Programmierkurs

Vorlesung 08

M.Sc. Laslo Hunhold

Department Mathematik/Informatik
Abteilung Informatik
Universität zu Köln

11. Januar 2021

Letzte Vorlesung

- Schlüsselwort `this`
- Zugriffsmodifikatoren (`final`, `private`, `public`, `protected`)
- Vererbung (Schlüsselwörter `extends`, `super`)
- Kompatibilität/Polymorphismus (statischer/dynamischer Typ, Operator `instanceof`)
- Abstrakte Klassen und Methoden (Schlüsselwort `abstract`)

Konstanten mit `final static`

```
1  final class Kreis {
2          final static double PI = 3.141592653589793;
3          final double radius;
4
5          Kreis(double radius) {
6                  this.radius = radius;
7          }
8  }
```

- Einziger Weg für „echte“ Konstanten in Java
- Konvention: Variablenname komplett in Großbuchstaben und mit Unterstrichen getrennt

Vertiefung zu Vererbung, Kompatibilität, Polymorphismus und abstrakten Klassen

Beispiel (klassische versus polymorphe Zeiger)

```
1  class Tier {
2          int gewicht; /* kg */
3          void fresse() { ... }
4          void schlafe() { ... }
5  }
```

```
1  class Hund extends Tier {
2          boolean reinrassig;
3          void belle() { ... }
4  }
```

```
1  Tier t = new Tier();
2  Hund h = new Hund();
3  Tier p = new Hund();
```

- `t` hat die (Instanz)variable `gewicht` und „erlaubt“ die Methoden `fresse()` und `schlafe()`
- `h` hat `gewicht`, `fresse()` und `schlafe()` geerbt, aber hat auch die (Instanz)variable `reinrassig` und „erlaubt“ die Methode `belle()`
- Statischer und dynamischer Typ stimmen jeweils bei `t` und `h` überein
- `p` hat den statischen Typ `Tier` und den dynamischen Typ `Hund` (polymorpher Zeiger); `p.fresse()` ist erlaubt, aber `p.reinrassig` und `p.belle()` nicht

Beispiel (Auflösung mit `instanceof`)

```
1  Tier p = new Hund();
2  p.gewicht = 15;
3  p.fresse();
4
5  if (p instanceof Hund) {
6          Hund h = (Hund)p;
7
8          h.reinrassig = true;
9          h.schlafe();
10 }
```

- Im Speicher liegt ein `Hund`
- Müssen polymorphen Zeiger `p` auf den „vollen“ statischen Typen `Hund` casten
- Operator `instanceof` hilft bei Unterscheidung des dynamischen Typs

Beispiel (Verkapselung)

```
1  class Vater {
2          int vater;
3  }
```

```
1  class Sohn extends Vater {
2          int sohn;
3  }
```

```
1  class Enkel extends Sohn {
2          int enkel;
3
4          Enkel() {
5                  super();
6          }
7  }
```

```
Enkel e = new Enkel();
```

- e hat die (Instanz)variablen `vater`, `sohn` und `enkel`
- Aufruf des Superklassenkonstruktors mit `super()` möglich, aber nicht darüber hinaus (weil man sonst die Hierarchie „umgeht“)

Beispiel (Verschachtelung abstrakter Klassen)

```
1  abstract class Vater {
2          int vater;
3          void vater_methode() { ... }
4          abstract void hallo();
5  }
```

```
1  abstract class Sohn extends Vater {
2          int sohn;
3          abstract void hallo();
4  }
```

```
1  class Enkel extends Sohn {
2          int enkel;
3          void hallo() {
4                  System.out.println("Hallo Welt!");
5          }
6  }
```

```
1  Enkel e = new Enkel();
2
3  e.vater = 0;
4  e.sohn = 0;
5  e.enkel = 0;
6  e.vater_methode();
7  e.hallo();
```

Interfaces

Motivation

- Man kann nur von einer Klasse direkt erben
- Oft ist aber eine strikte Hierarchie sinnlos (z.B. wenn es um Fähigkeiten geht, von denen man mehrere haben kann)
- Interfaces erlauben einer Klasse, mehr als nur eine Eigenschaft abstrakt zu „erben“

Deklaration und Verwendung

```
1  interface Beispiel1 {
2          int KONSTANTE1 = 1;
3          void methode1();
4  }
5
6  interface Beispiel2 {
7          int KONSTANTE2 = 2;
8          void methode2();
9  }
10
11 class Test implements Beispiel1, Beispiel2 {
12         public void methode1() { ... }
13         public void methode2() { ... }
14 }
```

- Schlüsselwörter `interface` und `implements`
- Ein Interface ist sehr ähnlich zu einer abstrakten Klasse (nicht instanziierbar; jede abstrakte Methode muß der Erbe bereitstellen)
- Jedes Interface muß in eine eigene `java`-Datei
- Methoden in einem Interface sind standardmäßig `abstract` und `public`, Variablen sind `static` und `final` (also Konstanten)
- Man kann mehrere Interfaces in einer Klasse implementieren; Angabe als kommagetrennte Liste

Beispiel

```
1  public class Tier {
2          int gewicht; /* kg */
3          void fresse() { ... }
4          void schlafe() { ... }
5  }
```

```
1  public interface Giftig {
2          String gebe_gift_name();
3  }
4
5  public interface Raubtier {
6          void toete();
7  }
8
9  public interface Beutetier {
10         void fliehe();
11 }
```

```
1  public class Schlange extends Tier implements Giftig, Raubtier {
2          String gebe_gift_name() { ... }
3          void toete() { ... }
4  }
```

```
1  public class Kugelfisch extends Tier implements Giftig, Beutetier {
2          String gebe_gift_name() { ... }
3          void fliehe() { ... }
4  }
```

Subinterfaces

```
public interface GiftigesRaubtier extends Giftig, Raubtier {
        ...
}
```

- Interfaces sind auch vererbbar
- Ein Interface kann, anders als Klassen, von mehreren gleichzeitig erben (hier `Giftig` und `Raubtier`)
- Implementiert eine Klasse `GiftigesRaubtier`, so muß es alle Methoden der Interfaces `Giftig` und `Raubtier` bereitstellen
- Geerbte abstrakte Methoden *gleicher* Signatur und Rückgabewert erzeugen keinen Konflikt, sondern werden einfach „gleichzeitig“ implementiert

Rekursion

Motivation

Aufgabe: Definiere eine Methode `fakultaet(int n)`, die $n!$ berechnet

```
1  static int fakultaet(int n) {
2          int fak;
3
4          for (fak = 1; n > 0; n--) {
5                  fak *= n;
6          }
7
8          return fak;
9  }
```

```
1  static int fakultaet(int n) {
2          return (n == 0 || n == 1) ? 1 : (n * fakultaet(n - 1));
3  }
```

Bemerkungen

- Methoden können sich selbst („rekursiv“) aufrufen
- Methoden ohne Selbstaufruf nennt man „iterativ“
- Rekursion braucht definitive Abbruchbedingung, sonst folgt ein (un)endlicher Abstieg in sich selbst

```
1 static int hoelle() {
2         return hoelle();
3 }
```

Führt zu einem Stapelüberlauf („Stack Overflow“)

- Nachteil: Eine rekursive Methode hat einen „Overhead“ und ist oft ineffizienter als ihr iteratives Pendant
- Vorteil: Rekursion erlaubt eine klare Aufspaltung eines Problems in Teil-/Unterprobleme

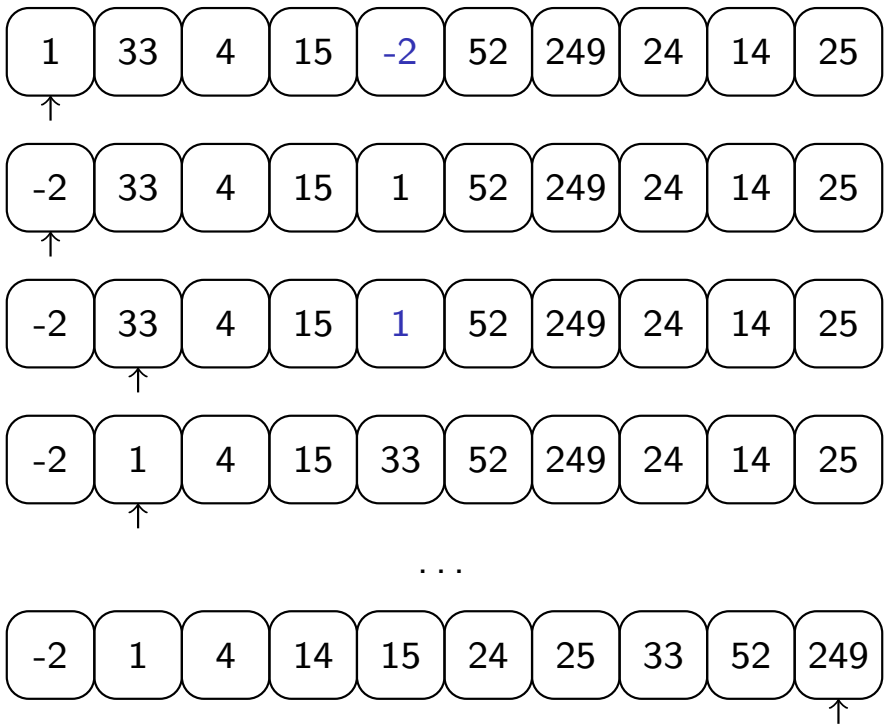
Sortieralgorithmen

Sortierproblem

- Gegeben: Array aus n Zahlen, z.B.
  ```
  { 1, 33, 4, 15, -2, 52, 249, 24, 14, 25 }
  ```
- Gesucht: (O.B.d.A.) aufsteigende Sortierung des Arrays, also z.B.
  ```
  { -2, 1, 4, 14, 15, 24, 25, 33, 52, 249 }
  ```
  Eine absteigende Sortierung erhält man durch die Umkehrung des aufsteigend sortierten Arrays (deshalb o.B.d.A.)
- Problem: Wie löst man die Aufgabe möglichst effizient?

„Selection Sort“-Algorithmus (Vorgehen)

0. Lege einen Markierer an den Anfang des Arrays
1. Bestimme das Minimum im Array beginnend am Markierer
2. Tausche das Minimum mit dem Element am Markierer
3. Bewege den Markierer um eins nach rechts
4. Wenn der Markierer nicht am Ende des Arrays ist, gehe zu Schritt 1



„Selection Sort“-Algorithmus (Implementierung)

```
 1  public class Main {
 2          public static void print(int[] arr) {
 3                  for (int i = 0; i < arr.length; i++) {
 4                          System.out.print(arr[i] + " ");
 5                  }
 6                  System.out.print('\n');
 7          }
 8          public static void selection_sort(int[] arr) {
 9                  for (int i = 0; i + 1 < arr.length; i++) {
10                          /* finde Index ind des kleinsten Elementes (ab Index i) */
11                          int ind = i;
12                          for (int j = ind + 1; j < arr.length; j++) {
13                                  if (arr[j] < arr[ind]) {
14                                          ind = j;
15                                  }
16                          }
17                          /* tausche das kleinste Element mit dem an Stelle i */
18                          int tmp = arr[i];
19                          arr[i] = arr[ind];
20                          arr[ind] = tmp;
21                  }
22          }
23          public static void main(String[] args) {
24                  int[] arr = new int[]{ 1, 33, 4, 15, -2, 52, 249, 24, 14, 25 };
25                  print(arr);
26                  selection_sort(arr);
27                  print(arr);
28          }
29  }
```

„Mergesort“-Algorithmus (Vorgehen)

0. Stoppe, wenn die Länge des Eingangsarrays `arr` 1 oder 0 beträgt
1. Teile den Eingangsarray `arr` in der Mitte in zwei Arrays `left_arr` und `right_arr`
2. Wende Mergesort jeweils auf `left_arr` und `right_arr` an
3. Füge die (sortierten) Arrays `left_arr` und `right_arr` in `arr` sortiert zusammen

„Mergesort“-Algorithmus (Implementierung)

```
 1  public class Main {
 2          public static void print(int[] arr) { ... }
 3          public static void mergesort(int[] arr) {
 4                  if (arr.length == 0 || arr.length == 1) { return; }
 5                  int half = arr.length / 2;
 6                  int[] left_arr = new int[half],
 7                        right_arr = new int[arr.length - half];
 8
 9                  for (int i = 0; i < arr.length; i++) {
10                          if (i < half) { left_arr[i] = arr[i];
11                          } else { right_arr[i - half] = arr[i]; }
12                  }
13                  mergesort(left_arr); mergesort(right_arr);
14
15                  /* fuege sortierte Arrays left_arr und right_arr zusammen */
16                  for (int i = 0, j = 0, k = 0; i < arr.length; i++) {
17                          arr[i] = (j == left_arr.length) ? right_arr[k++] :
18                                   (k == right_arr.length) ? left_arr[j++] :
19                                   (left_arr[j] < right_arr[k]) ? left_arr[j++] :
20                                   right_arr[k++];
21                  }
22          }
23          public static void main(String[] args) {
24                  int[] arr = new int[]{ 1, 33, 4, 15, -2, 52, 249, 24, 14, 25 };
25                  print(arr);
26                  mergesort(arr);
27                  print(arr);
28          }
29  }
```

Übersicht

- Selection Sort ist ein iterativer Algorithmus; der Rechenaufwand für einen Array der Länge n liegt in der Größenordnung n^2
- Mergesort ist ein rekursiver Algorithmus; Rechenaufwand für einen Array der Länge n liegt in der Größenordnung $n \log(n)$
- Mergesort ist ein Beispiel für ein sogenanntes „Teile-und-herrsche-Verfahren“ („divide and conquer“)

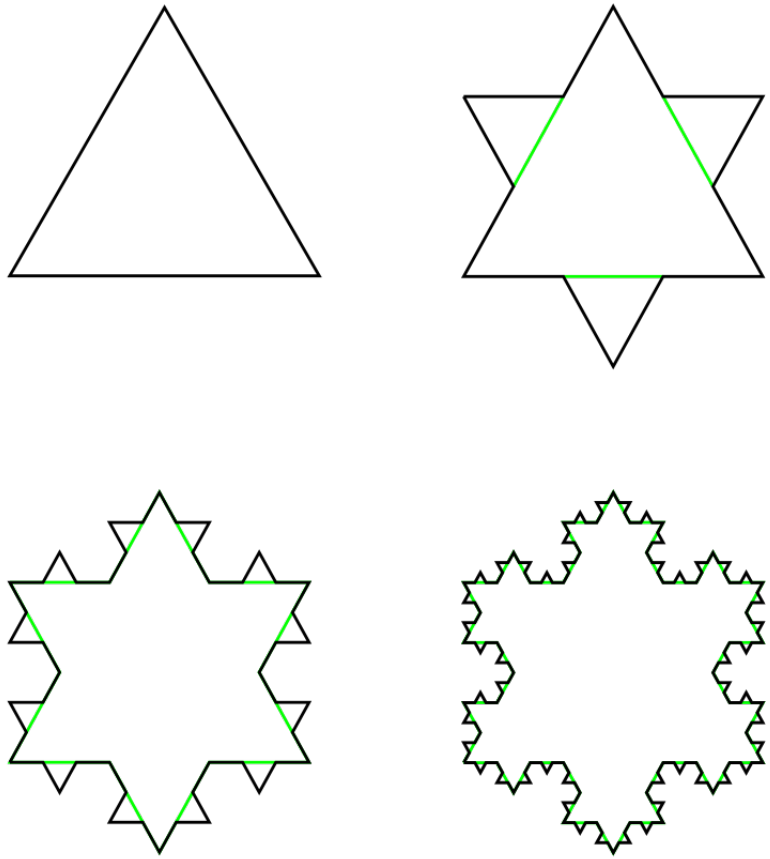


Abbildung: Die ersten vier Iterationen der KOCHschen Schneeflocke (Quelle: WxS, CC BY-SA 3.0)