin{cases} d \cdot \frac{1}{n} + (1 - d) \cdot \frac{l_{ij}}{c_i} & \text{falls } c_i \neq 0 \\ \frac{1}{n} & \text{sonst} \end{cases}$$

#### wobei

- $l_{ij}$  Anzahl Links von i nach j und
- $c_i$  Anzahl ausgehende Links von i

# Übergangsmatrix (analog zur Wanderameise):

 $\left(p_{ij}\right)$  lässt sich als Matrix darstellen und es ergibt sich im vorliegenden Beispiel, mit d=0.1

W't von der Seite 1 startend auf die Seite 4 zu wechseln

$$\mathbf{P} := (p_{ij})_{0 \le i,j < n} = \begin{bmatrix} .02 & .92 & .02 & .02 & .02 \\ .02 & .02 & .38 & .38 & .20 \\ .02 & .02 & .02 & .92 & .02 \\ .92 & .02 & .02 & .02 & .02 \\ .47 & .02 & .47 & .02 & .02 \end{bmatrix}$$

Jede Zeile ist ein Wahrscheinlichkeitsvektor

### Zwei getrennte Aufgaben

#### 1. Generieren der Übergangsmatrix

- a) Überlegen eines Eingabemodells, Einlesen der Daten
- b) Berechnen der Übergangsmatrix

```
public static double[][] InputLinks(java.util.Scanner scanner)
```

#### Simulation

"Separation of Concerns"

- a) Überlegen wie pro Zeile vorgegangen wird
- b) Simulation durch repetitive Anwendung auf jeweilige Zeile

```
public static double[] Simulate(double[][] p, int iterations)
```

# Eingabemodell

#### Benötigt:

- 1. Anzahl Seiten
- 2. Links Quelle → Ziel
- 3. Terminierung

```
Eingabe:
                         1. Anzahl Seiten
                       2. Folge von Paaren
                          Quelle → Ziel
1 2 1 2 1 3 1 3 1 4
2 3
  0
4 0 4 2
end
                         3. Terminierung:
```

Kein Integer

#### Einlesen der Links

```
public static double[][] InputLinks(java.util.Scanner scanner) {
  int N = scanner.nextInt();  // Anzahl Seiten
  int[][] counts = new int[N][N];  // counts[i][j] = #links Seite i >> j
  // Akkumulieren der link counts
  while (scanner.hasNextInt()){
    int i = scanner.nextInt();
    int j = scanner.nextInt();
    outDegree[i]++;
                                                             Spezifikation der
    counts[i][j]++;
                                                                Kanten
                                        0 1
                                        1 2 1 2 1 3 1 3 1 4
                                                             Übergangsmatrix
                                        2 3
                                                               berechnen
                                        3 0
                                        4 0 4 2
```

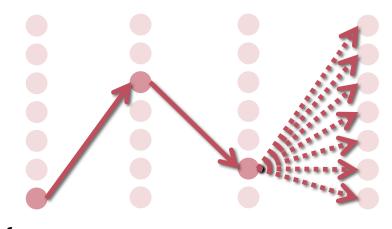
end

**Simulation** 

# Berechnung der Übergangsmatrix

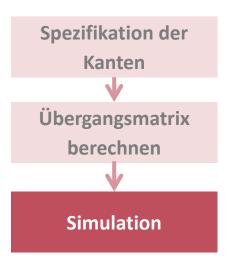
```
public static double[][] InputLinks(java.util.Scanner scanner) {
  ... // Einlesen: vorige Folie
  double[][] matrix = new double[N][N];
  for (int i = 0; i < N; i++) { // W'tsverteilung Zeile i
     for (int j = 0; j < N; j++) { // W'tswert Spalte j
       if (outDegree[i] > 0)  // Seite i hat ausgehende Kanten
          matrix[i][j] = 0.9*counts[i][j]/outDegree[i] + 0.1/N;
                                    // Seite i hat keine ausgehende Kanten
       else
          matrix[i][j] = 1.0/N;
                                                                      Spezifikation der
                                                                         Kanten
                                                                      Übergangsmatrix
  return matrix;
                                                                        berechnen
                                                                        Simulation
```

#### Simulation der Kette



```
public static double[] Simulate(double[][] p, int iterations) {
  int N = p.length;
  // start auf Seite 0
  int page = 0;
  for (int t = 1; t <= iterations; t++){
    page = UnfairDice(p[page]);
    freq[page]++;
  for (int i = 0; i < N; ++i)
    freq[i] /= iterations;
  return freq;
```

Eigentliche Kernaufgabe sehr einfach



# Übrigens: MCMC

Die Simulation, die wir oben ausgeführt haben, ist die Simulation einer **Markov-Kette** 

Zufallsbasierte Simulationen nennen wir ja auch Monte-Carlo Verfahren.

Hier haben wir es also sogar mit einem Markov Chain Monte Carlo (MCMC) Verfahren zu tun!

MCMC wird vor allem dann verwendet, wenn der Zustandsraum sehr sehr gross ist und man kein anderes Verfahren anwenden kann.

# Übrigens: Page Rank

Die Frequenzen, die wir pro Seite berechnen sind ein Mass für die "Beliebtheit" einer Seite und werden als *Pagerank* bezeichnet.

Der Algorithmus von Google zur Auswahl der angezeigten Seiten basiert auf der effizienten Berechnung des Pageranks.

Da das Problem ein Eigenwertproblem ist (siehe Ameisenbeispiel), basiert(e) das Geschäftsmodell von Google auf der Lösung eines Eigenwertproblems für sehr grosse Matrizen.

Klassen und Objekte, Dynamische Speicherallokation, Überladen, Klassen als Datencontainer, Datenkapselung, Klassen, Fallstudie Online Statistik

#### 4. KLASSEN

# Klassen und Objekte

An die Stelle von RECORDs in Pascal treten bei Java die Klassen

Pascal	Java
RECORDs in Pascal sind reine <b>Datenobjekte</b> . Auf ihnen wird mit  Prozeduren operiert.	Klassen in Java beherbergen <b>Daten und Code</b> . Sie stellen einen  Teil des Codes bereit, mit dem auf ihnen operiert wird.
RECORDs sind wertesemantische Typen: Instanzen werden automatisch "in place" alloziert.	Klassen in Java haben Referenzsemantik: Instanzen müssen mit "new" alloziert werden. Instanzen heissen Objekte.

#### Automatische Speicherallokation: Der Stack

Teil der dynamischen Speicherallokation bereits bekannt: Stack (Aufrufstapel).

- Beim Aufruf von Prozeduren / Funktionen / Methoden wird automatisch
   Speicher für lokale Variablen und Parameter angelegt, welcher beim Rücksprung wieder freigegeben wird.
- Die Struktur dieses Speichers ist wegen der Unmöglichkeit des gleichzeitigen Aufrufs mehrerer Prozeduren besonders einfach: ein Stapel
- Um Verwaltung und Aufbau des Aufrufstapels müssen wir uns glücklicherweise nicht kümmern.

#### Der Stack: Beispiel

```
static int fac(int n){
       if (n>1)
              return n*fac(n-1);
       else
              return 1;
                                                                          Zustand des
                                                                            Stapels
System.out.println(fac(4));
                                                                                  24
                                       fac(4)
                                            fac(3)
                                                 fac(2)
                                                      fac(1) ret 1
                                                                ret 2
                                                                      ret 6 ret 24 println(24)
```

### Dynamische Speicherallokation

Für die Implementation von dynamischen Datenstrukturen (später!) und für die Nutzung von Klassen benötigt man

dynamischen Speicher,

also Speicher den man explizit anfordern muss.

Bei Java wird der Speicher, sobald nicht mehr verwendet, von einem *Garbage Collector* abgeräumt.

#### Allokation mit new

Semantik: neuer Speicher für ein Objekt vom Typ *T* wird angelegt Wert des Ausdrucks ist die Adresse des Objekts

**new** 
$$T(par_0, ..., par_n)$$
 Ausdruck vom Typ  $T$ 

(Optionale) Parameter werden dem Konstruktor des Typs weitergegeben

```
java.util.Scanner scanner = new java.util.Scanner(System.in);
```

### Dynamische Speicherallokation

Mit **new** erzeugte Objekte haben dynamische Speicherdauer. Sie leben, bis sie nicht mehr erreichbar sind.

```
String a = new String("Hallo");
String b = new String ("Welt");
System.out.println(a + " " + b); // "Hallo Welt"
b = new String("Leute"); // alter String "Welt" nicht mehr erreichbar
System.out.println(a + " " + b); // "Hallo Leute"
```

Zum Glück müssen wir uns um freigegebene Objekte nicht kümmern. Der Garbage Collector räumt "hinter uns" auf.

# Überladen (Overloading)

Methoden sind durch ihren Namen im Gültigkeitsbereich ansprechbar.

Es ist sogar möglich, mehrere Methoden des gleichen Namens zu definieren.

Die richtige Version wird aufgrund der Signatur der Funktion ausgewählt.

## Overloading

Die *Signatur* einer Methode ist bestimmt durch Art, Anzahl und Reihenfolge der Argumente

```
static double sq(double x) { ... }
static int sq(int x) { ... }
static double log(double x, double b) { ... }
static double log(double x) { return log(x,2); }
```

Der Compiler wählt beim Funktionsaufruf automatisch die Funktion, welche "am besten passt"

```
System.out.println(sq(3.3));
System.out.println(sq(3));
System.out.println(log(4,10));
System.out.println(log(3));
```

# Overloading

Mit dem Überladen von Funktionen lassen sich also verschiedene Aufrufvarianten des gleichen Algorithmus realisieren

```
log(d,2), log(d)
```

und / oder verschiedene Algorithmen für verschiedene Datentypen mit dem gleichen Namen verbinden.

```
sq(d), sq(i);
```

Funktioniert ein Algorithmus für verschiedene Datentypen identisch, so verwendet man in Java *Generics*.

Überladen wird auch verwendet bei der Auswahl des geeigneten Konstruktors beim Aufruf von new (kommt gleich!)

#### Klassen und Objekte: Motivation

Ziel: Netzberechnung für Wasserversorgung

Komponenten: Leitungen, Reservoire, Pumpen

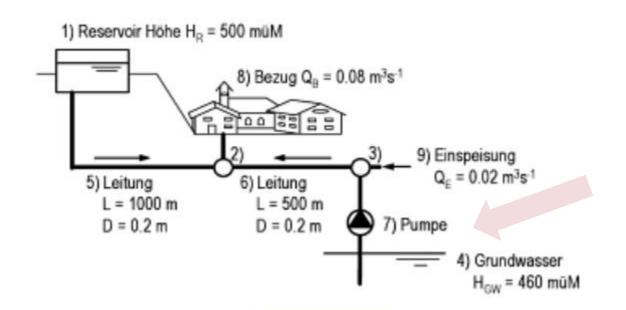




Abb. 11.12. Beispiel zur Netzberechnung: Ein einfaches Wasserversorgungsnetz

Quelle: Siedlungswasserwirtschaft, Willi Gujer, Springer

#### Klassen als Datencontainer

```
class Pumpe{
   public double h; // manometrische Förderhöhe in m
   public double q; // Fördermenge in m^3/s
Pumpe p1 = new Pumpe(); // eine Pumpe
                                                        Macht man
p1.h = 40; p1.q = 0.02;
                                                        das wirklich
                                                           so?
Pumpe p2 = new Pumpe(); // eine andere Pumpe
p2.h = 30; p2.q = 0.01;
```

#### Szenario

Jemand verwendet meine Pumpendaten

soweit ok

double v = p1.q \* 20; // Fördermenge in 20 Sekunden

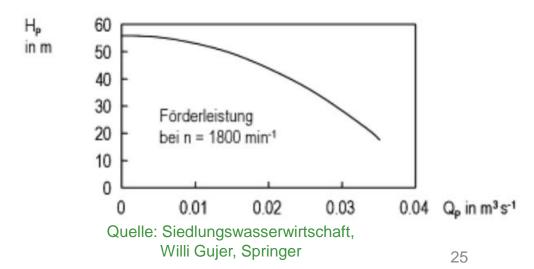
und verändert die Parameter der Pumpe:

$$p1.h = 50; p1.q = 0.02;$$

oh nein! h = 50 und q = 0.02 liegen doch nicht auf der Kennlinie dieser Pumpe!

→ Inkonsistenz!

Vermeidbar?



### Kapseln der Daten!

```
public class Pumpe{
    private double h; // .. in m
    private double q; // .. in m^3/s
}
```

```
Pumpe p1 = new Pumpe();
p1.h = 10; p1.q = 0.02;// Compilerfehler: kein Schreibzugriff
double v = p1.q * 10; // Compilerfehler: kein Lesezugriff
```

#### Konstruktoren

- sind spezielle Methoden, die den Namen der Klasse tragen und keinen Rückgabetyp haben
- können überladen werden, also in der Klasse mehrfach, aber mit verschiedener Signatur vorkommen (Überladen!)
- werden beim Aufruf von new wie eine Funktion aufgerufen. Der Compiler sucht die naheliegendste passende Funktion aus.
- wird kein passender Konstruktor gefunden, so gibt der Compiler eine Fehlermeldung aus.

#### Konstruktoren

```
public class Pumpe{
  private double h; // .. in m
  private double q; // .. in m^3/s
  // Konstruktor
  public Pumpe (double H, double Q){
     h = H; // Zugriff auf h in diesem Objekt
     q = Q; // Zugriff auf q in diesem Objekt
// Initialisiere Pumpe p1 mit h = 40 und q = 0.02
Pumpe p1 = new Pumpe(40, 0.02); // ok
double v = p1.q * 10;  // Compilerfehler
```

### Methoden für Zugriff auf private Daten

#### **Deklaration:**

```
class Pumpe{
  private double h; // .. in m
  private double q; // .. in m^3/s
  // Konstruktor
  public Pumpe (double H, double Q){
     h = H; q = 0;
   // Getter methoden
  public double GetH(){
     return h;
  public double GetQ(){
     return q;
```

#### Verwendung:

```
Pumpe p1 = new Pumpe (40, 0.02);
Pumpe p2 = new Pumpe(30, 0.01);
// Jeweilige Fördermengen in 1 Stunde
double v1 = p1.GetQ() * 3600;
double v2 = p2.GetQ() * 3600;
// Förderhöhe bei Reihenschaltung
double h = p1.GetH() + p2.GetH();
p1.h = 10; // Compilerfehler. Gut so!
```

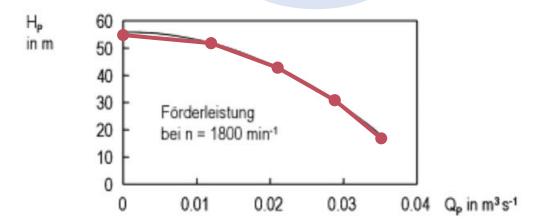
# Datenkapselung

- Eine komplexe Funktionalität wird auf einer möglichst hohen
   Abstraktionsebene semantisch definiert und durch ein vereinbartes minimales *Interface* zugänglich gemacht
- Wie der Zustand durch die Datenfelder einer Klasse repräsentiert werden, sollte für den Benutzer nicht sichtbar sein
- Dem Benutzer der Klasse wird eine repräsentations-unabhängige
   Funktionalität angeboten

# Pumpe mit Kennlinie

```
class Pumpe{
   private double h, q;
   private double[] ha, qa;
   // Konstruktor, wie vorher
   public Pumpe (double H, double Q){
      h = H;
      q = Q;
   // zusätzlicher Konstruktor
   public Pumpe (double[] H, double[] Q){
      ha = H; qa = Q; // Kopien der Referenzen
      h = ha[0]; q = qa[0] // Default Werte
```

# Für meine Netzberechnungen brauche ich H in Abhängigkeit von Q und umgekehrt!

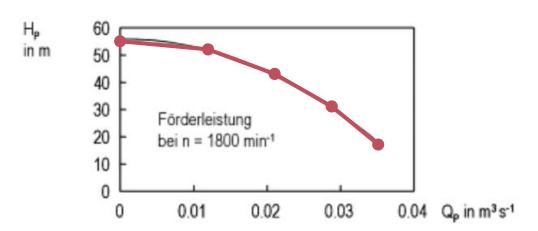


```
double[] q = {0,0.01,0.02,0.03,0.035};
double[] h = {55, 50, 40, 30, 20};
Pumpe p = new Pumpe(h, q)
```

#### Pumpe mit Kennlinie

```
class Pumpe{
                                             . .
   public Pumpe (double H, double Q) { ... }
   public Pumpe (double[] H, double[] Q) {...}
   public double GetH() { ... }
   public double GetQ() { ... }
   public static double Interpolate(double x, double[] in, double[] to) {...}
   public double SetH(double H) {
      h = H;
      q = Interpolate(H,ha,qa);
   public double SetQ(double Q) {
      q = Q;
      h = Interpolate(Q,qa,ha);
```

Die Pumpe ist immer noch kompatibel zur vorigen Version. Aber sie hat nun noch mehr Funktionalität.



### Pumpe mit Kennlinie: Verwendung

```
double[] kennlinieH = \{55,50,40,30,20\};
double[] kennlinieQ = \{0,0.01,0.02,0.03,0.035\};
Pumpe p = new Pumpe(kennlinieH, kennlinieQ);
p.SetH(45); // Setze Pumpenparameter (Förderhöhe)
double v = p.GetQ() * 3600; // Fördermengen in 1 Stunde
             0.015
```

#### Klassen

Klassen beherbergen das Konzept der Datenkapselung.

Klassen sind Bestandteil nahezu jeder **objektorientierten Programmiersprache.** 

Klassen können in Java in Paketen zusammengefasst werden.

# Modifizierer für Datenkapselung

```
public class Pumpe{
    private double h;
    private double q;
}
```

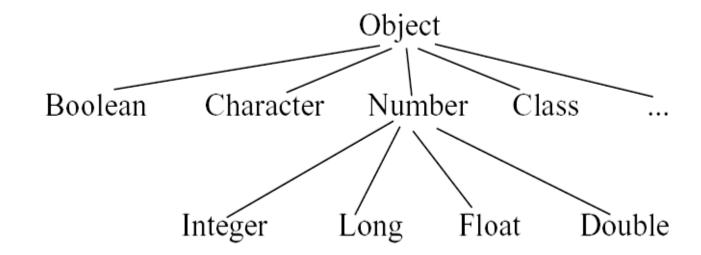
Sichtbarkeiten nach Modifizierer					
Modifizierer	Klasse	Paket	Sub-Klasse	Global	
public	✓	✓	✓	✓	
protected	✓	✓	✓		
(keiner)	✓	✓			
private	✓				

# Wrapperklassen

Primitive Typen (int, float,...) sind keine Objekte (im OO Sinne)

Java hat Wrapperklassen für jeden primitiven Typ

Manche Methoden der API\* fordern Objektreferenzen als Parameter



<sup>\*</sup> Application Programming Interface