Die Schnittstelle Comparable

Wir wollen Such- und Sortieroperationen für beliebige Objekte definieren.

Dazu verwenden wir die vordefinierte Schnittstelle Comparable:

```
public interface Comparable {  
    int compareTo(Object other);  
}
```

Wenn $o : Comparable$ und $other : Object$ und o mit $other$ vergleichbar ist, dann sollte gelten

$o.compareTo(other) < 0$, falls o kleiner $other$

$o.compareTo(other) = 0$, falls o gleich $other$

$o.compareTo(other) > 0$, falls o größer als $other$

Beispiele

Die Klasse `String` implementiert automatisch die Schnittstelle `Comparable`.

Die Ordnung ist dabei die *lexikographische* Ordnung.

Der Ausdruck

`"AAAaaa".compareTo("Mein Schluesseldienst")` hat einen Wert < 0 .

Der Ausdruck `"AAAaaa".compareTo(new Point(2,3))` ist typkorrekt (warum?) führt aber zu einem Laufzeitfehler (Programmabbruch).

Beispiele

Bankkonten nach ihrer Kontonummer sortieren:

```
public class BankkontoAngeordnet extends Bankkonto implements Comp
    public int compareTo(Object other) {
        return getKontonummer() - ((Bankkonto)other).getKontonumme
    }
}
```

Problem: die Ordnung ist jetzt festgelegt.

Binäre Suche

Wdh.: aus Info I (NB: vect = array).

```
sucheVonBis = function(l:string vect, w:string, i:int, j:int)bool
  if i > j then false else
  if i = j then get(l, i) = w else
    let m = [(i + j)/2] in
    let wm = get(l, m) in
    if wm = w then true else
      if w < get(l, m) then
        sucheVonBis(l, w, i, m - 1)
      else sucheVonBis(l, w, m + 1, j)
suche = function(l:string vect, w:string)sucheVonBis(l, w, 1, dim(l))
```

Dasselbe in Java mit Comparable

```
public static boolean sucheVonBis(Comparable[] l, Object w, int i, int j)
{
    if (i > j) return false;
    if (i == j) return l[i].compareTo(w) == 0;
    int m = (i+j) / 2;
    Comparable wm = l[m];
    int comp = wm.compareTo(w);
    if (comp == 0) return true;
    if (comp < 0) // wm < w
        return sucheVonBis(l, w, m+1, j);
    else
        return sucheVonBis(l, w, i, m-1);
}
```

Äquivalente iterative Version

```
public static boolean sucheIter(Comparable[] l, Object w) {
    int i = 0;
    int j = l.length-1;
    boolean gefunden = false;
    while(!gefunden && i < j) {
        int m = (i + j)/2;
        Comparable wm = l[m];
        int comp = wm.compareTo(w);
        if (comp == 0) gefunden = true;
        else if (comp < 0)
            j = m - 1;
        else
            i = m + 1;
    }
    return gefunden;
}
```

Geringfügig effizienter, kein wesentlicher Unterschied.

Die Schnittstelle Comparator

Die Verwendung von Comparable ist ungünstig, wenn man ein- und dieselben Daten nach mehreren Ordnungen sortieren muss (Kontonummer, Betrag, PLZ, ...).

Dasselbe gilt, wenn Sie beim Schreiben einer Klasse noch nicht wissen, welche Ordnung “natürlich” ist.

Die Schnittstelle Comparator enthält eine Methode:

```
int compare(Object o1, Object o2)
```

Die intendierte Semantik ist dieselbe wie bei Comparable. Ergebnis kleiner Null bedeutet, o1 vor o2 etc.

Man könnte jetzt die binäre Suche auch mit Komparatoren definieren:

```
public static boolean sucheVonBis(Object[] l,  
    Object w, int i, int j, Comparator c) {...}
```

Beispiel

Obwohl `Integer` die Schnittstelle `Comparable` implementiert, möchten wir vielleicht einen `Comparator` dafür haben:

```
import java.util.*;
public class IntegerComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        return ((Integer)o1).intValue() - ((Integer)o2).intValue()
    }
}
```

Jetzt ist `new IntegerComparator()` ein `Comparator` für `Integers`.

Obwohl die beiden Argumente von `compare` den Typ `Object` haben, lässt sich die Methode doch nur für `Integer` verwenden!

Typvariablen und Generics

- Der Programmierer trägt die Verantwortung, dass ein Komparator nur auf solche Objekte angewandt wird, für die er gedacht ist. Wendet man ihn irrtümlich auf falsche Objekte an, so kommt es zum Programmabbruch oder (noch schlimmer!) zu falschen Ergebnissen.
- In OCAML kann das Typsystem diese Verantwortung übernehmen.

Man definiert etwa

```
type 'a comparator = {compare : 'a -> 'a -> bool}
sucheVonBis : 'a array * 'a * int * int * 'a comparator -> bool
...
```

- Die neue Java-Version 1.5 (vorauss. Herbst '04) wird ML-artige Typvariablen (“*generics*”) beinhalten.

Quicksort

Ein von HOARE erstmals beschriebenes Sortierverfahren mit quadratischer Laufzeit im schlechtesten Fall, aber exzellenter mittlerer Laufzeit. In der Praxis für die meisten Anwendungen das beste Verfahren.

Idee: teile zu sortierenden Bereich in obere (O) und untere Hälfte (U), sodass alle Elemente von O oberhalb (in der Ordnung) von allen Elementen von U liegen.

Sortiere O und U rekursiv und hänge sie aneinander.

```
let rec split l x = match l with
  [] -> ([],[])
|h::t -> let (u,o) = split t x in
          if h < x then (h::u,o) else (u,h::o)
let rec qsort l = match l with
  [] -> []
| h::t -> let (u,o) = split t h in
          qsort u @ h :: qsort o
```

Quicksort

Eine effizientere Version ergibt sich, wenn mit Arrays gearbeitet wird und alle Operationen “in-place”, also an Ort und Stelle stattfinden. Man muss dann nämlich nach dem Sortieren nichts mehr zusammenfügen.

- Es gelte, $a[p..r]$ zu sortieren (gemäß einem Comparator).
- Wähle *Pivotelement* x .
- Arrangiere a um und bestimme q , sodass $a[p..q] \leq x \leq a[q+1..r]$. (Partitionieren)
- Sortiere rekursiv $a[p..q]$ und $a[q+1..r]$.
- Beachte: damit ist $a[p..r]$ sortiert.

Schwierigkeiten: 1. das Umarrangieren. 2. die Termination (Sicherstellen, dass $p \leq q$ und $q < r$).

Partitionieren

```
public static int partition(Object[] a, int p, int r, Object x, Co
    if (p==r) {
        if (c.compare(a[p],x) <= 0)
            {return p; }
        else
            {return p-1;}
    }
    else if (c.compare(a[p],x) < 0) return partition(a,p+1,r,x
    else if (c.compare(a[r],x) > 0) return partition(a,p,r-1,x
    else {
        Object hilf = a[p];
        a[p] = a[r];
        a[r] = hilf;
        return partition(a, p, r-1, x, c);
    }
}
```

Korrektheit

Die Vorbedingung lautet $p \leq r$.

Nach dem Aufruf

```
q = partition(a, p, r, x, c);
```

gilt folgendes:

- Die Reihenfolge der Einträge von $a[p..r]$ wurde verändert.
- Es gilt $a[p..q] \leq x \leq a[q+1..r]$ (im Sinne des Komparators c).
- Es gilt $p - 1 \leq q \leq r$.
- Existiert in $a[p..r]$ ein Element y mit $y \leq x$, so gilt $p \leq q$.
- Existiert in $a[p..r-1]$ ein Element y mit $y \geq x$, so gilt zusätzlich $q < r$.

Dies sieht man mit dem Satz von der partiellen Korrektheit (Info I).

Terminierung folgt mit der Abstiegsfunktion $r - p$.

Sortieren

Die Sortierfunktion ist jetzt ganz einfach:

```
public static void sort(Object[] a, int p, int r, Comparator c) {  
    if (p==r) ;  
    else {  
        int q = partition(a, p, r, a[p], c);  
        sort(a,p,q,c);  
        sort(a,q+1,r,c);  
    }  
}
```

Bewertung

Die Laufzeit von Quicksort ist *im Mittel* $O(n \log n)$, wobei $n = r - p$.

Im schlechtesten Fall ist die Laufzeit aber $O(n^2)$ (eine der beiden Partitionen hat immer die Größe eins).

Der große Vorteil von Quicksort gegenüber z.B. Mergesort (Info I) liegt darin, dass nach den beiden rekursiven Aufrufen keine Nachbearbeitung mehr erforderlich ist.

In der Praxis ist Quicksort (nach einigen Verbesserungen) das effizienteste Sortierverfahren.

Effizienzsteigerung von Quicksort

- Iterative Implementierung von `partition` mit While-Schleifen. (Bei effizienten Compilern nicht erforderlich, da `partition` ja bereits endrekursiv ist.)
- Zufällige Wahl des Pivotelements oder gar:...
- ... Wahl des Pivotelements als das mittlere von drei zufällig ausgewählten.
- Aber Vorsicht: man darf nicht das letzte Element als Pivot wählen (wg. Folie 276, Korrektheit). Besser durch Vertauschen das gewünschte Pivotelement in die Position `a[0]` bringen und dann mit `a[0]` als Pivot fortfahren.
- Verwendung eines anderen Sortierverfahrens bei $n < 10$. Dadurch weniger Verwaltungsaufwand an den Blättern des Rekursionsbaumes.