Generische Polymorphie in Java-1.5

Johannes Waldmann (HTWK Leipzig)

Plan

- Beispiel
- Collections, Iteratoren, Maps
- Schnittstellen und Implementierungen
- Genaueres zum Typsystem
- Details zu Implementierungen
- zur Didaktik

Anagramme (Aufgabe)

Aufgabe: finde alle Anagramme in einem Eingabestrom E!
Definition: Wort u ist Anagramm von Wort v, geschrieben $u \sim v$, falls man v durch Permutieren (Umstellen, unter Beachtung der Vielfachheit) der Buchstaben von u erhält.
Exakte Formulierung: Bestimme die Äquivalenzklassen von $E/\sim$ mit mehr als einem Element!

(ist klassische Übungsaufgabe aus: Jon Bentley: *Programming Pearls*, ACM Press 1995, 2000, `http://www.cs.bell-labs.com/cm/cs/pearls/`)

Anagramme (Ansatz)

Bestimme zu jedem Wort u der Eingabe das lexikografisch kleinste v mit $u \sim v$

```
static String key (String w) { char [] k = w.toCharArray();
  Arrays.sort(k); return new String (k); }
```

und benutze v als Schlüssel für die Klasse von u:
Hilfsdatenstruktur `book` ist eine (endliche, partielle) *Abbildung* von String (Schlüssel)
nach Menge von Strings (Äq-Klassen).

```
static SortedSet<Set<String>>
            anagrams (Set<String> ws) {
  Map<String, Set<String>> book = build_book (ws
  return groups (book);
}
```


Anagramme (Details)

```
static Map<String, Set<String>> build_book (Set<String> ws) {
  Map<String, Set<String>> book =
       new HashMap<String, Set<String>>();
  for (String w : ws) {
    String k = key (w);
    Set<String> prev = book.get(k);
    if (null == prev) {
      prev = new HashSet<String>();
    }
    prev.add(w);
    book.put(k, prev);
  }
  return book;
}
```


Collections: Überblick

Schnittstellen:	Implementierungen:
• `Collection<E>`	• –
• `List<E>`	• `LinkedList<E>`, `ArrayList<E>`
• `Set<E>`	• `HashSet<E>`
• `SortedSet<E>`	• `TreeSet<E>`

Listen: Zugriff über Index,
Mengen: Zugriff über Relationen (`equals`, `compareTo`) auf Elementen
beachte: Schnittstellen-Vererbung, aber keine Implementierungs-Vererbung!

Collections: Detail

```
interface Collection<E> {
    boolean isEmpty ();
    int size ();
    boolean add (E o);
    boolean contains (E o);
}

interface List<E> extends Collection<E> {
    E get (int i);
    void add (int i, E o);
}
interface Set<E> extends Collection<E> { }
```


Collections und Iteratoren

```
interface Collection<E> { ...
    Iterator<E> iterator();
}
interface Iterator<E> {
    boolean hasNext ();
    E next ();
}
Collection<E> c = ...
for (Iterator<E> it = c.iterator(); it.hasNext (); ) {
    E x = it.next (); ...
}
```

ab Java-1.5 einfach so: `for (E x : c) { ... }`

Maps (Überblick)

Abbildungen mit endlichem Definitionsbereich

```
interface Map<K,V> {
    V get (K key);    void put (K key, V value);
    boolean isEmpty (); int size ();
    Collection<V> values ();
    Set<K> keySet ();
}
```

Schnittstellen:

- `Map<K,V>`
  - `SortedMap<K,V>`

Implementierungen:

- `HashMap<K,V>`
  - `TreeMap<K,V>`

Schnittstelle vs. Implementierung

Prinzip des *information hiding*:
Programme sollten so wenig wie möglich über Interna verraten, z. B: welcher konkrete Datentyp benutzt wird.
In Deklaration (Typname Bezeichner = Initialwert):
Typname soll immer *abstrakt* (Schnittstelle) sein!

```
// schlecht (konkreter Typ ist sichtbar)
HashSet<String> c = new HashSet<String> ()

// gut (nur abstrakter Typ ist sichtbar)
Set<String> c = new HashSet<String> ();
```


Schnittstellen (II)

```
interface Map<K,V> { ...
    Set<Map.Entry<K,V>> entrySet ();
    interface Entry<K,V> {
        K getKey (); V getValue ();
}   }
```

versteckt Implementierung der Menge und der Paare:

```
static <K,V> Map<V,K> invert (Map<K,V> f) {
  Map<V,K> g = new HashMap<V,K> ();
  for (Map.Entry<K,V> e : f.entrySet()) {
     g.put (e.getValue(), e.getKey());
  }
  return g;
}
```


Polymorphie

Idee: Polymorph = vielgestaltig: *ein* Programm(-teil) ist auf *verschiedene* Daten(-typen) anwendbar.
Anwendung: Wiederverwendung.

- (ad-hoc-Polymorphie: Überladen von Bezeichnern)
- generische Polymorphie (hier gesehen)
- Vererbungs-Polymorphie

generische Polymorphie in Java seit 1.5 (= 5.0),
Vorläufer: Generic Java/Pizza, Templates in C++,
polymorph getypter Lambda-Kalkül (Haskell, ML)

Typ-Prüfung für generische Polymorphie

- nur zur *Laufzeit* (LISP usw.)
- zur *Übersetzungszeit*:
  bei *Anwendung* polymorpher Funktionen
  (d. h. *nach* Macro/Template-Expansion in C/C++)
- zur *Entwurfszeit*:
  bei *Schreiben* der polymorphen Funktionen
  (Haskell, ML, Java-1.5)

Design-Frage (scheinbar):
zwischen Flexibilität und Sicherheit

Generische Typen in Java-1.5

- strenge Unterscheidung zw *Typschablone* und *Typ*
- Schablone mit passender Argumentliste (= Folge von Typen) ergibt Typ
- in Objekt-Deklarationen/Konstruktor-Aufrufen dürfen (wie bisher) nur *Typen* (= instantiierte Schablonen) benutzt werden.

Klassen/Schnittstellen-Schablone:

```
interface Map<K,V> { ... }
```

Methodenschablone (Instantiierung durch Compiler)

```
static <K,V> Map<V,K> invert (Map<K,V> f) { .. }
```


Eingeschränkte Schablonen-Parameter

```
interface Foo <P> { ... } // <-- beliebige P
     // nur P, die Q implementieren:
interface Foo <P extends Q> { ... }
     // nur Ober-Schnittstellen von R:
interface Foo <P super Q> { ... }
```

benutzt Hierarchie auf Schnittstellen.
Jede Schnittstelle ist einstellige Relation (auf Typen), die Hierarchie besteht aus Implikationen.
Erweiterung: Schnittstellen sind *mehrstellige* Relationen (multi parameter type classes in Haskell)

Details zu Implementierungen

- Standard-Datentypen und Algorithmen (verkettete Listen, Hashtabellen, balancierte Suchbäume)
- Aha!-Algorithmen (Jon Bentley: *Programming Pearls*)
  z. B. für Rotation zyklischer Listen
  (Collections.rotate)
- OO-Entwurf(smuster), z. B. Delegation
  `TreeSet<E> -> TreeMap<E,Object>`,
  `TreeMap<K,V> -> Entry<K,V>`

Zur Didaktik

derzeit Java-1.5 als (wesentlicher) Teil von *Objektorientierte Konzepte* (4. Sem) bzw. als (kleinerer) Teil von *Compilerbau* (7. Sem)
(geplant: *Prinzipien von Programmiersprachen*, 4. Sem) ... *oder früher*?

- polymorphe Typen gibt es sowieso schon seit 1. Semester sowohl als Modell (Liste, Stack, Queue) als auch real (Array, Zeiger)
- Trennung abstrakter/konkreter Datentyp wird von Anfang an gelehrt, spätestens in Softwaretechnik / Praktikum benötigt, und kann in der Sprache direkt abgebildet werden