Unterabschnitt 5.3.7

Rekursive Klassen

Definition: (rekursive Klassendeklarationen)

Eine Klassendeklaration K heißt ***direkt rekursiv***, wenn Attribute von K den Typ K haben.

Eine Menge von Klassendeklarationen heißt ***verschränkt rekursiv*** oder ***indirekt rekursiv*** (engl. mutually recursive), wenn die Deklarationen gegenseitig voneinander abhängen.

Eine Klassendeklaration heißt ***rekursiv***, wenn sie direkt rekursiv ist oder Element einer Menge verschränkt rekursiver Klassendeklarationen ist.

Bemerkung:

- Wir identifizieren Klassen mit ihren Deklarationen.
- Wichtige Anwendung rekursiver Klassen ist die Implementierung von Listen-, Baum- und Graphstrukturen.

Implementierung von Listen

Im Folgenden betrachten wir rekursive Klassen für Listen. Dabei variieren wir die ***Programmierstile*** und die bereitgestellten Schnittstellen:

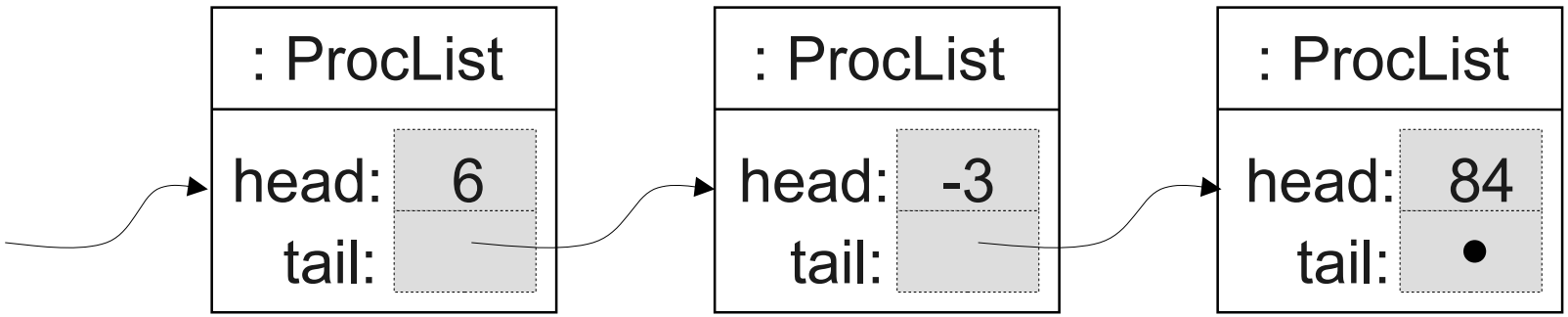
- prozedural
- funktional
- objektorientiert

Prozedurale einfachverkettete Liste

In der prozeduralen Programmierung sind Datentypen und Prozeduren zunächst getrennt (Zusammenfassung erst auf Modulebene).

Bei einfachverketteten Listen gibt es für jedes Listenelement ein Verbund/Objekt mit zwei Instanzvariablen:

- zum Speichern des Elements
- zum Speichern der Referenz auf den Rest der Liste.



Prozedurale einfachverkettete Liste (2)

```
class ProcList {
  int head;
  ProcList tail;
}

class ProcListMain  {

  static boolean sortiert(ProcList l){
    if( l == null || l.tail == null ) {
      return true;
    } else if( l.head <= l.tail.head ){
      return sortiert( l.tail );
    } else {
      return false;
    }
  }
  ...
```

Prozedurale einfachverkettete Liste (3)

```
public static void main( String[] argf ){
  ProcList l1 = new ProcList();
  ProcList l2 = new ProcList();
  ProcList l3 = new ProcList();

  l1.head = 1;
  l2.head = 2;
  l3.head = 3;
  l1.tail = l2;
  l2.tail = l3;
  l3.tail = null;

  System.out.println( sortiert(l1) );
  l3.head = 0;
  System.out.println( sortiert(l1) );
} }
```

Prozedurale einfachverkettete Liste (4)

Diskussion:

Die prozedurale Fassung erlaubt es jedem, der eine Referenz auf ein Listenknoten hat, das Objektgeflecht unkontrolliert zu verändern.

Zum Beispiel könnte man Listen in ein zyklisches Geflecht verändern und damit Invarianten verletzen.

Funktionale einfachverkettete Liste

Unterbindet man den beliebigen Zugriff auf die Attribute und bietet nur Methoden an, um Listen auf- und abzubauen, kann man ein Verhalten wie in der funktionalen Programmierung erreichen.

Das Mehr an Garantien wird durch weniger Flexibilität bezahlt. Insbesondere ist das direkte Einfügen und Modifizieren in der "Mitte" einer Datenstruktur nicht mehr möglich.

Funktionale einfachverkettete Liste (2)

```
class FunctionalList {
  private int head;
  private FunctionalList tail;

  static FunctionalList empty() {
    return new FunctionalList();
  }
  boolean isempty(){
    return tail == null;
  }
  int head(){
    if( isempty() ){
      throw new NoSuchElementException();
    }
    return head;
  }
```

Funktionale einfachverkettete Liste (3)

```
FunctionalList tail(){
  if( isempty() ){
    throw new NoSuchElementException();
  }
  return tail;
}

FunctionalList cons( int i ) {
  FunctionalList aux = new FunctionalList();
  aux.head = i;
  aux.tail = this;
  return aux;
}
```

Funktionale einfachverkettete Liste (4)

```
  static boolean sortiert( FunctionalList l ){
    if( l.isempty() || l.tail().isempty() ) {
        return true;
    } else if( l.head() <= l.tail().head() ){
      return sortiert( l.tail() );
    } else {
      return false;
    }
  }

}
```

Funktionale einfachverkettete Liste (5)

```
class FunctionalListMain  {

  public static void main( String[] argf ){
    FunctionalList le, l1, l2, l3;
    le = FunctionalList.empty();
    l3 = le.cons(3);
    l2 = l3.cons(2);
    l1 = l2.cons(1);
    System.out.println( sortiert(l1) );
    l1 = l3.cons(4);
    System.out.println( sortiert(l1) );
  }
}
```

Objektorientierte Listen

Aus objektorientierter Sicht ist eine Liste ein ***Behälter*** (engl. ***container***), in den man etwas hineintun und aus dem man etwas herausnehmen kann.

```
class SLinkedList {
  // Liefert erstes Element, Liste bleibt unveraendert
  int getFirst(){ ... }

  // Fuegt vorne ein neues Element an diese Liste an
  void addFirst( int n ) { ... }

  // Loescht das erste Element und liefert es zurueck
  int removeFirst() { ... }

  // Liefert die Elementanzahl dieser Liste
  int size() { ... }
}
```

Objektorientierte Listen (2)

Beispiel:

Anwendung von `SLinkedList`:

```
class SLinkedListMain  {
  public static void main( String[] argf ){
    SLinkedList l = new SLinkedList();
    l.addFirst(3);
    l.addFirst(2);
    l.addFirst(1);
    System.out.println( l.removeFirst() );
    System.out.println( l.size() );
    System.out.println( l.removeFirst() );
  }
}
```

Objektorientierte Listen (3)

Auszüge aus der Implementierung von `SLinkedList`:

```
class SEntry {  // Klasse fuer die Verkettung
  int head;
  SEntry tail;
}

class SLinkedList {

  private SEntry entries = null;
  private int size = 0;

  int getFirst(){
    if( size == 0 ){
      throw new NoSuchElementException();
    }
    return entries.head;
  }
```

Objektorientierte Listen (4)

```
  void addFirst( int n ) {
    size++;
    SEntry auxe = new SEntry();
    auxe.head = n;
    auxe.tail = entries;
    entries   = auxe;
  }

  int removeFirst() { ... }

  int size() { return size; }
}
```

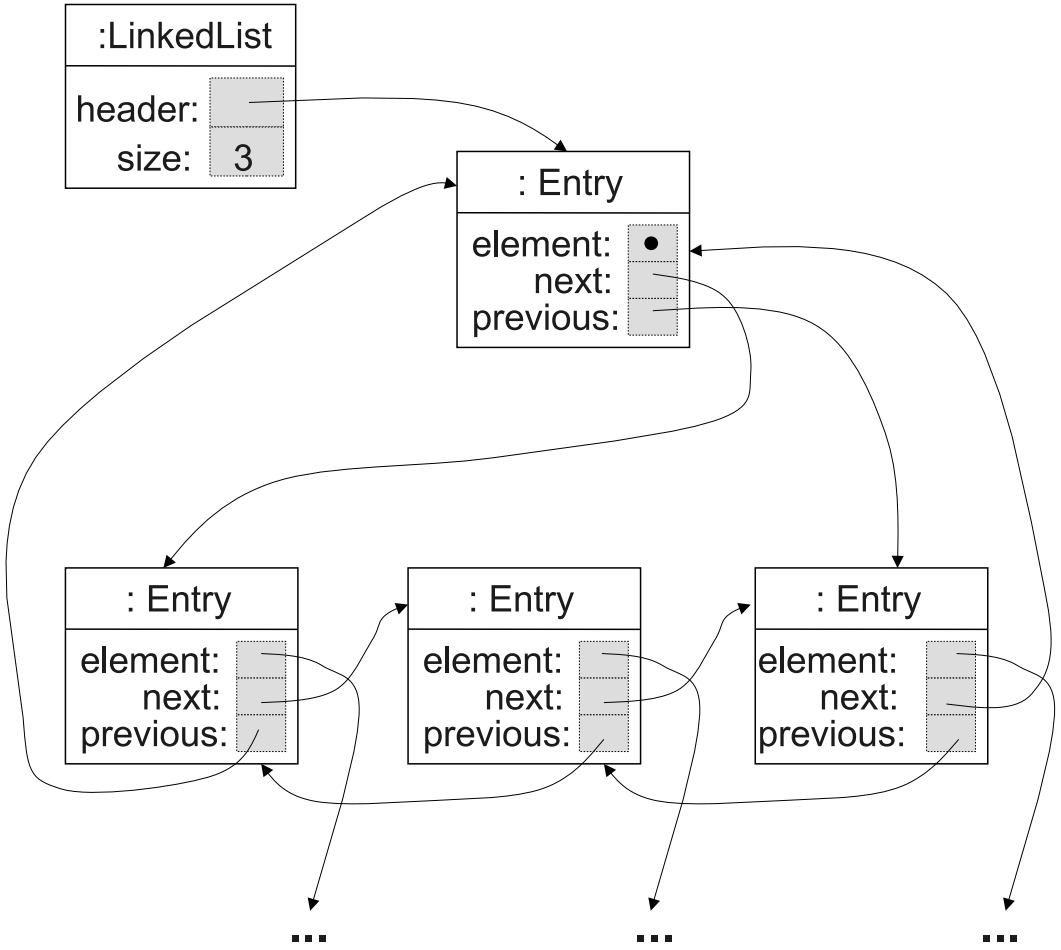
Objektorientierte Listen (5)

Diskussion:

Rekursives oder iteratives Durchlaufen durch die Liste ist bei der erläuterten Implementierung nicht möglich (Abhilfe: s. Iteratoren unten).

Andere Formen von Listenimplementierungen speichern die Elemente in Feldern oder nutzen eine doppelte Verkettung der Eintragsknoten:

Objektorientierte Listen (6)



Implementierung von Bäumen

Binäre Bäume sind wichtige Datenstrukturen, z.B. zur Darstellung von Mengen.

Beispiel: (Bäume)

```
class BinTree {
  private int elem;
  private BinTree left, right;

  BinTree( int e ) {
    elem = e;
  }
```

Implementierung von Bäumen (2)

```
void sorted_insert( int e ) {
  if( e < elem  ) {
    if( left == null ) {
      left = new BinTree(e);
    } else {
      left.sorted_insert(e);
    }
  } else if( elem < e ) {
   if( right == null ) {
      right = new BinTree(e);
    } else {
      right.sorted_insert(e);
    }
  }
}
// weiter auf naechster Folie
```

Implementierung von Bäumen (3)

```
  boolean contains( int e ) {
    if( e < elem && left != null ) {
      return left.contains(e);
    } else if( elem > e && right != null ) {
      return right.contains(e);
    } else {
      return e==elem;
    }
  }

  void printTree() {
    if( left != null ) left.printTree();
    System.out.println(elem);
    if( right != null ) right.printTree();
  }
}
```

Implementierung von Bäumen (4)

```
class BinTreeMain  {
  public static void main( String[] argf ){
    BinTree bt = new BinTree(12);
    bt.sorted_insert(3);
    bt.sorted_insert(12);
    bt.sorted_insert(11);
    bt.sorted_insert(12343);
    bt.sorted_insert(-2343);
    bt.sorted_insert(233);
    bt.printTree();
  }
}
```

Iteratoren

Iteratoren erlauben es, schrittweise über Behälterdatenstrukturen zu laufen, so dass alle Elemente der Reihe nach besucht werden.

Im Zusammenhang mit Kapselung (s.u.) sind sie unverzichtbar.

Beispiel: (Iteratoren)

Wir reichern die Klasse `SLinkedList` mit Iteratoren an und zeigen deren Anwendung:

```
class Iterator {
  private SEntry current;

  Iterator( SEntry se ) {
    current = se;
  }

  boolean hasNext() {
    return current!=null;
  }
```

Beispiel: (Iteratoren) (2)

```
  int next() {
    if( current==null ){
       throw  new NoSuchElementException();
    }
    int res = current.head;
    current = current.tail;
    return res;
  }
}

class SLinkedList { // Erweiterung um Iteratoren
  private SEntry entries = null;
  ...
  Iterator iterator() {
     return new Iterator( entries );
  }
}
```

Beispiel: (Iteratoren) (3)

```
class SLinkedListMain  {
  public static void main( String[] argf ){
    SLinkedList l = new SLinkedList();
    l.addFirst(3);
    l.addFirst(2);
    l.addFirst(4);

    Iterator iter = l.iterator();
    while( iter.hasNext() ) {
      System.out.println( iter.next() );
    }
} }
```

Bemerkung:

Der Iterator muss Zugriff auf die interne Repräsentation der Datenstruktur haben, über die er iteriert.

Unterabschnitt 5.3.8

Typsystem von Java und parametrische Typen

Typsystem von Java

Werte in Java sind

- die Elemente der elementaren Datentypen,
- Referenzen auf Objekte,
- der Wert null.

Jeder Wert in Java gehört zu mindestens einem Typ. Vordefinierte und benutzerdeklarierte Typen sind

- die vordefinierten elementaren Typen,
- die durch Klassen deklarierten Typen,
- die durch Schnittstellen deklarierten Typen (s.u.).

Typsystem von Java (2)

Implizit deklariert sind die ***Feldtypen*** zu den Klassen- und Schnittstellentypen (Typkonstruktor „[]“).

Feld-, Klassen- und Schnittstellentypen fasst man unter dem Namen Referenztypen zusammen. (null gehört zu allen Referenztypen.)

Klassen- und Schnittstellentypen beschreiben, welche Nachrichten ihre Objekte verstehen bzw. welche Methoden sie besitzen.

Bemerkung:

In typisierten objektorientierten Sprachen können Werte zu mehreren Typen gehören (Subtypen).

Parametrische Typen

Auch objektorientierte Sprachen unterstützen parametrische Typsysteme wie in Haskell. Für Java ist eine derartige Unterstützung ab Version 1.5 verfügbar.

Beispiel: (Parametrische Typen)

Wir betrachten eine parametrische Fassung der Klasse SLinkedList:

```
class SLinkedList<A> {

  A getFirst(){ ... }

  void addFirst( A n ) { ... }

  A removeFirst() { ... }

  int size() { ... }
}
```

Beispiel: (Parametrische Typen) (2)

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

   SLinkedList<String> l1 = new SLinkedList<String>();
   l1.addFirst("Die Ersten werden");
   l1.addFirst("die Letzten sein");
   int i = l1.getFirst().indexOf("sein"); // liefert 13

   SLinkedList<W3Seite> l2 = new SLinkedList<W3Seite>();
   l2.addFirst(new W3Seite("Titel","Inhalt"));
   l2.addFirst(new W3Seite("Title","Content"));
   int i = l2.getFirst().indexOf("sein");
         // liefert Uebersetzungsfehler
 }
}
```