Aufbau einer Kompilationseinheit, Qualified Identifer und Scopes, Prozedurales Programmieren in Java, Typkonversionen, Arrays, *Fallstudie Zufallssurfer*

2. JAVA

Aufbau einer Kompilationseinheit

- Kompilationseinheit (Datei) besteht aus *Klassen*.
  - Höchstens eine Klasse pro Einheit ist *public* und trägt den Namen der Kompilationseinheit
- Mit `package` kann eine Klasse einem Paket (einer Ansammlung von Klassen) zugeordnet werden.
- Mit `import` können Klassen im aktuellen Namensraum sichtbar gemacht werden. Das erlaubt kürzere Schreibweisen.
  - Import entspricht *nicht* dem `include` von C++ oder dem `uses` von Pascal.
  - In jedem java-Programm gibt es "versteckt" ein `import java.lang.*`
  - "Fremde" Klassen können über *voll qualifizierenden Namen (qualified identifier)* immer verwendet werden.

```
package ...;
import ...;

public class A{
        ...
}
class B{
        ...
}

class C{
        ...
}
```


Qualifizierende Namen

```
package MyLibrary;
import java.lang.*;

public class Hello {

  public static void main(String[] args)  {
    java.lang.System.out.println("Hello World.");
  }
}
```

Der Java Compiler fügt dieses hier (für uns unsichtbar) ein

Wegen des (versteckten) Imports oben kann das auch kürzer geschrieben werden:

```
System.out.println("Hello World.");
```

Aufruf: `java Mylibrary.Hello`

Qualifizierende Namen Analogie

Schweiz.Zürich.Clausiusstr.59.H.13

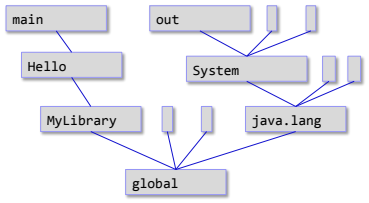
Diese Adresse ist weltweit eindeutig und ist somit *voll qualifizierend*.

Bin ich schon in der Schweiz und Zürich und in der Clausiusstrasse, so genügt die Angabe von

59.H.13

Scopes (Sichtbarkeitsbereiche)

... haben übrigens Baumstruktur!



Aufbau einer Klasse

- Klasse besteht aus
  - Instanzen- und Klassenvariablen
  - Benannte Konstanten
  - Methoden
    - Methoden übernehmen die Rolle von Funktionen / Prozeduren in anderen Sprachen
    - Konstruktoren sind spezielle Methoden, sie werden beim Erzeugen einer Klasse automatisch aufgerufen
  - Code im Klassenkörper
    - wird beim Instanzieren der Klasse ausgeführt
    - falls `static` so wird er bei erstmaliger Verwendung der Klasse ausgeführt

```
public class A{

  public static int F(int x)
  {
      ...
      return j
   }

  A()
  {
     ...
  }

  { // Code im Klassenkörper
    ...
  }
}
```


Aufbau einer Klasse

- Bei eigenständigen Programmen muss es eine main-Methode geben:

```
public static void main(String args[]) {
    ...
}
```

- Jede Klasse kann eine solche main-Methode enthalten. Sie wird beim *Aufruf der Klasse* ausgeführt.
- Klassen können getrennt übersetzt werden
- Variablen im Klassenkörper (ausserhalb von Methoden) sind global zu allen Methoden der Klasse

```
public class A{

  ... main (....)
  {
      ...
      return j
   }

  ...
  int k;

}
```

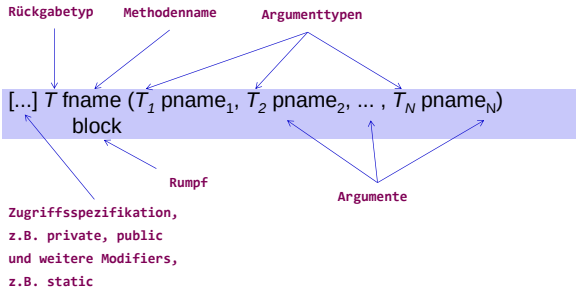

Methoden

- Methoden haben immer
  - Name
  - Rückgabetyp
  - (potentiell leere) Parameterliste
- und optional
  - Zugriffsspezifizierer
  - Weitere Modifizierer

Beispiel:

```
public static void main(String args[])
{ ... }
```

Methodendeklaration



Prozedurales Programmieren in Java

Statische Methoden spielen bei Java die Rolle von Prozeduren in Pascal

```
public class Newton {

    // return the square root of c, computed using Newton's method
    public static double sqrt(double c) {
        if (c < 0) return Double.NaN;
        double EPS = 1E-15;
        double t = c;
        while (Math.abs(t - c/t) > EPS*t)
            t = (c/t + t) / 2.0;
        return t;
    }

   ...
}
```


Prozedurales Programmieren in Java

Aufruf der statischen Methode aus der gleichen Klasse heraus

```
public class Newton {
   public static double sqrt(double c) { ... }

   public static void main(String[] args) {
       for (int i = 0; i < args.length; i++) {
         double a = Double.parseDouble(args[i]);
         double x = sqrt(a);
         System.out.println(x);
       }
   }
}
```


Prozedurales Programmieren in Java

Aufruf der statischen Methode aus einer anderen Klasse heraus (im gleichen Paket)

```
public class NewtonTest {

   public static void main(String[] args) {
       for (int i = 0; i < args.length; i++) {
         double a = Double.parseDouble(args[i]);
         double x = Newton.sqrt(a);
         System.out.println(x);
       }
   }
}
```

Klassenname

Methodenname

Statische Methoden können also über den Namen der beinhaltenden *Klasse* angesprochen werden.

Globale Variablen?

Statische Variablen (also Variablen mit Modifizierer `static`) spielen bei Java die Rolle von globalen Variablen in Pascal

- Statische Variablen existieren jeweils nur einmal pro Klasse, sie werden auch als Klassenvariablen bezeichnet.
  - Nicht statische Variablen existieren jeweils nur einmal pro Instanz einer Klasse (Objekt) – später mehr dazu
- Statische Variablen werden *äusserst selten* verwendet

Typkonversionen

- Java ist streng typisiert.
  - Compiler detektiert statische Inkompatibilitäten
  - Laufzeit detektiert dynamische Inkompatibilitäten

```
int i = 100;
float f = 2.712f;
i = f;          // type mismatch. Cannot convert from double to int.
i = (int) f;    // das geht: explizit konvertiert
f = i;          // das geht auch: implizit konvertiert zu einem grösseren Wertebereich

Tier tier;
...
Hund hund = (Hund) tier; // explizit konvertiert, dynamisch geprüft
                         (erklären wir später)
```

aber stimmt "grösser" eigentlich ?

Arrays

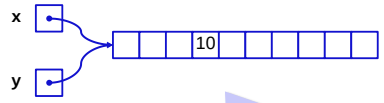
- Arrays sind konzeptuell grundsätzlich *dynamisch* erzeugt

```
int [] b; // array von ints
b = new int[10]; // Grösse 10: Indices von 0..9
...
b = new int[20]; // kann neu zugewiesen werden
int [] x = new int[10]; // das geht auch
```

- Die Grösse eines Arrays kann also zur Laufzeit festgelegt werden.
  - Ein Array kann in dieser Form jedoch nicht einfach "wachsen".

Arrays sind Zeiger

```
int [] x = new int[10];
int [] y;
y = x; // was passiert hier?
y[3] = 10; // und hier?
```



Wann gilt `y == x` ?

Arrays sind Zeiger auf Speicherobjekte: Vorsicht bzgl. Kopiersemantik ("Aliaseffekt") und beim Vergleich zweier Array-Variablen.

Arrays sind nicht primitiv

- Arrays tragen *Metadaten* mit sich herum:

```
int[] sq = new int[7];
for (int i=0; i < sq.length; i++) sq[i]=i*i;
sq[8] = 100;
```

java.lang.ArrayIndexOutOfBoundsException

- Es funktioniert sogar `print(sq);` mit

```
static void print (int a[])
{
    for (int i=0; i < a.length; ++i)
    { System.out.println("a[" + i + "]=" +a[i]);}
}
```

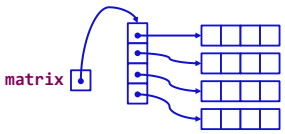

Mehrdimensionale Arrays

```
float [][] matrix = new float [4][4];
for (int r=0; r < matrix.length; ++r)
  for (int c=0; c < matrix[r].length; ++c)
    matrix[r][c] = (r==c) ? 1 : 0;
```

ternärer Operator
`b ? a : b` ergibt
a wenn b wahr ist, sonst b

matrix[x] ist ein float[] somit geht auch matrix[x] = new float[y];

Eine mehrdimensionales Array ist also ein Array von Zeigern!



Argumente

Beim Aufruf eines Programmes mit Parametern via

`java Programmname [Parameter]`

werden die (whitespace getrennten) Parameter der `main` Methode als ein Array von Strings in der Variable `args` übergeben.

```
public class Parameter{
    public static void main(String[] args) {
        for (int i = 0; i<args.length; ++i)
            System.out.println("arg[" + i + "]=" + args[i]);
    }
}
```

Eingabe `java Parameter a13 b c=1`

Ausgabe:
```
arg[0]=a13
arg[1]=b
arg[2]=c=1
```


Ein Java Programm mit Argumenten

```
public class Test{
    public static void main(String[] args) {

        for (int argIndex = 0; argIndex < args.length; ++argIndex)
        {
            String numString = args[argIndex];
            int num=0;
            for (int numIndex = 0; numIndex < numString.length(); ++numIndex)
            {
                num *= 2;
                if (numString.charAt(numIndex) == '1') ++num;
            }
            System.out.print(numString + "=" + num + "\n");
        }
    }
}
```

Übung: Welche Ausgabe erzeugt der Aufruf **`"java Test 1001"`** ?

System.in, System.out, System.error

Beim Aufruf eines Programmes (genauer: beim Start der Virtuellen Maschine) werden ein Ausgabe-, ein Fehlerausgabe- und ein Eingabestrom instanziert.

- `System.out, System.err`
  Ausgabe und Fehlerausgabe unterscheiden sich nur marginal (z.B. wird Ausgabe über Fehlerstrom in Eclipse rot ausgegeben).
  - Typische Verwendung `System.out.print()`
- `System.in`
  Eingabestrom liest typischerweise Zeichen von der Tastatur und liefert Buchstaben.
  - Typische Verwendung über `java.util.Scanner`

Umleiten der Ein- und Ausgabe

- Ausgabe in eine Datei
  `java Klassennamne [Parameter] > Dateiname`
- Eingabe von einer Datei
  `java Klassenname [Parameter] < Dateiname`
- Umleitung: Ausgabe von Klasse A als Eingabe von Klasse B
  `java A [Parameter] | java B [Parameter]`

Sehr nützliches Tool zur generischen Verwendung von Funktionalität über die Kommandozeile

Ein-/Ausgabe Einfaches Beispiel

```
public class Ausgabe {
public static void main(String[] args) {
  int len = 0;
  if (args.length > 0)
    len = Integer.parseInt(args[0]);
  System.out.print(len);
  for (int i = 0; i < len; ++i)
    System.out.print(" "+ i);
  } }
```

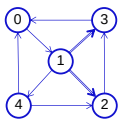
```
import java.util.Scanner;
public class Eingabe {
public static void main(String[] args) {
  Scanner scanner = new Scanner(System.in);
  if (scanner.hasNextInt()) {
    int len = scanner.nextInt();
    int sum = 0;
    while (len-- > 0 && scanner.hasNextInt())
      sum += scanner.nextInt();
    System.out.println("sum = "+sum);
  }
  scanner.close();
} }
```

```
Aufruf    java Ausgabe  5
Ausgabe   5 0 1 2 3 4 5

Aufruf    java Eingabe 2
Eingabe   2 1 2
Ausgabe   sum = 3

Aufruf    java Ausgabe 100| java Eingabe
Ausgabe   sum = 4950
```


Fallstudie: Der 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 jeder Entscheidung wird jeder 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 gleichverteilt von beliebiger Seite gestartet.
- Wie gross ist die Wahrscheinlichkeit, dass er nach sehr langer Zeit auf Seite x landet?




```
Eingabe:
5
0 1
1 2 1 2 1 3 1 3 1 4
2 3
3 0
4 0 4 2
end
```

Sedgewick, 1.6.

Übergangsmatrix

Für jede Entscheidung gilt

$$p_{ij} := \mathbb{P}(s_t = j \mid s_{t-1} = i, \dots) = \mathbb{P}(s_t = j \mid s_{t-1} = i) = \begin{cases} d \cdot \frac{1}{n} + (1-d) \cdot \frac{l_{ij}}{c_i} & falls\ c_i \neq 0 \\ \frac{1}{n} & sonst \end{cases}$$

Die Entscheidung hängt nur vom derzeitigen Aufenthaltsort ab (Markov-Eigenschaft)

wobei

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

Übergangsmatrix

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

Jede Zeile ist ein Wahrscheinlichkeits-vektor

W't von der Seite 1 startend auf die Seite 3 zu wechseln

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

Generierung der Übergangsmatrix

```
public class Transition {
 public static void main(String[] args) {
   java.util.Scanner scanner = new java.util.Scanner(System.in);
   int N = scanner.nextInt();          // Anzahl Seiten
   int[][] counts = new int[N][N];    // counts[i][j] = # links Seite i → j
   int[] outDegree = new int[N];      // outDegree[i] = #links Seite i

    while (scanner.hasNextInt()) { // Akkumulieren der link Counts.
       int i = scanner.nextInt();
       int j = scanner.nextInt();
       outDegree[i]++;
       counts[i][j]++;
   }
   System.out.println(N + " " + N);
   ....
```

Eingabe von System.in

Generierung der Übergangsmatrix

```
    ....
    for (int i = 0; i < N; i++) {    // W'tsverteilung Zeile i
        for (int j = 0; j < N; j++) { // W'tswert Spalte j
          double p;
          if (outDegree[i] > 0) // Zeile i hat ausgehende Kanten
              p = .90*counts[i][j]/outDegree[i] + .10/N;
          else  // Spezialfall Zeile i hat keine ausgehende Kanten
              p = 1.0/N;
          System.out.printf("%7.5f ", p);
        }
        System.out.println();
    }
    scanner.close();
  }
}
```

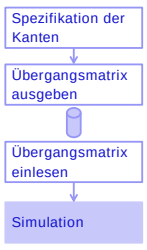
Umlenkung der Eingabe

```
z.B. Aufruf: java Transition < "tiny.txt"
Quelle http://introcs.cs.princeton.edu/java
Section 2.6
```


Separation of Concerns

Mit der Generierung der Übergangsmatrix und folgender generischer Einlesefunktion für Matrizen ...

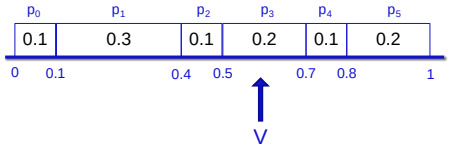
```
public static double[][] InputMatrix()
{
   java.util.Scanner scanner = new java.util.Scanner(System.in);
   int M = scanner.nextInt(); // Zeilen
   int N = scanner.nextInt(); // Spalten
   // Daten
   double[][] p = new double[M][N];
   for (int i = 0; i < N; i++)
         for (int j = 0; j < N; j++)
         p[i][j] = scanner.nextDouble();
   return p;
}
```



... können wir uns nun mit der Essenz des Problems beschäftigen

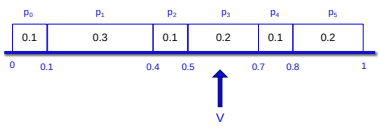
Simulation eines Schrittes

- Wir wollen das Verhalten des Surfers simulieren
- Wie erzeugen wir das zufällige Verhalten gemäss $\boldsymbol{P}$?
- Methode zur Generierung von Zufallszahlen uniform verteilt auf Intervall [0,1) ist vorhanden: `Math.random()`
- Also: bereits vorhanden: Zufallsvariable V mit
  $$\mathbb{P}(V \in [l,r)) = r - l, \qquad 0 \le l \le r \le 1$$
- Sampling Idee:



Simulation eines Schrittes

```
public static int Simulate(double[] prob)
{
   int res=0;
   double r = Math.random();
   double sum = 0.0;
   for (int j = 0; j < prob.length; j++) {
      sum += prob[j];
      if (r < sum) {res = j; break;}
   }
   return res;
}
```



Simulation der Kette

```
public class RandomWalk{
   public static double[][] InputMatrix() {...}
   public static int Simulate(double[] prob) {...}

   public static void main(String[] args) {
      int T=100;
      if (args.length > 0) T = Integer.parseInt(args[0]); // number moves
      double [][] p = InputMatrix();
      assert p.length == p[0].length;
      int N = p.length;
      int[] freq = new int[N];  // freq[i] = # times surfer hits page i
      int page = 0;             // start auf Seite 0

      for (int t = 1; t <= T; t++)
      {
         page = Simulate(p[page]);
         freq[page]++;
      }

      for (int i = 0; i < N; i++)
         System.out.printf("%8.5f", (double) freq[i] / T);
   }
}
```

Eigentliche Kernaufgabe sehr einfach geworden

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 (in mittlerweile leicht modifizierter Form).

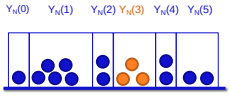
MCMC

- Die Simulation, die wir oben ausgeführt haben, ist die Simulation einer Markov-Kette
- Zufallsbasierte Simulationen nennt man auch Monte-Carlo Verfahren.
  - Hier haben wir es also sogar mit einem Markov Chain Monte Carlo (MCMC) Verfahren zu tun!

Anzahl Iterationen?*

Frage: Wie viele Iterationen genügen bei gegebener gewünschter Genauigkeit, z.B. auf die 3. Stelle nach dem Komma?

Vereinfachtes (Gedanken-)Experiment: Ziehe identisch verteilte, unabhängige Zufallszahlen $X_i, i = 1 \dots N$ mit (unbekannter) Erfolgswahrscheinlichkeit $p_x = \mathbb{P}(X_i = x)$, $x \in \{0 \dots n-1\}$



- Betrachte $Y_N(x) = \sum_{i=1}^{N} \delta(X_i = x)$
- $Y_N(x)$ haben Binomialverteilung.
- Erwartungswert $\mathbb{E}\left(\frac{Y_N(x)}{N}\right) = p_x$
- Varianz $\mathbb{V}\left(\frac{Y_N(x)}{N}\right) = \frac{p_x \cdot (1-p_x)}{N}$

Anzahl Iterationen?*

- Approximation der Binomialverteilung durch Normalverteilung mit Varianz $\sigma_x^2 = \frac{p_x \cdot (1-p_x)}{N}$

  Das ist nicht die beste Approximation, aber für unsere Zwecke hier reicht's

- 95% aller Messwerte haben eine Abweichung von höchstens 2 Standardabweichungen σ_x
- Das heisst die Genauigkeit ε der Approximation skaliert mit $\frac{1}{\sqrt{N}}$
- 95% Konfidenzintervall, worst case (p=0.5) ergibt $N = \frac{1}{\varepsilon^2}$
- $\varepsilon = 0.001 \Rightarrow N = 10^6$ (!)

Geht das nicht schneller?*

- Wir kennen die Übergangswahrscheinlichkeiten
- Beginnen wir (z.B.) mit dem Wahrscheinlichkeitsvektor

$$\nu = (1,0,\dots,0),$$

starten also mit W't 1 im ersten Knoten, dann können wir die Wahrscheinlichkeiten für jeden Ort nach einem Schritt berechnen:

$$\mu_1 = \nu \cdot \boldsymbol{P}$$

nach zwei Schritten

$$\begin{bmatrix} \nu_0 & \nu_1 & \nu_2 \end{bmatrix} \begin{bmatrix} p_{00} & p_{01} & p_{02} \\ p_{10} & p_{11} & p_{12} \\ p_{20} & p_{21} & p_{22} \end{bmatrix}$$

$$\mu_2 = \mu_1 \cdot \boldsymbol{P} = \nu \cdot \boldsymbol{P} \cdot \boldsymbol{P}$$

... nach t Schritten:

Vektor-Matrix Multiplikation

$$\mu_t = \mu_{t-1} \cdot \boldsymbol{P} = \nu \cdot \underbrace{\boldsymbol{P} \cdot \dots \cdot \boldsymbol{P}}_{t \text{ mal}}$$

Markov Ketten!*

Wir haben es hier mit einer homogenen Markov-Kette mit primitivem Kern zu tun. Es gilt folgende Aussage:

Es gibt einen W'tsvektor μ mit

das impliziert Konvergenz für jeden Eintrag

$\nu P^n \to \mu$ (in Totalvariation) für $n \to \infty$

Darüber hinaus gilt

$\mu P = \mu$

Das können wir verwenden, um zu prüfen wie nahe wir am Grenzwert sind

Man nennt μ die *stationäre Verteilung* der Markov-Kette.

G.Winkler, Image Analysis, Random Fields and Markov Chain Monte Carlo Methods: A Mathematical Introduction, Springer 2003

Was nützt uns das?*

- Wir haben den Nachweis: der Pagerank Algorithmus konvergiert tatsächlich.
- Bei der direkten Berechnung von $\mu \cdot \boldsymbol{P} \cdot \boldsymbol{P} \cdot \dots \cdot \boldsymbol{P}$ wird die Approximation der Wahrscheinlichkeiten durch Simulation vermieden. Der Algorithmus ist viel schneller.
- Wie viele Schritte benötigt diese Variante? Probieren Sie es aus! (Übung)

MCMC Burn-In*

- Bei Simulation einer Markov-Kette gilt es die sogenannte "Burn-In Phase" der Markov-Kette zu überwinden.
- Nach der Burn-In Phase ist die stationäre Verteilung genügend nahe approximiert. Die Bestimmung der Länge der Burn-In Phase ist ein wichtiges Thema im Zusammenhang mit MCMC Verfahren.
- Für unsere kleine Übergangsmatrix ist die Burn-In Phase sehr kurz.