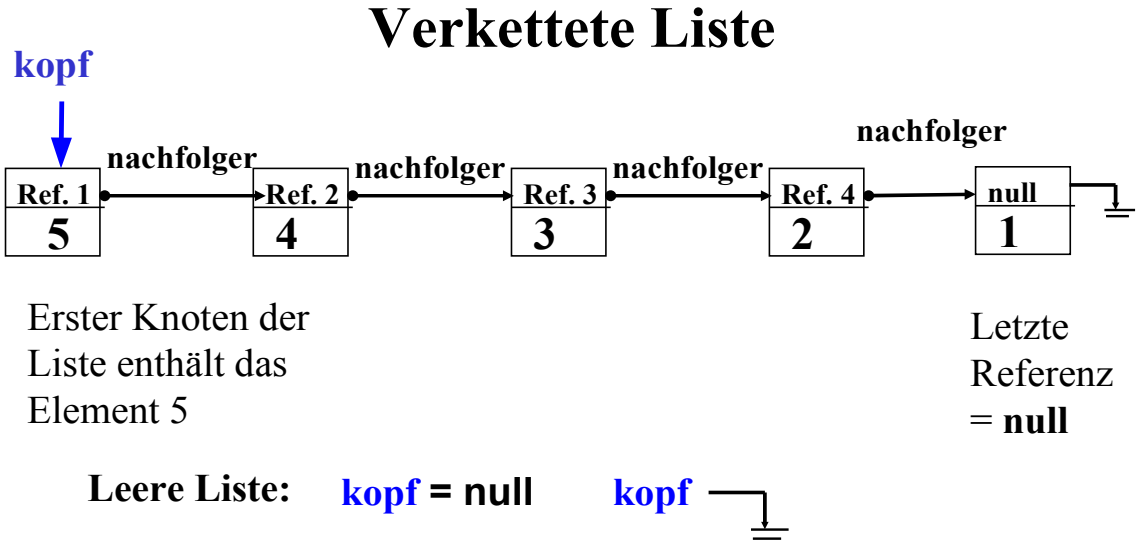
Der Datentyp verkettete Liste (ADT Liste)

Eine verkettete Liste von Zahlen



- Die Referenz **kopf** zeigt auf den ersten Knoten
- Jeder Knoten enthält neben seinem ***Datenfeld (Inhalt)*** eine Referenz (***Verweis***) auf den nächsten Knoten.
- Bei einer **leeren Liste** gibt es keinen Kopfknoten: **kopf = null** .

Konzept

Ein **Knoten** ist eine Struktur, bestehend aus einem **Datenelement** und einer **Referenz** auf einen Nachfolgeknoten.
Eine verkettete Liste ist eine Menge von Knoten, die (wie beim Feld) sequenziell angeordnet sind. Bei der verketteten Liste ist jeder Knoten mit seinem Nachfolgeknoten verkettet.

Die Knoten bestehen aus zwei Komponenten:

Referenz auf den nächsten Knoten und aus dem **Datenfeld**.

Über die Referenzvariable **kopf** (head) ist die gesamte Liste erreichbar.

Die Referenz zeigt auf den Rest (tail) der Liste. Vom Kopf aus kann man die gesamte Liste durchlaufen.

Die Nachfolgereferenz des letzten Knotens ist leer (null).

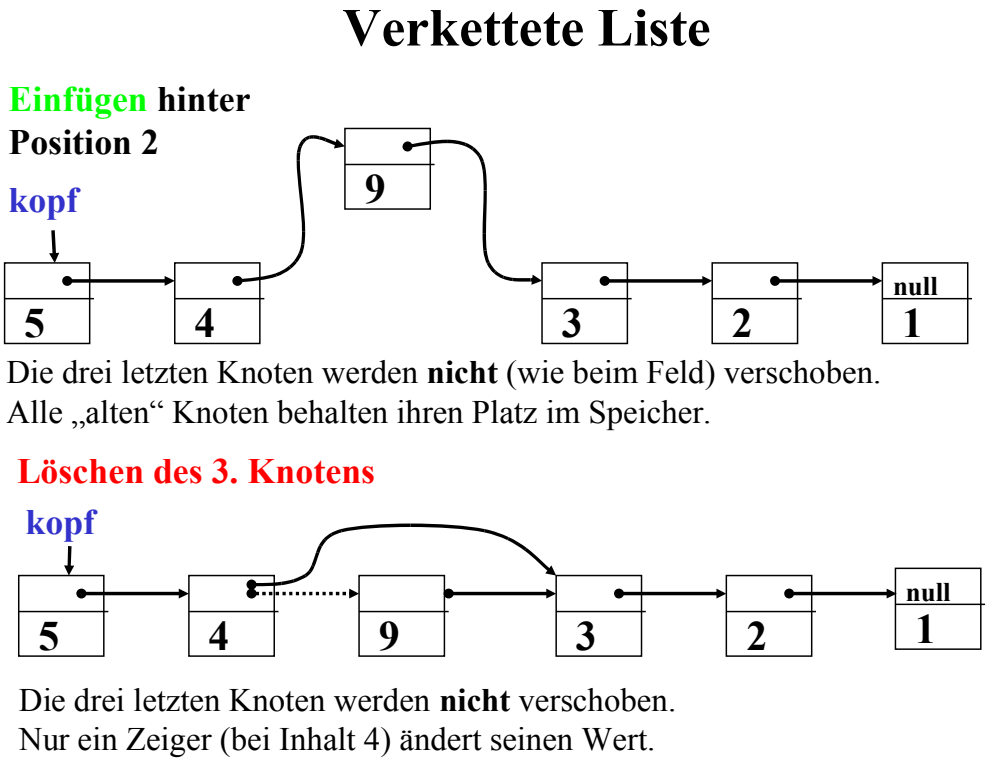
Als Sonderfall kann eine verkettete Liste auch **leer** sein!

Wir stellen die Knoten in Zukunft durch Kästchen und die Verkettungen durch Pfeile dar.

Sofern die Liste nicht leer ist, gilt:

- **Es gibt einen ersten Knoten (ohne Vorgänger)**
- **Es gibt einen letzten Knoten (ohne Nachfolger)**
- **Jeder 'innere' Knoten hat genau einen Vorgänger und genau einen Nachfolger**

Einfügen und Löschen eines Knotens



Das JAVA-Laufzeitsystem erkennt, dass auf das Knotenelement **9** keine Referenz mehr existiert und löscht den Knoten.

Vergleich mit Feldern

Arrays

- Vordefinierte, feste Anzahl von Elementen
- Die Feldgröße kann nachträglich nicht verändert werden
- Löschen von Elementen nicht möglich

Listen

- Ihre Größe kann zu- und abnehmen
- Beliebige Knotenzahl möglich
- Elemente können in effizienter Weise umgeordnet werden
- einziger Nachteil: Der Zugriff muss über den Listenkopf erfolgen.

Der abstrakte Datentyp Liste (ADT Liste)

Um Implementierungsdetails vor dem Programmierer zu verbergen, werden sog. ***abstrakte Datentypen*** eingeführt.

Es ist oft zweckmäßiger, Datenstrukturen mit Hilfe der ***Operationen*** zu beschreiben, als durch Einzelheiten der Implementierung. Wenn eine Datenstruktur auf diese Weise beschrieben wird, wird sie ***abstrakter Datentyp*** genannt.

Die Grundidee ist, dass das, was die Datenstruktur leisten soll, von der Implementierung getrennt ist. Die "von Außen" zugelassenen Operationen werden über wohldefinierte Methodenaufrufe abgewickelt (**Kapselungsprinzip, Geheimnisprinzip**). Die Kapselung versteckt die innere Struktur und Implementierung des Datentyps. Zugriffe dürfen nur über die **Schnittstelle** erfolgen.

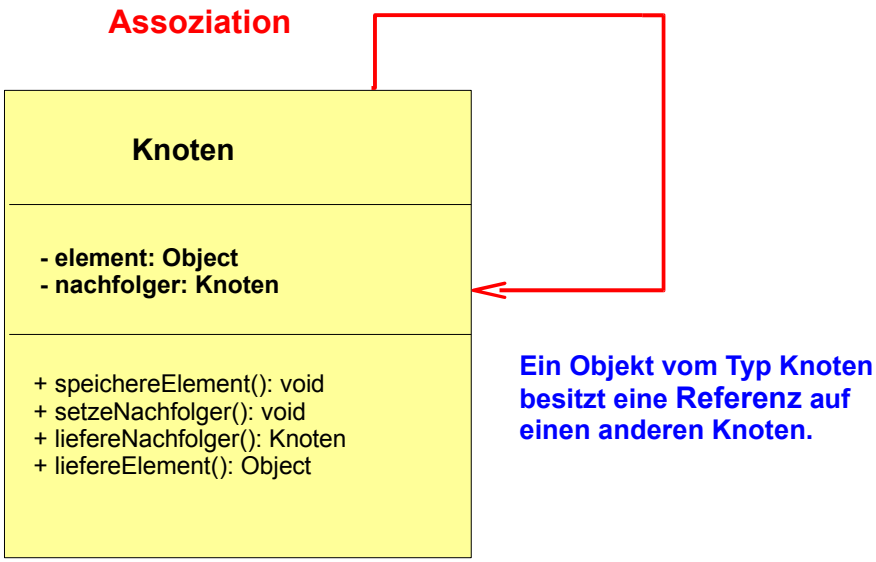
- **Ein abstrakter Datentyp (ADT) ist die Zusammenfassung der Eigenschaften und Operationen einer Datenstruktur.**
- **Man kann einen ADT als *Blackbox mit Gebrauchsanweisung* verstehen. Der Programmierer verwendet diesen Datentyp gemäß seiner veröffentlichten Spezifikationen.**
- **Ein ADT kann ohne Wissen des Anwenderprogrammierers gegen eine neue Version ausgetauscht werden.**

Implementierung des Datentyps Liste

Wir modellieren zunächst eine Klasse Knoten

```
public class Knoten {
//        Klasse, die einen Knoten darstellt
//        Ein Zugriff auf element und nachfolger
//        ist nur über Methoden möglich!


        private Object element;    //(fast) beliebiges Datenelement
        private Knoten nachfolger; // Assoziation & Rekursion!

        // Die Konstruktoren:

        Knoten() {
        //        Leerer Knoten , kein Nachfolger
          this(null, null); //element=null und nachfolger = null
        //         ruft den nachfolgenden Konstruktor auf
        }

        Knoten(Object element, Knoten nachfolger) {
          this.element=element;
          this.nachfolger=nachfolger;
        }

        public void speichereElement(Object element) {
          this.element = element;   // Element im Knoten ablegen
        }

        public void setzeNachfolger(Knoten nachfolger) {
          this.nachfolger=nachfolger; // aktuellen Knoten mit
          // Knoten(referenz) der Parameterliste verketten
        }

        Knoten liefereNachfolger() {
          return nachfolger;
        }

        Object liefereElement() {
          return element;
        }

} // Knoten
```

Klassendiagramm Knoten



Anmerkungen zur Klasse Knoten:

1. Jede Klasse, auch die Klasse ***Knoten,*** ist Unterklasse von ***Object***. Dies bedeutet, dass sich der Typ der Variablen **element** zur Laufzeit spezialisieren lässt (z.B. als *int* oder als *Schueler* etc.).

2. Die Variable **nachfolger** ist hat den Typ ***Knoten.*** *S*ie ist **nur** eine **Referenzvariable** (=Adresse) auf den Nachfolgeknoten!
Es liegt eine **Assoziation** vor, da kein Knoten innerhalb der Klasse erzeugt wird!

Somit kann man Objekte vom Typ Knoten über diese Referenzen zu einer ***Liste*** verketten.

3. **this (null, null)** ruft den zweiten Konstruktor mit den Parameterwerten **null** auf.

Ein anschauliches Beispiel:

Referenzen und Objekte

- Deklaration `Topf t;  // t ist der Objektname`
  → Der Handle (Griff, Referenz) t wird erzeugt; **der Topf existiert noch nicht!**

- Zuordnung von Speicherplatz über Schlüsselwort „new“
  Syntax: **objektname = new Klassenname( );**
  t = new Topf(); // jetzt existiert der Topf (mit Griff)

  → Bereitstellung von Speicher für das Objekt
  → Verbinden des **handle** mit dem Objekt

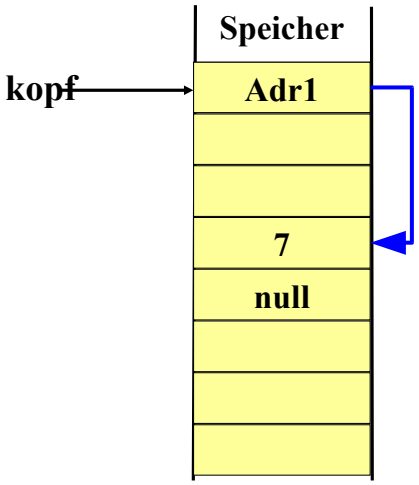
  In Java gibt es Objekte nur zusammen mit einem **Handle**, einer Referenz!

Übungen

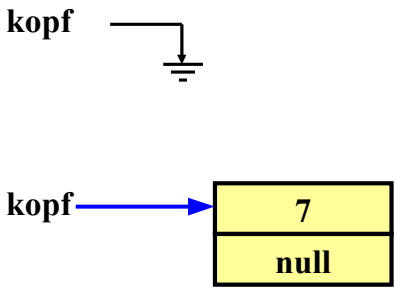
Variable vom Typ Knoten

Knoten kopf;

kopf = new Knoten(7, null);



Kurzform



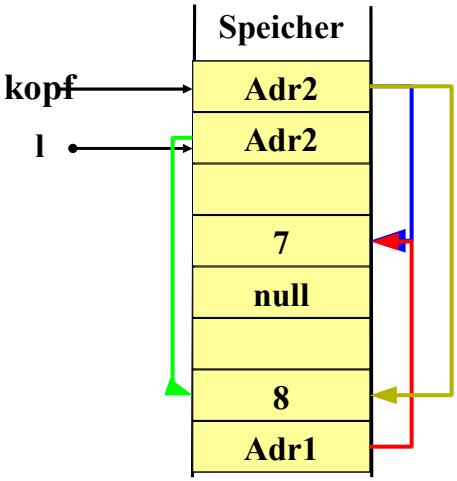
2 Variablen vom Typ Knoten

Knoten kopf;

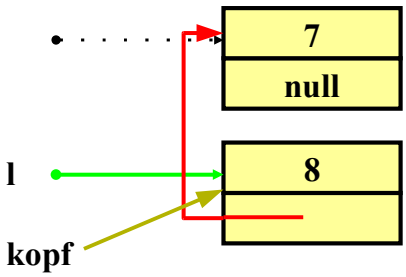
Kopf = new Knoten(7, null);

Knoten l = new Knoten(8, kopf);

kopf = l;



Kurzform



Aufgabe
Was ist in folgenden Anweisungen falsch?

```
Knoten k = new Knoten;
Knoten p; p= null;
k.nachfolger=p;
```

Zeichne die fertige Struktur:

```
Knoten ende = new Knoten('c', null);
Knoten hilf = new Knoten('b', ende);
Knoten k = new Knoten('a', hilf);
ende.setzeNachfolger(hilf);
```

Schritt 2: Wir spezifizieren den abstrakten Datentyp Liste0:

Beschreibung der Operationen

Konstruktor

Liste0( ) erzeugt leere Liste, kopf = null, kein Nachfolger

Beschreibung der Operationen/Methoden

```
public boolean listeLeer(Knoten k);  //TRUE, wenn Liste leer
public void einfuegen(Object element); // Fügt Knoten am Listenkopf an

public boolean einfuegen(Object element, int index);
// Fügt Knoten hinter Stelle index ein; index >=0;
// index =0 Einfügen am Listenkopf

public int suchen(Object element);
//gibt Index des Vorkommens an; -1, wenn element nicht in Liste vorhanden

public boolean loeschen(Object element);
//True, wenn element gelöscht wurde

public String toString(Knoten k)

// Gibt die Listenelemente aus
```

Beispiel

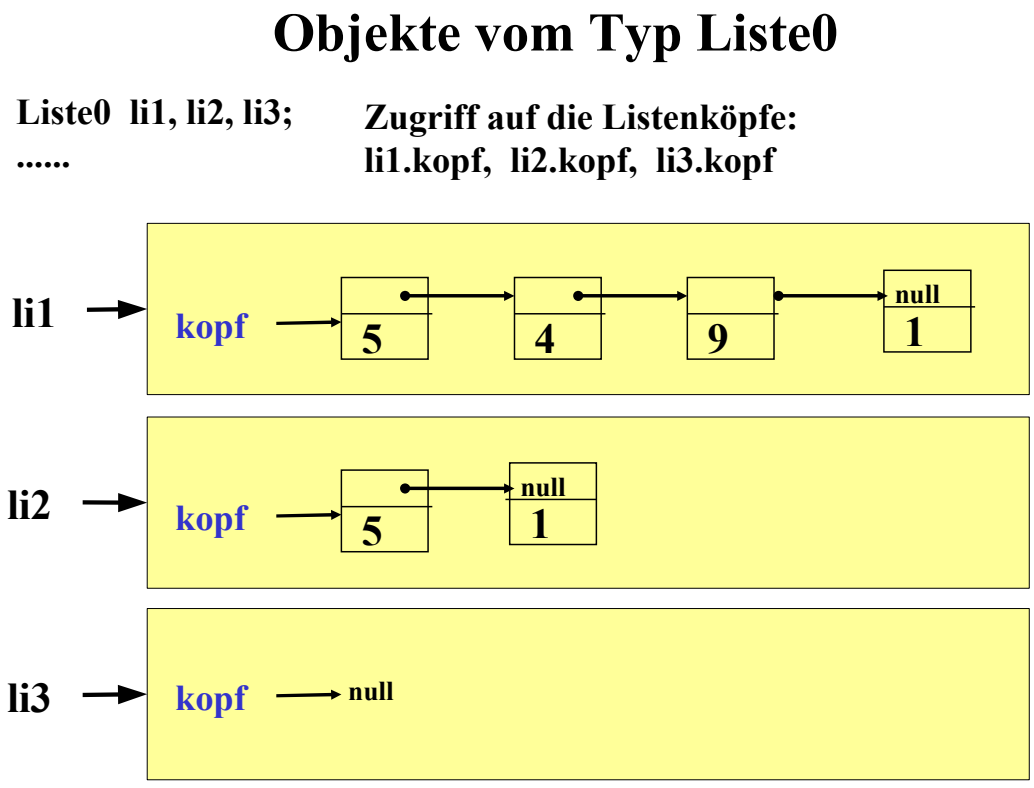
Schrittweise Erzeugung der Liste (Knotenelemente 5 bis 1)

Hinweise: Die JAVA-Klasse hat den Namen **Liste0**

Der Konstruktoraufruf: `Liste0 li = new Liste0( )`

erzeugt eine leere Liste mit dem Namen **li** mit dem Kopf **kopf (**Die Knotenelemente sind vom Typ **String)**.

```
li.einfuegen("1");
li.einfuegen("2");
li.einfuegen("3");
li.einfuegen("4");
li.einfuegen("5");
```



Aufgaben

Geben Sie für obige Liste die ADT-Operationen zu folgenden Aufgaben an

A1: Suchen der Position des Knotens mit dem Wert "4".

A2: Einfügen eines Knotens mit dem Inhalt 9 hinter dem zweiten Knoten.

A3: Löschen des Knotens mit dem Inhalt "9".

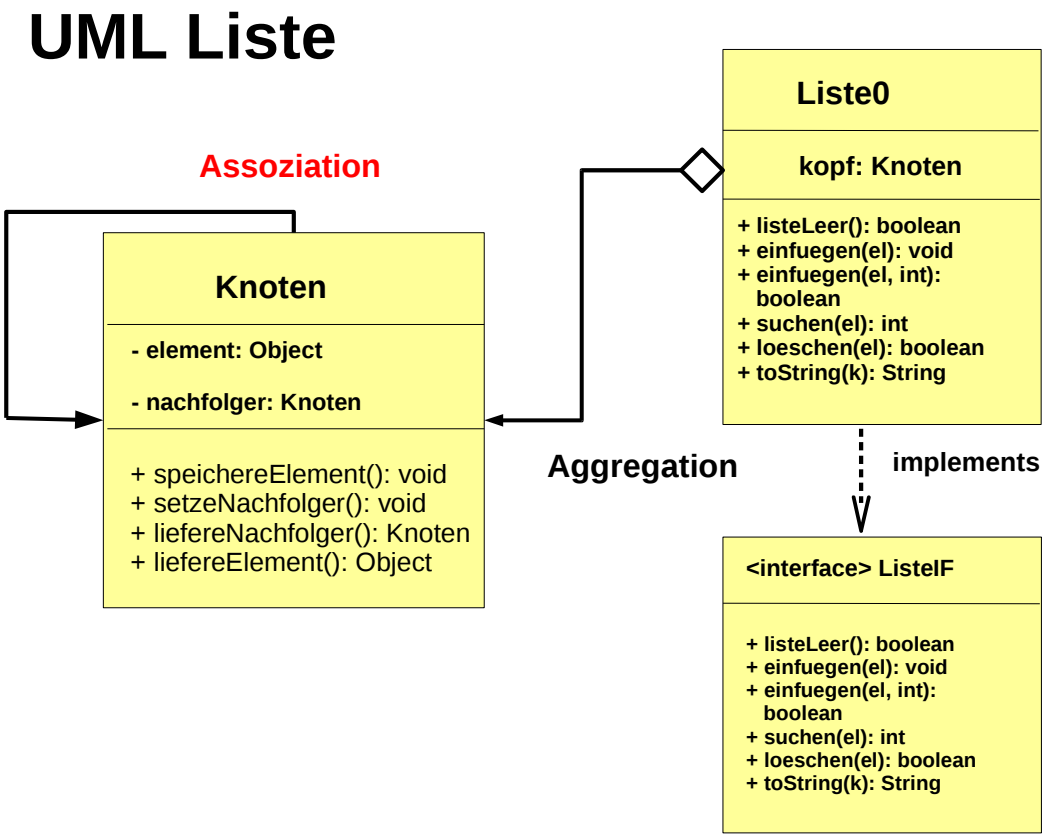
Die Implementierung der Liste

Die Struktur des Programmierprojekts

1. Entwurf der Klasse **Knoten**
2. Formulierung des Interfaces **ListeIF**
3. Die Klasse **Liste0** (implementiert den ADT, der im Interface spezifiziert ist)
4. Das Applet **ListenTest** mit graphischer Oberfläche

Die Klasse Liste0

Wir modellieren die Klasse **Liste0**, welche die Hilfsklasse **Knoten** verwendet und listentypische Operationen bereitstellt.



Wir nutzen das **Interface-Konzept** von Java zur Implementierung des ADT Liste0:

Interfaces in Java sind reine Schnittstellen. Sie enthalten **keine Implementierung**, sondern nur die Angabe sog. **abstrakter Methoden**. Die Definition von Interfaces ist von der Struktur her identisch zu Klassen und beginnt mit dem Schlüsselwort **interface**.

Die Interfacedefinition ist in Java nicht zwingend vorgeschrieben. Interfaces sind aber sinnvoll, da sie die Operationen spezifizieren, die der ADT unterstützt.

Ein Interface stellt die Vereinbarung zu implementierender Methoden dar!

Das Prinzip des Zugriffs über die in der Schnittstelle angegebenen Operationen heißt ***Kapselung***. Sie versteckt die innere Struktur der Klasse.

```
public interface ListeIF {
        /* Die Operationen
           Im Interface sind die Elementtypen noch nicht bekannt
           es wird die Oberklasse Object benutzt
        */
        public boolean listeLeer(Knoten k);  //TRUE, wenn Liste leer
        public void einfuegen(Object element); // Fügt Knoten am Listenkopf an

        public boolean einfuegen(Object element, int index);
        // Fügt Knoten hinter Stelle index ein; Index >=0

        public int suchen(Object element);
        //gibt Index des Vorkommens an; -1, wenn element nicht in Liste vorhanden

        public boolean loeschen(Object element);
        //True, wenn element gelöscht wurde

        public String toString(Knoten k)
        // Gibt die Listenelemente aus
} // Interface
```

Bemerkungen

Die Klasse ***Object*** bildet die Wurzel der Objekthierarchie in Java. Von ihr sind alle Subklassen abgeleitet. Die Klasse ***Object*** hat auf Grund ihrer Position als Wurzelklasse eine zentrale Bedeutung. Sie kann verwendet werden, um Datenstrukturen zu definieren, die ***Objekte beliebigen Typs*** aufnehmen können.

Der Vorteil besteht darin, dass der Typ der Variablen ***element*** erst später (z.B. als int, double, String, ...) festgelegt werden muss.

Implementierung der Listenklasse **Liste0** , welche als Basis obige Knotenklasse verwendet und das Interface **ListeIF** implementiert.

Die Liste OHNE Dummy-Elemente realisiert

```
public class Liste0 implements ListeIF {

     public Knoten kopf;

//    Konstruktor
     public Liste0() { kopf=null; }


     public boolean listeLeer(Knoten k) {
          // TODO Auto-generated method stub
          if (k == null)
               return true;
          else return false;
     }


     public void einfuegen(Object element) {
          // TODO Auto-generated method stub
            Knoten k = new Knoten(element,null);
            if (kopf == null)
               kopf = k;
            else {
               k.setzeNachfolger(kopf);
               kopf=k;
            }
     } // einfuegen
```

```
public boolean einfuegen(Object element, int index) {
// Fügt Knoten hinter Stelle index ein; Index >=0
        Knoten k = new Knoten(element, null);

        Knoten hilf, nach;
        hilf=kopf;nach=hilf;
        if (kopf == null) {
          return false;
        }
        else  {  // else 1
          // mindestens ein Knoten in der Liste
          if (index >=0) {
             for (int i=0; i<index; i++) {
               nach=hilf;hilf=hilf.liefereNachfolger();
               if( (hilf==null) && (i<index)) { // ungültige Position
                  return false;
                } // if
              } // for
               // Knoten einfügen
                nach.setzeNachfolger(k);

                k.setzeNachfolger(hilf);
                return true;
           } //if index...
          else return false;
         } // else 1
} // einfuegen
```

```
public int suchen (Object element) {
        Knoten hilf;
        boolean gefunden = false;

        int index = -1;
        if (kopf !=null) {
          index =0;hilf=kopf;
          do {
            if (element.equals(hilf.liefereElement()) ) {
                gefunden=true; return index;
            }
            else {
                hilf=hilf.liefereNachfolger(); index++;
            }
          }  while (hilf != null);
          if (gefunden) return index; else return -1;
         } // if
         else return -1;
       }  // suchen

public boolean loeschen(Object element) {
        Knoten hilf, nach;

        if (kopf != null) {
         if( element.equals(kopf.liefereElement()))  {
           // zu löschendes Element ist am Kopf
           kopf=kopf.liefereNachfolger(); return true;
         }
         else  {
           // zu löschendes Element hinter dem Kopf
            hilf=kopf.liefereNachfolger();nach=kopf;
            while( hilf != null) {
               if (element.equals(hilf.liefereElement())) {
                 // jetzt kann gelöscht werden
                  nach.setzeNachfolger(hilf.liefereNachfolger());
                  return true;
                }
               else { // weiter durch liste
                  nach=hilf; hilf=hilf.liefereNachfolger();
                }
             } //while
           } // else
          return false;
        }
        else return false;
} // loeschen
```

```
public String toString(Knoten k) {
        // durchläuft die Liste ab der Referenz k

        String s="( ";
      Knoten x = k;

      while(x != null) {
        s = s+x.liefereElement().toString()+" ";  // Methode der Klasse Object
        x= x.liefereNachfolger();
      }

        s=s +" )";
        return s;
}  // toString

} // Liste0
```

Die Liste mit Dummy-Elementen an Anfang & Ende:

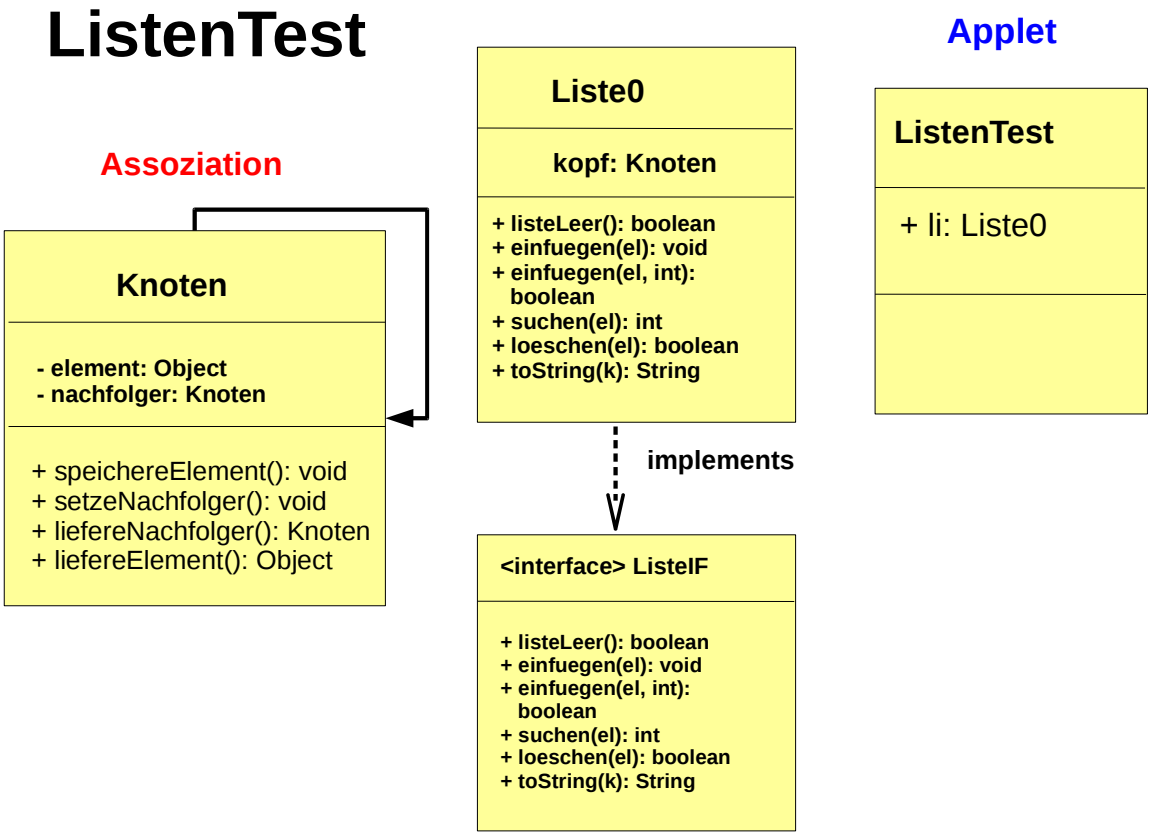
```
public class Liste1 implements ListeIF {
        // Liste mit Dummy-Element am Anfang und Ende

        public Knoten kopf;

//       Konstruktor
        public Liste1() {
                Knoten hilf = new Knoten(null, null);
                kopf= new Knoten(null, hilf);
        }


        public void einfuegen(Object element) {
          Knoten hilf = new Knoten(element, null);
           hilf.setzeNachfolger(kopf.liefereNachfolger());
           kopf.setzeNachfolger(hilf);
        }


        public boolean einfuegen(Object element, int index) {
                // TODO Auto-generated method stub
                // Einfügen HINTER Stelle 'index'
                // Der Dummy-Knoten hat die Position 0
                  Knoten k = new Knoten(element, null);
                  Knoten hilf, nach;
                  hilf=kopf;nach=hilf;

       for (int i=0; i<=index; i++) {
          nach=hilf;hilf=hilf.liefereNachfolger();
          if( (hilf==null) && (i<index))  { // ungültige Position

                  return false;
            }
          } // for

       // Knoten einfügen
       nach.setzeNachfolger(k);

       k.setzeNachfolger(hilf);
       return true;


          } // einfügen
```

```
public boolean listeLeer(Knoten k) {
        // TODO Auto-generated method stub
        if ( k.liefereNachfolger().liefereNachfolger()==null)

        return true;
        else return false;

} // listeLeer



public boolean loeschen(Object element) {
         Knoten hilf, vorgaenger;
         boolean geloescht=false;

         vorgaenger =kopf; hilf=vorgaenger.liefereNachfolger();
         while(hilf.liefereElement() != null ) {

                 if(element.equals(hilf.liefereElement())) {
                         // es kann gelöscht werden
                         vorgaenger.setzeNachfolger(hilf.liefereNachfolger());
                         // NICHT: geloescht =true;
                         return true;
                 }
                 else {
                          vorgaenger=hilf;
                          hilf=hilf.liefereNachfolger();
                 }
         } // while
          return geloescht;

}  // loeschen
```

```
    public int suchen(Object element) {

            Knoten hilf=kopf.liefereNachfolger();
            int index = 0;

            while(hilf.liefereElement() != null ) {
                    index++;

                    if(element.equals(hilf.liefereElement())) {
                            // Element gefunden
                            return index;
                    }
                    else {
                             hilf=hilf.liefereNachfolger();
                    }
            } // while
            return -1;

    } // suchen



    public String toString(Knoten k) {
            // durchläuft die Liste ab der Referenz k

            String s="( ";
            Knoten x = k;

            while(x != null) {
                    s = s+x.liefereElement().toString()+" ";
                    x= x.liefereNachfolger();
             }

      s=s +" )";
              return s;

    }  // toString

}  // Class
```

Das komplette Applet mit dem Namen **ListenTest**

Achtung: Die Beziehungen fehlen (bitte noch ergänzen)!



Im Konstruktor des Applets

```
li = new Liste0( );  // leere Liste erzeugen
```

Der Button Einfügen

```
protected void jButton1ActionPerformed(ActionEvent evt){

  li.einfuegen(jTextField1.getText());
  jTextField3.setText(li.toString(li.kopf));

  jTextField1.requestFocus();
  jTextField1.selectAll();

} // einfügen
```

Der Button Löschen

```
protected void jButton2ActionPerformed(ActionEvent evt){

String s = jTextField1.getText();
   if (!li.listeLeer(li.kopf) ) {
       if (li.loeschen(s))
         jTextField2.setText(s+" gelöscht");
       else  jTextField2.setText(s+" nicht vorhanden");

       jTextField3.setText("LÖSCHEN: "+li.toString(li.kopf));
   } // if
   else jTextField2.setText("Liste ist LEER");
jTextField1.requestFocus();
jTextField1.selectAll();
} // löschen
```

Der Button Suchen

```
protected void jButton3ActionPerformed(ActionEvent evt){
        //TODO add your handler code here
        int stelle;
        String s = jTextField1.getText();
        if( !li.listeLeer(li.kopf) ) {
                stelle = li.suchen(s);
                if (stelle>=0)
                        jTextField2.setText(s+" an Pos. "+stelle);
                else jTextField2.setText(s+" NICHT in Liste: "+stelle);

        } // if
        else jTextField2.setText("Suchen: Liste ist leer!");

jTextField3.setText("Suchen "+li.toString(li.kopf));
jTextField1.requestFocus();
jTextField1.selectAll();

} // suchen
```

Schreibe eine **rekursive** Methode, die eine Liste von von hinten nach vorne ausgibt.