Listen

Dynamische Daten, verkettete Listen, Listen mit Cursor, ListenIteratoren, Doppelt verkettete Listen, ein Adressbuch mit GUI, Skip-Listen, adaptive Listen, Listen als Werte, rekursives Programmieren, Hashing

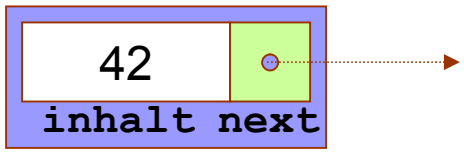
Behälter mit Standardreihenfolgen

- Listen
  - Standardreihenfolge
  - veränderbare Größe
  - kein schneller Zugriff auf die Elemente

- Bäume
  - Hierarchie und meist Standard-Reihenfolge der Söhne
  - Elemente
    - in Knoten oder in Blättern oder in Knoten und in Blättern
  - Reihenfolge der Söhne gibt Anlass zu Standardreihenfolgen
    - pre-order, in-order, postorder
  - Hierarchie gibt Anlass zu Standardreihenfolge
    - depth-first und breadth-first

Verkettete Listen

- Listen sind Folgen von Elementen
- Im Unterschied zu Arrays können Listen beliebig wachsen und schrumpfen
  - Neue Elemente werden an die Liste angehängt
  - Nicht mehr benötigte Elemente können entfernt werden
- Verkettete Listen kann man sich wie eine Perlenschnur vorstellen
  - Von jeder Perle gibt es eine Verbindung zur folgenden
  - In den Perlen befindet sich die Information
- Die Liste besteht aus Zellen.
  - Jede Zelle hat einen Inhalt, und
  - Einen Verweis auf die folgende Zelle



Die Zelle – unsere Perle

- Die Liste besteht aus Zellen.
  - Jede Zelle hat einen Inhalt, und
  - einen Verweis auf die folgende Zelle
  - Zelle ist eine rekursive Datenstruktur

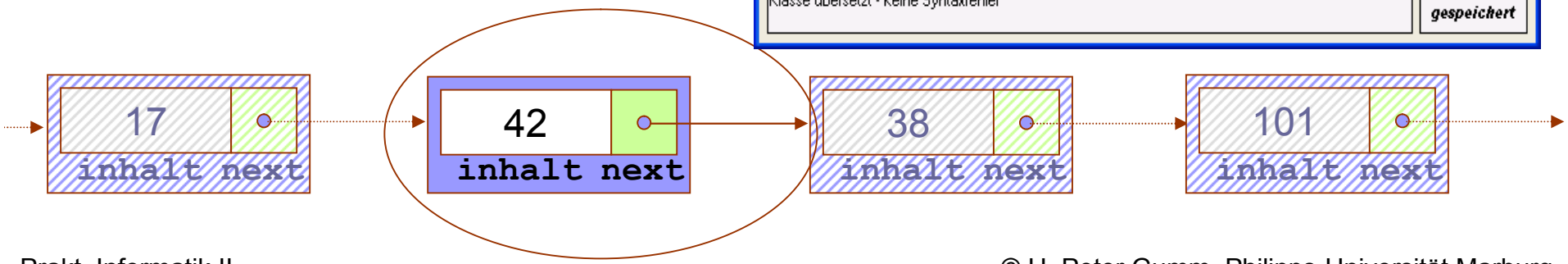
Zelle

Klasse Bearbeiten Werkzeuge Optionen

Übersetzen | Rückgängig | Ausschneiden | Kopieren | Einfügen | Suchen... | Weitersuchen | Schlie

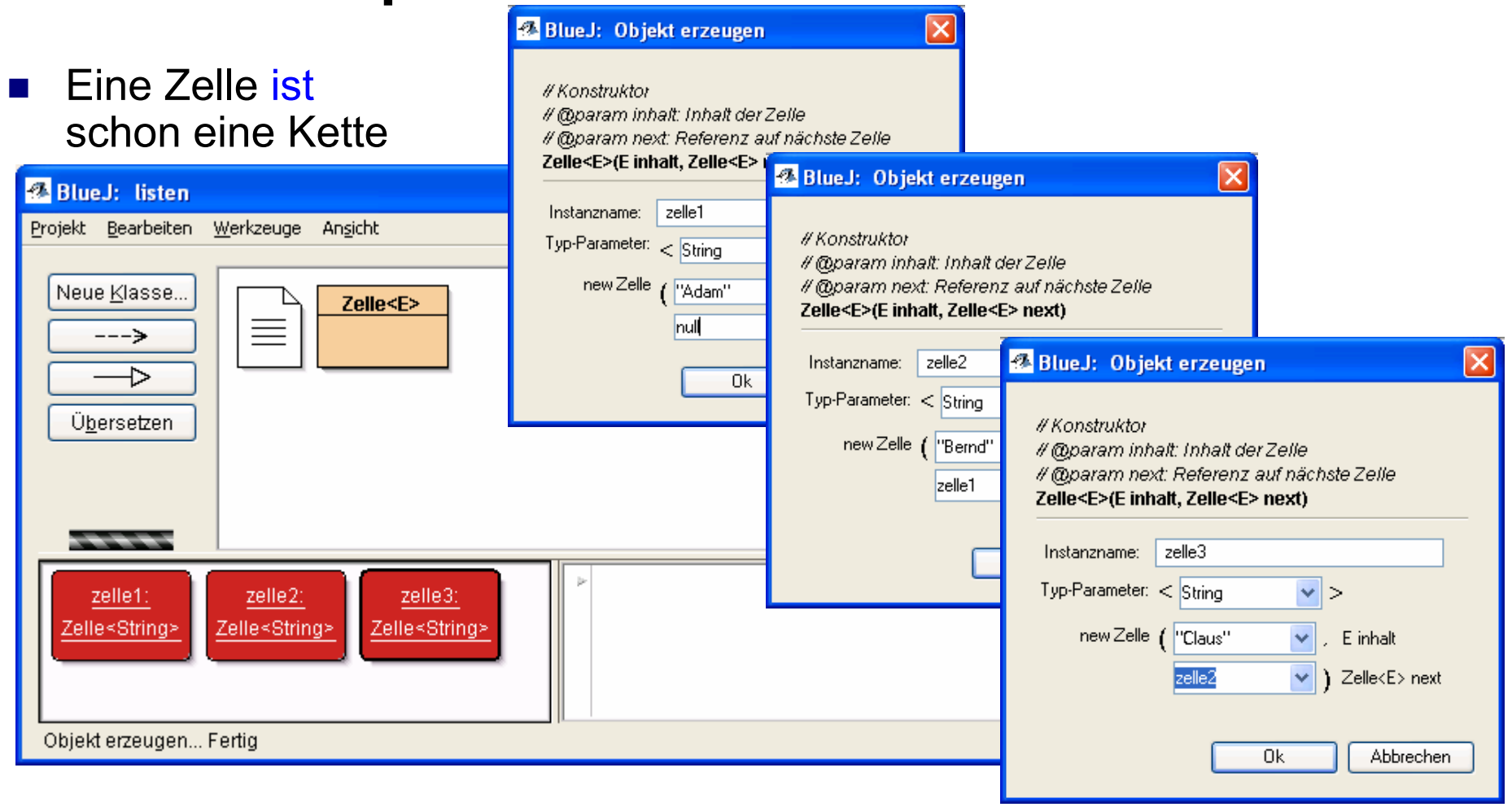
```
public class Zelle<E>{

   E inhalt;          // Inhalt der Zelle
   Zelle<E> next; // Verweis auf nächste Zelle

   /** Konstruktor
    *  @param inhalt: Inhalt der Zelle
    *  @param next: Referenz auf nächste Zelle
    */
   Zelle(E inhalt, Zelle<E> next){
      this.inhalt = inhalt;
      this.next = next;
   }
}
```

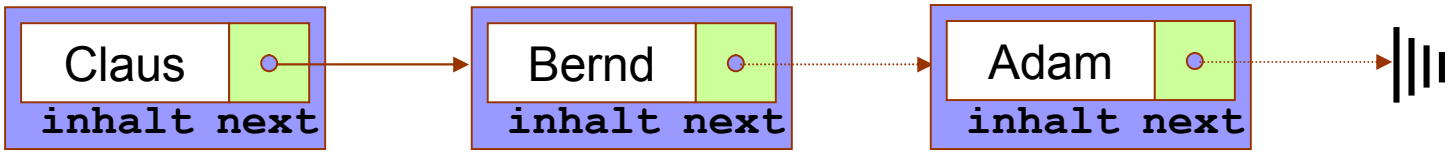
Klasse übersetzt - keine Syntaxfehler | gespeichert



Wir knüpfen Zellen zu Ketten

- Eine Zelle ist schon eine Kette



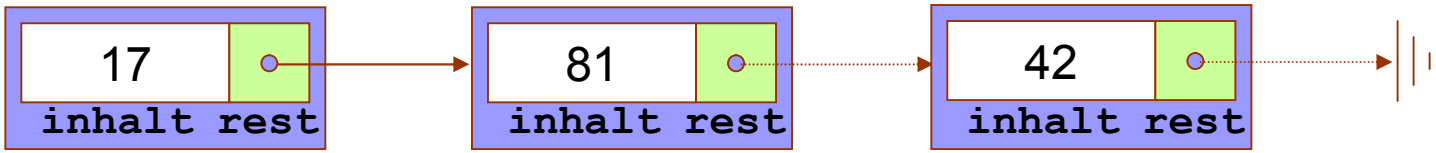


Ist eine Kette schon eine Liste ?

Zur Vereinfachung betrachten wir im Folgenden nur Zellen mit int-Inhalt

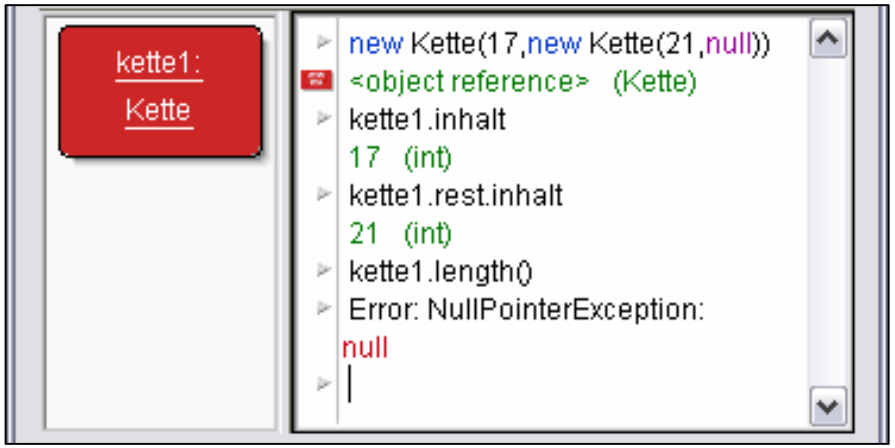
- Manche meinen :
  - eine Zelle sei schon eine Liste
  - Zelle sei nur der falsche Name ein besserer wäre : *Kette*
- Wir können uns eine Kette bauen
  - Beispielsweise die Kette mit den Elementen [17, 81, 42]
  - **`new Kette(17, new Kette(81, new Kette(42,null)))`**

```
public class Kette {
// Objektfelder
int inhalt;  // erstes Element der Kette
Kette rest;     // Rest der Kette

// Konstruktor
 Kette(int element, Kette restKette){
        inhalt = element;
        rest = restKette;
    }
} // Ende der Klasse Kette
```



`null` ist kein Objekt !!

- Entfernt man Elemente einer Liste, so wird sie irgendwann leer.
  - Und wenn man das letzte Element entfernt ?
  - Wie repräsentieren wir die leere Kette ?
    - Haurucklösung: null
    - `Kette leer = null;` ist erst einmal erlaubt
- Aber Vorsicht :
  - `null` ist kein Objekt !!!
  - Anwendung einer Methode führt zu einem Laufzeitfehler:
    - NullPointerException

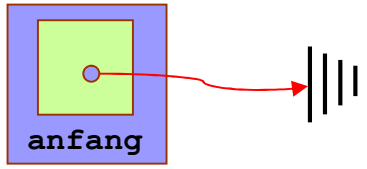



```
public class Kette {
// Objektfelder
int inhalt;  // erstes Element der Kette
Kette rest;    // Rest der Kette

// Konstruktor
 Kette(int element, Kette restKette){
        inhalt = element;
        rest = restKette;
   }

 int length(){ if (this==null) return 0;
                 else return 1+rest.length(); }

} // Ende der Klasse Kette
```

NullPointerException:
null

gespeicher

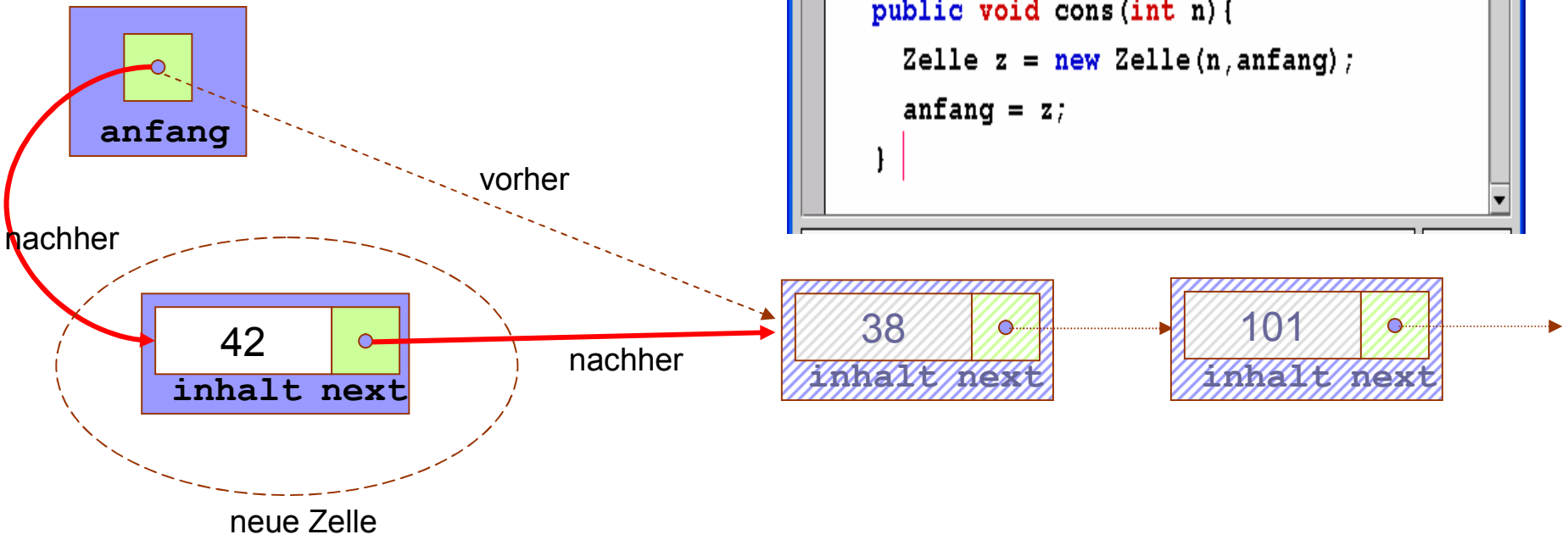
Liste: Verankerte Kette



Die leere Liste

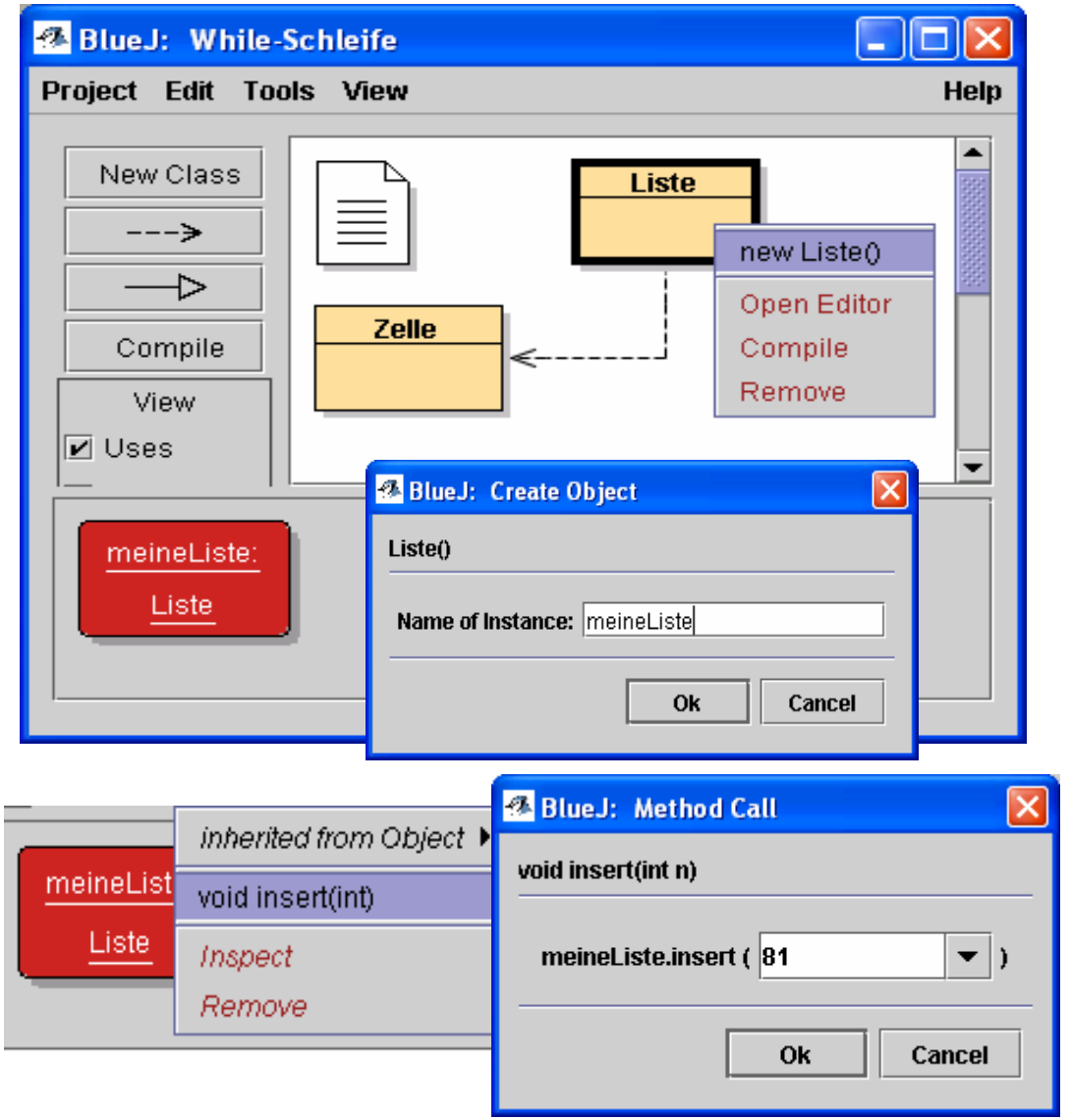
- Ein Liste besteht aus
  - einem Link auf die erste Zelle
  - Methoden um Elemente
    - aufzunehmen
    - zu suchen
    - zu löschen, etc.

```
public class Liste{
  Zelle anfang;

/* Insert at front (Traditionell:"Cons") */
   public void cons(int n){
     Zelle z = new Zelle(n,anfang);
     anfang = z;
   }
```

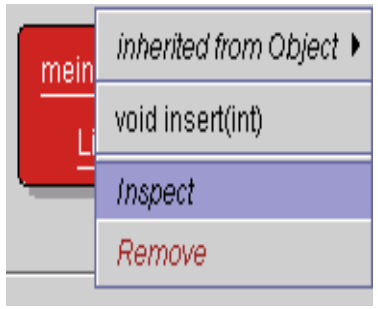


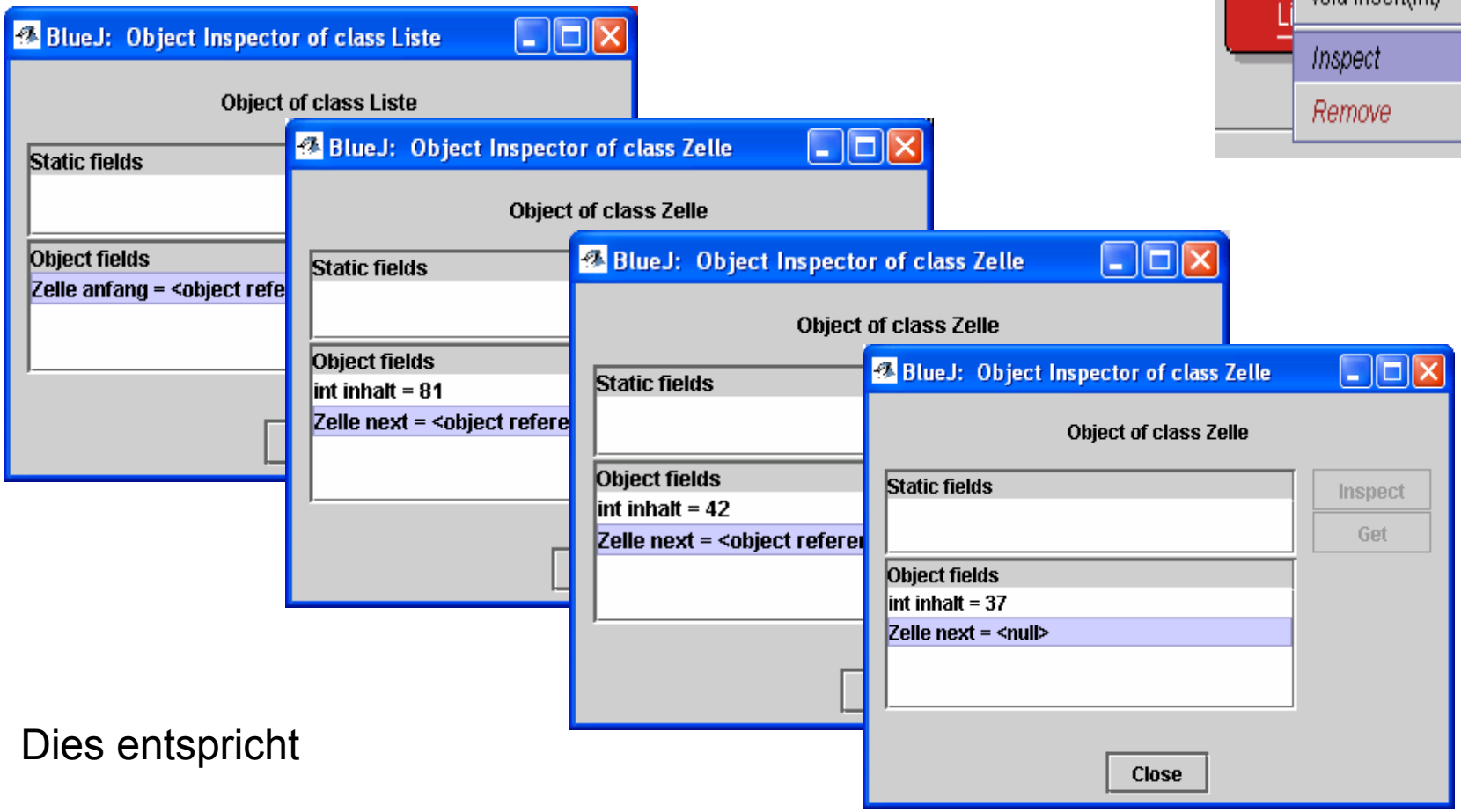
Wir bauen uns eine Liste in BlueJ

- BlueJ zeigt durch einen gestrichelten Pfeil an, dass Liste die Klasse Zelle *benutzt*.
- wir erzeugen eine Liste *meineListe*
- und speichern nacheinander
  - 37
  - 42
  - 81

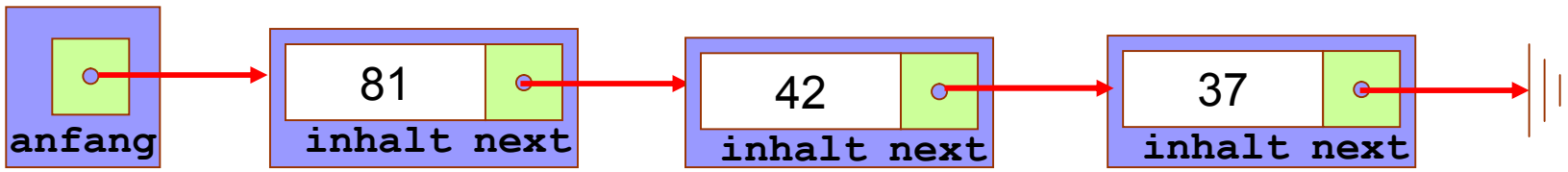


Wir inspizieren die Liste



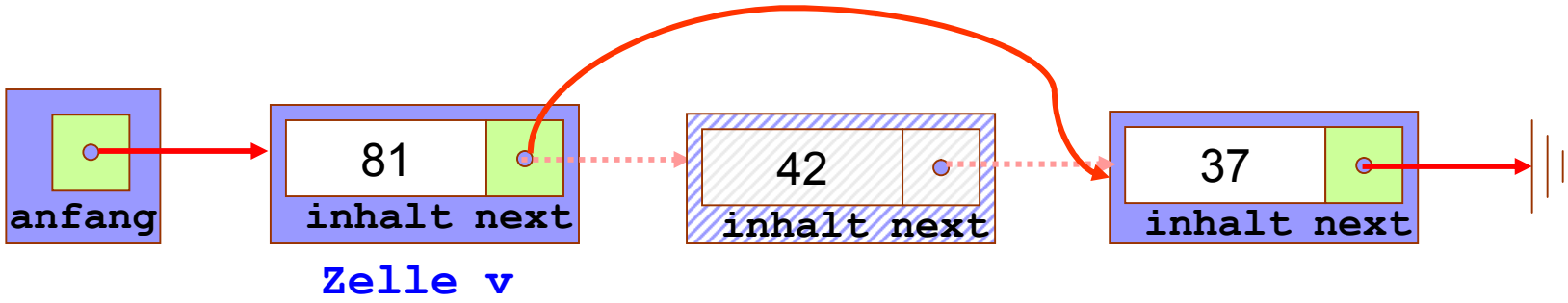


Dies entspricht



Entfernen eines Elementes

- Um ein Element einer Liste zu entfernen, verbinden wir die Vorgängerzelle mit der Nachfolgerzelle

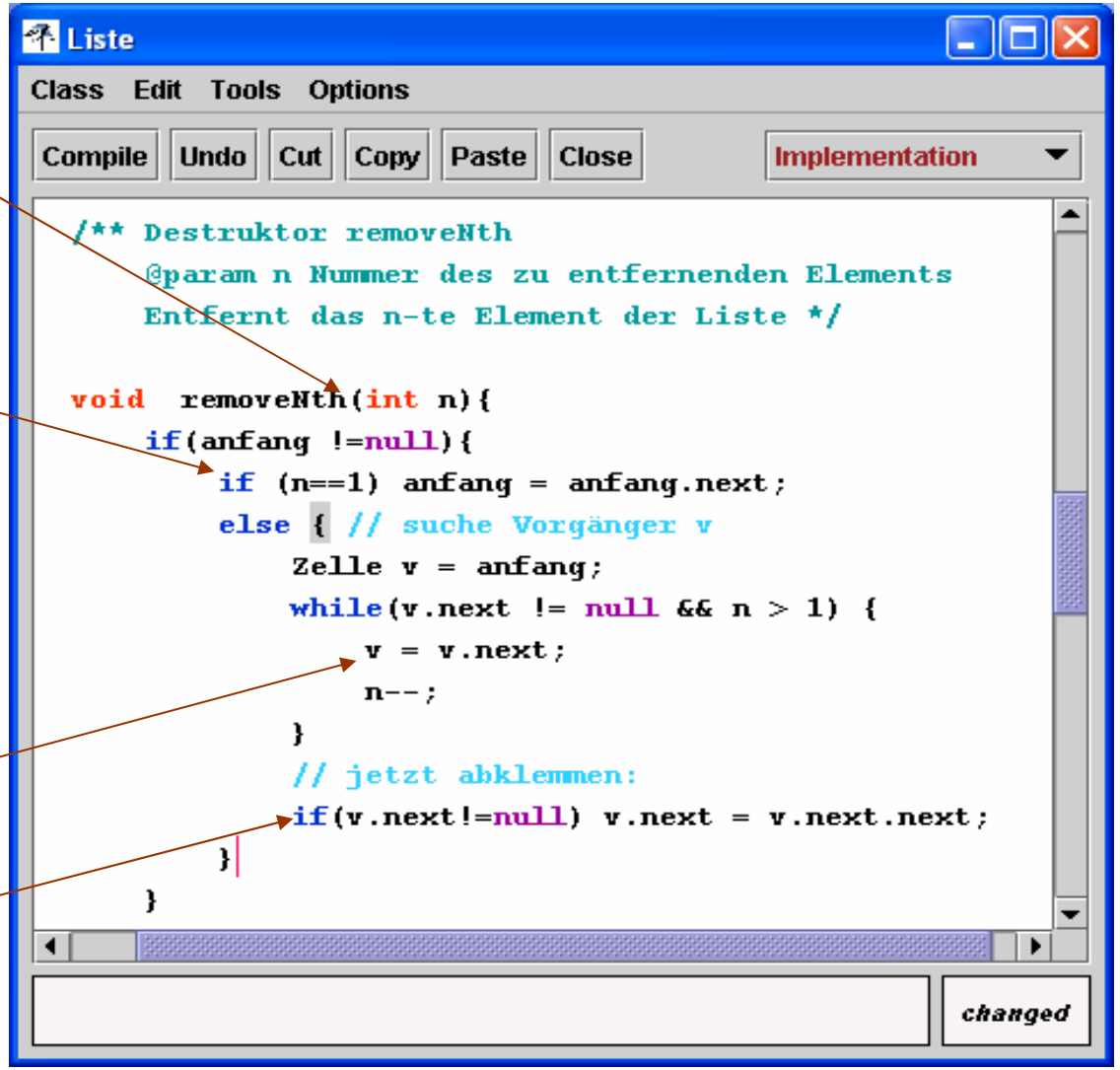


- Das abgeklemmte Element wird irgendwann automatisch von der Java-Speicherbereinigung (garbage collection) gefunden und recycled.

removeNth – entferne n-tes Element

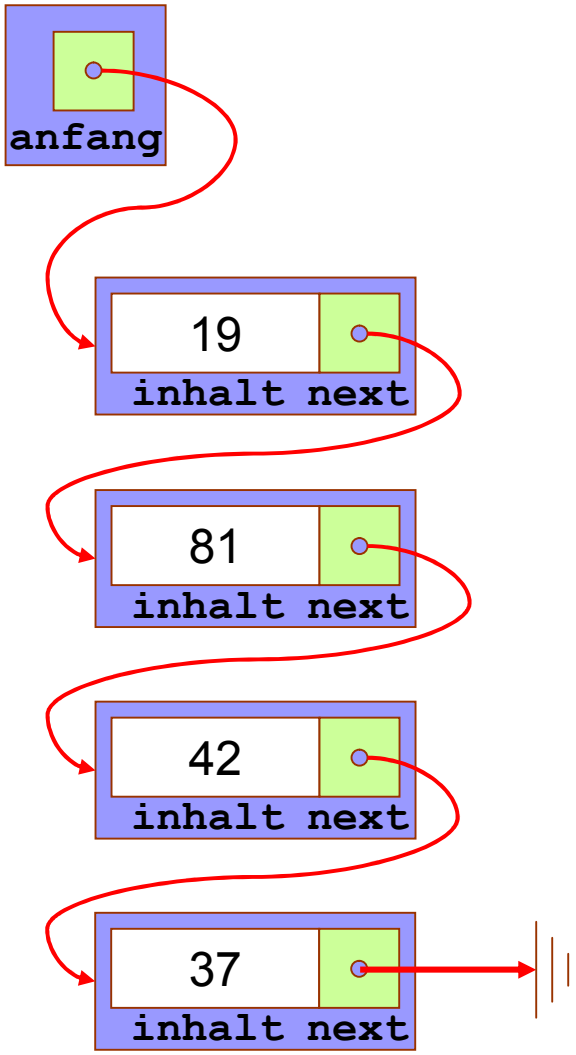
- Von der leeren Liste kann man nichts entfernen
- Das erste Element hat keine Vorgängerzelle, daher müssen wir es gesondert behandeln
- Solange
  - Zähler n > 1, und
  - v einen Nachfolger hat

  vermindere n und rücke v eine Zelle weiter.
- Falls v einen Nachfolger hat, klemme ihn ab.



Verwendung von Listen als Stacks

- Die Liste kann als Stack verwendet werden
  - push(n) ≙ insert(n)
  - pop() ≙ removeNth(1)
  - top() ≙ anfang.inhalt
  - isEmpty() ≙ (anfang==null)
- Vorteil
  - Größe nur durch Hauptspeichergröße begrenzt



Einfache Liste mit Cursor (SimpleList)

- Liste mit Arbeitsposition (cursor)
  - Der Cursor ist null (atHome()) oder zeigt auf ein aktuelles Element
    - dann kann das dort gespeicherte Objekt erfragt werden item()
    - andernfalls tritt eine Exception auf.
  - Der Cursor kann durch die Liste wandern
    - nur von vorne nach hinten : next()
    - nach dem Ende (atEnd()), springt er wieder nach vorne (home())
    - er kann auch sofort an das Ende springen (toEnd()).
  - **Hinter** der Cursorposition können Zellen
    - entfernt: delete() oder
    - eingefügt werden: insert()

SimpleList

«Konstruktor»
+ SimpleList( )

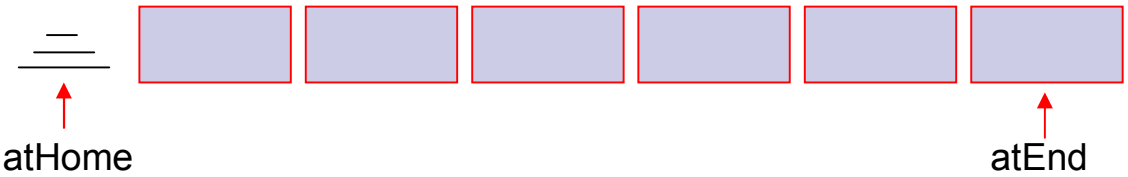
«updates»
+ void insert (*Object*)
+ void delete(Object)

+ void home( )
+ void toEnd( )
+ void next( )

«query»
+ boolean atHome()
+ boolean atEnd()

Navigation

+ Object item( )
+ boolean isEmpty()



Cursorpositionierung

- Wir stellen Operationen zur Cursorpositionierung und –abfrage bereit.
- Cursor zeigt hier immer
  - **vor die Liste, oder**
  - **auf ein aktuelles Element**.
- next()
  - bewegt den Cursor auf das folgende Element
  - und beginnt wieder von vorne

SimpleList

Class Edit Tools Options

Compile | Undo | Cut | Copy | Paste | Find... | Find Next | Close | Implementat

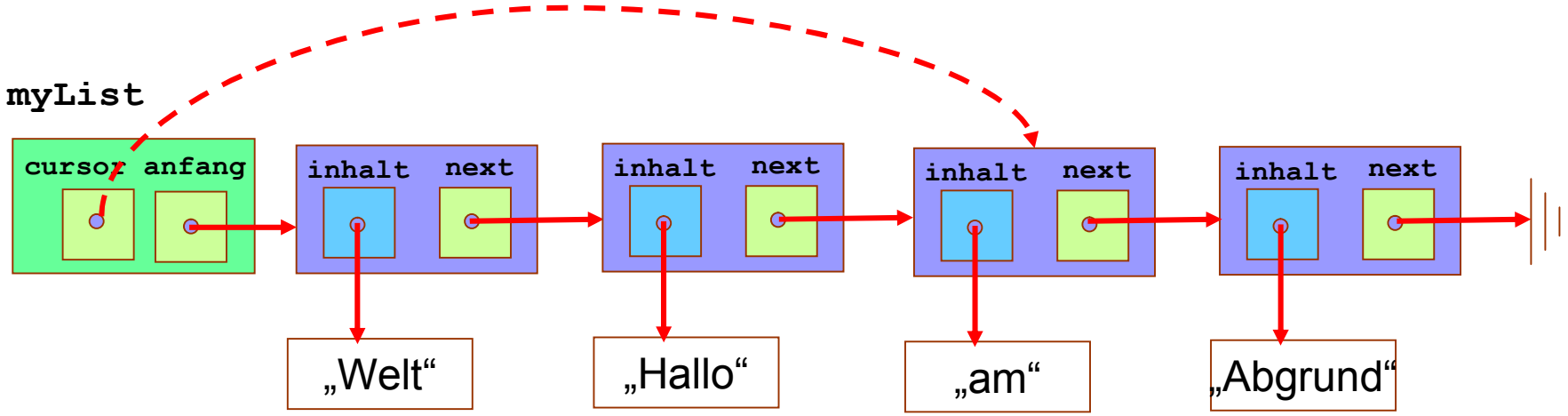
```
public class SimpleList{
    private Zelle anfang;
    private Zelle cursor;

/* Cursorpositionierungen */
    public boolean atHome(){ return cursor==null; }
    public boolean atEnd(){
        return (cursor != null && cursor.next==null);}
    public void home(){ cursor = null;}
    public void next(){
        cursor = atHome() ? anfang : cursor.next;    }

/* Listenoperatoren */
    public boolean isEmpty(){ return anfang == null;}

/* Einfügen nach Cursorposition */
```

changed

Speichern und navigieren

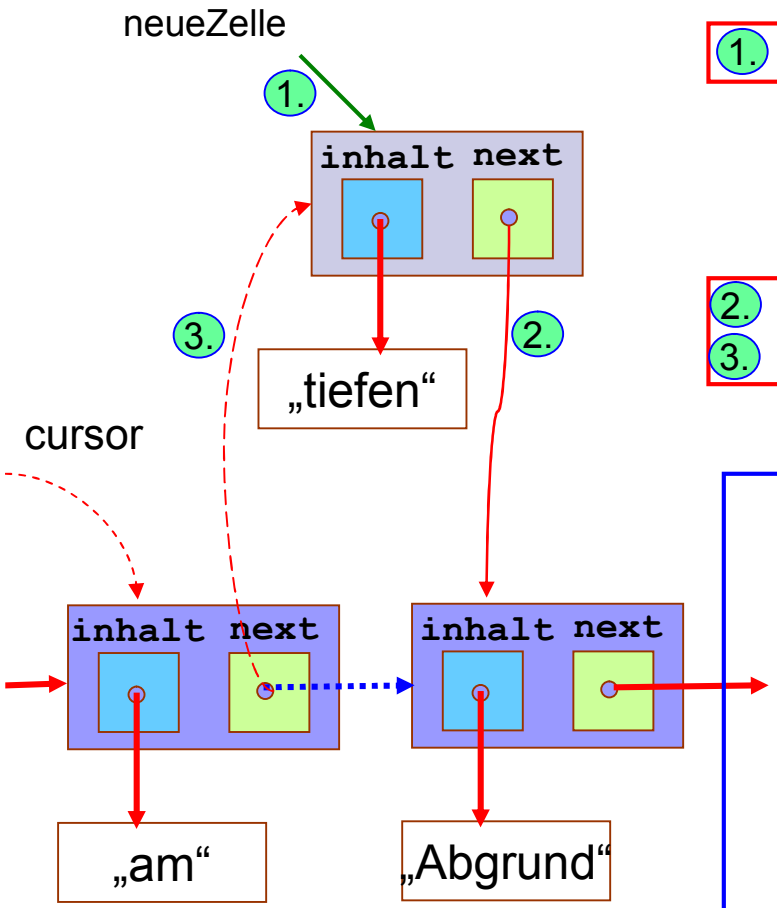
Aufbau der einer Liste nach den Operationen

- SimpleList myList = new SimpleList();
- myList.insert(„Hallo“);
- myList.insert („Welt“);
- toEnd();
- insert („am“);
- next(); insert(„Abgrund“)



SimpleList – einfügen, entfernen

- Man muss unterscheiden
  - am Anfang der Liste
  - ínmitten der Liste




```
/* Einfügen nach Cursorposition */
    public void insert(Object o){
1.      Zelle neueZelle = new Zelle(o);
        if (atHome()){ // cursor vor der Liste
              neueZelle.next = anfang;
              anfang = neueZelle;
        }else{          // cursor in der Liste
2.            neueZelle.next=cursor.next;
3.            cursor.next=neueZelle;
        }
    }
/* Entfernen nach Cursorposition */
    public void delete(){
      if ( isEmpty()|| atEnd() ) return;
      if ( atHome() ) {
              anfang = anfang.next;
      }else{
              cursor.next=cursor.next.next;
      }
    }
```

Elemente suchen und löschen

```
/* Inhalt der gegenwärtigen Zelle */
public Object item(){ return cursor.inhalt;}

/* Schaue eine Zelle nach vorne */
public Object peek(){
    if (cursor.next == null) return null;
    else return cursor.next.inhalt;
}
/* Suche Zelle mit bestimmtem Inhalt */
public Zelle suche(Object o){
   home(); next();
   while( ! atHome() && !o.equals(item()))
     next();
   return cursor;
 }
/* Suche eine Zelle und entferne sie ggf. */
public void entferne(Object o){
   home();
   while( ! atEnd() && !o.equals(peek()))
     next();
   if (! atEnd()) delete();
 }
```

Um ein Element zu suchen

- laufe durch die Liste
- bis Element gefunden ist
- oder am Ende der Liste angelangt

Um ein Element zu suchen und zu löschen

- bleibe vor der Zelle stehen
- lösche das folgende Element

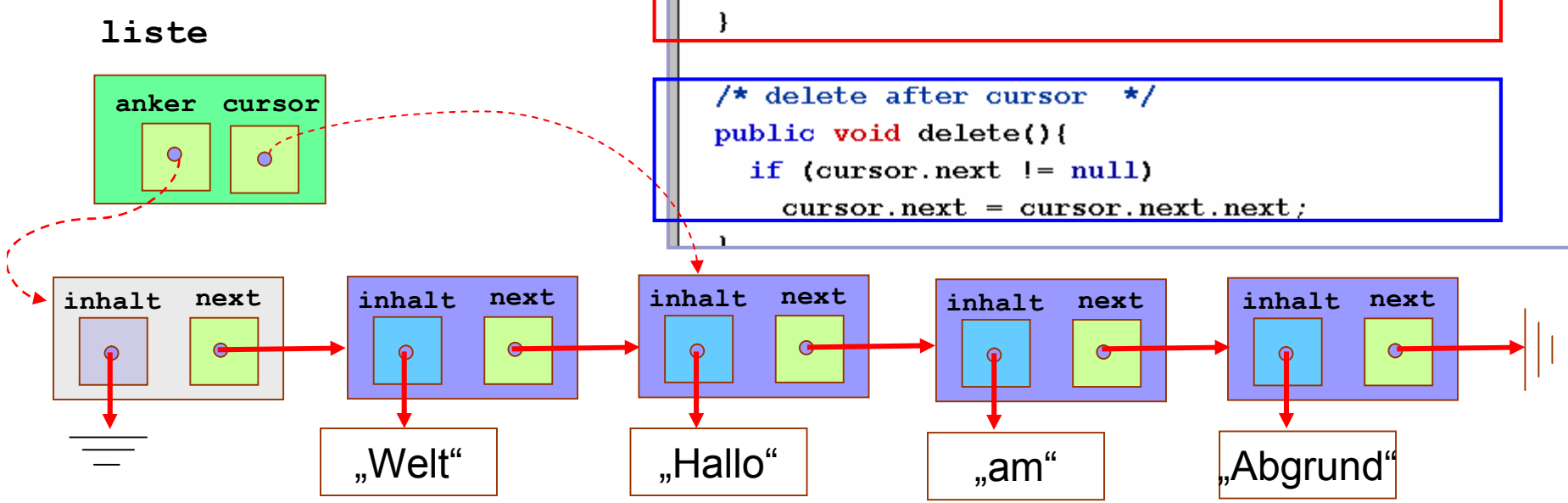
Man muss ein Element nach vorne schauen

- peek()

Implementierung mit Anker-Zelle

- Einfügen und Entfernen einer Zelle
  - am Anfang der Liste
  - mitten in der Liste
- unterscheiden sich.
- Das kann man durch Voranstellen einer Anker-Zelle vermeiden
  - erstes reguläre Listenelement ist Nachfolger der Anker-Zelle
- Code für Einfügen und Entfernen vereinfacht sich deutlich

```
/** Single linked list mit Cursor und Anker-Zelle */
public class SCDList{

Zelle anker;
Zelle cursor;

public SCDList(){
    cursor = anker = new Zelle(null,null){};
}

/* Einfügen hinter Cursor. */
public void insert(Object o){
  cursor.next = new Zelle(o,cursor.next);
}

/* delete after cursor  */
public void delete(){
  if (cursor.next != null)
    cursor.next = cursor.next.next;
}
```



Implementierung mit Iterator

- Liste mit Anker
- implements *Iterable* daher *forEach* möglich
- *Cursor* implementiert *Iterator*;
  - *hasNext()*
  - *next()*
  - *remove()*
- und zusätzliche Methoden
  - *insert()*
  - *locate()*

```
public class AnkerListe<E> implements Iterable<E>{

   Zelle anker;
   /** Konstruktor */

   AnkerListe(){ anker = new Zelle(null,null);}

   /** Konstruktor mit Argumentliste */
   AnkerListe(E...es){
      this();
      Cursor c = iterator();
      for(E e: es)c.insert(e);
   }

   public Cursor iterator(){
       return new Cursor();
   }

   /** Klasse Cursor  */
   public class Cursor implements Iterator<E>{
```

Der Iterator

- Invariante:
  - *cursor* $\neq$ *null*
- Iterator-Methoden
  - *hasNext()*
  - *next()*
  - *remove()*
- Zusätzlich
  - *boolean locate(E e)*
  - Seiteneffekt:
    ready for *remove()*

```
/** Klasse Cursor - für insert, remove, ... */
public class Cursor implements Iterator<E>{

  Zelle cursor;

  // KlassenInvariante:
  private boolean inv(){ return cursor != null; }

  /** Konstruktor */
  Cursor(){ cursor = anker; }

// Das Iterator-Protokoll:
  public boolean hasNext(){
      return cursor.next!=null;
  }

  public E next(){
  /* Pre  */ assert inv();
    cursor=cursor.next;
  /* Post */ assert inv();
    return cursor.inhalt;
  }

  public void remove(){
    if(hasNext())
       cursor.next = cursor.next.next;
   /* Post */ assert inv();
  }
```

Iterator-Protokoll

- ohne foreach-Syntax

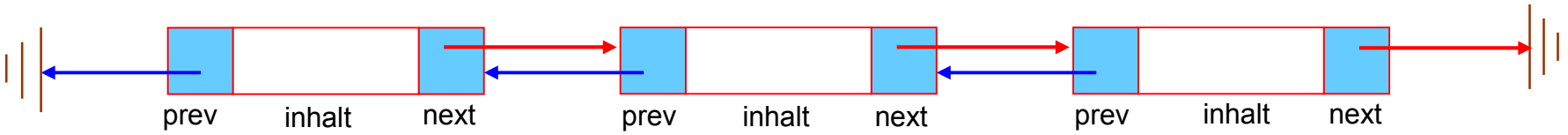
```
public void zeige(){
    Iterator<E> c = iterator();
    while(c.hasNext()){
        E e = c.next();
        System.out.print(e+" ");
    }
}
```

- mit foreach Syntax

```
public void show(){
    for(E e : this)
        System.out.print(e+" ");
}
```

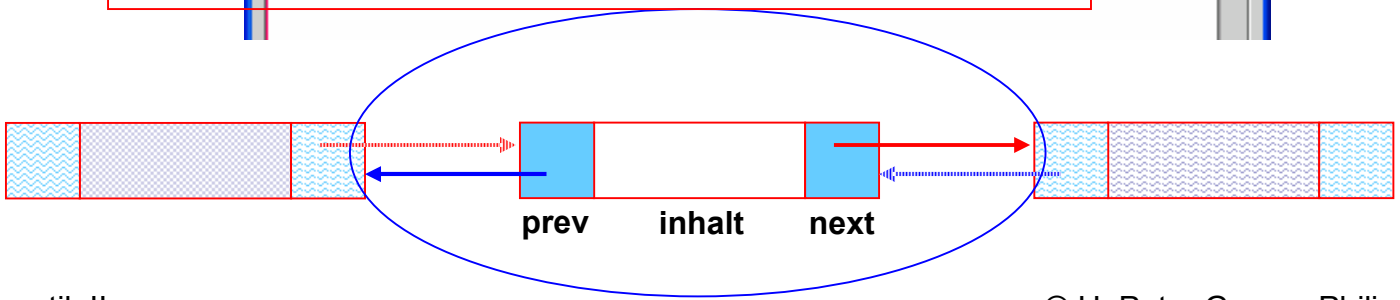
Doppelt verkettete Listen

- Verkettete Listen kann man nur in einer Richtung durchlaufen
- Um den Vorgänger einer Zelle cell zu finden muss man
  - zum Anfang springen
  - durch die Liste laufen, bis
  - cursor.next=cell
- Aufwand um den Vorgänger zu finden : O(n)
  - n Anzahl der Zellen
- Wenn es angebracht ist, oft vor- und zurück zu navigieren
  - baue zusätzlich Referenzen zum Vorgänger ein
  - Aufwand um Vorgänger zu finden : O(1)
  - Nachteil:
    - zusätzliche Referenzen



Doppelzelle als innere Klasse

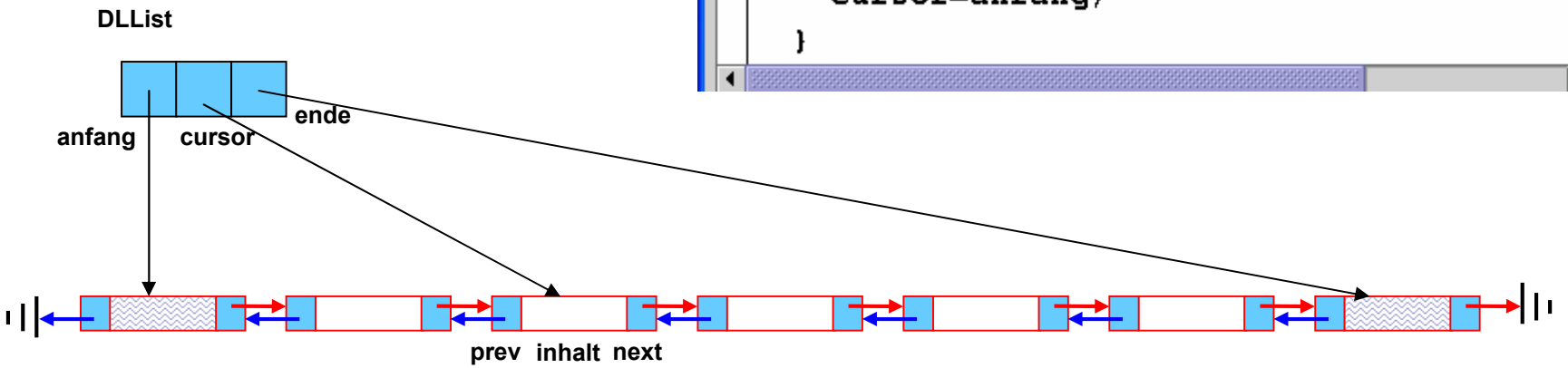
```
/** Doppelt verLinkte Liste mit Cursor*/
public class DLList{

  /** Innere Klasse DCell */
  private class DCell{
    DCell prev;
    Object inhalt;
    DCell next;

    /** Konstruktor für DCell*/
    DCell(DCell p,Object i,DCell n){
      prev=p; inhalt=i; next=n;
    }

    /** Inhalt einer DCell */
    Object inhalt(){return inhalt;}
  }// Ende der inneren Klasse DCell
```



Anker-Zellen *anfang* und *ende*

- Wir setzen Anker-Zellen
  - an den Anfang
  - an das Ende
- Damit entfällt die Betrachtung von Spezialfällen
  - beim Einfügen
  - beim Löschen

```
/** Doppelt verLinkte Liste */
public class DLList{
  DCell anfang;
  DCell cursor;
  DCell ende;

 /* Mit dummy-Zellen "anfang","ende"*/
  DLList(){
    anfang = new DCell(null,null,null);
    ende   = new DCell(anfang,null,null);
    anfang.next = ende;
    cursor=anfang;
  }
```

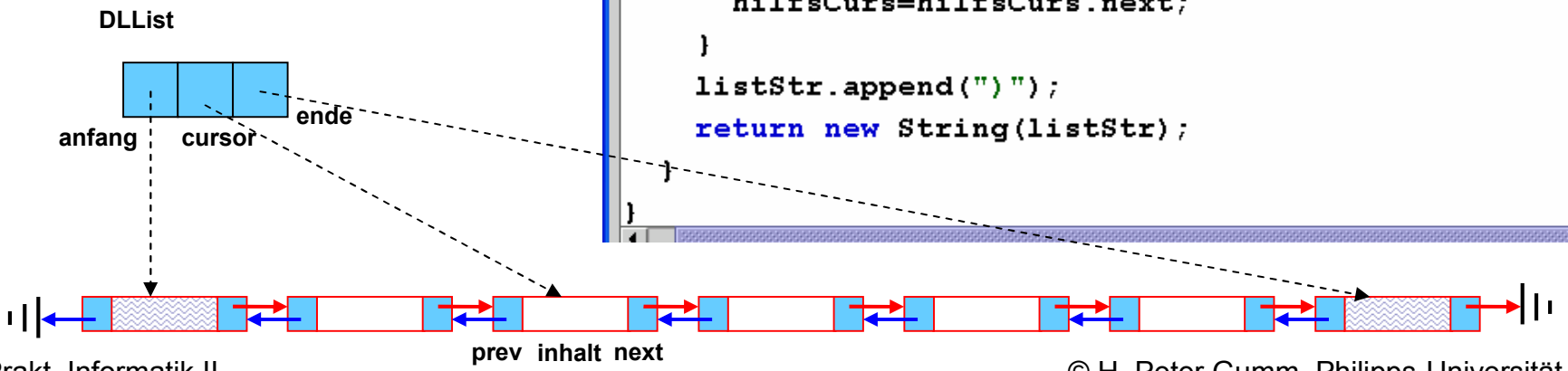


Cursornavigation und Ausgabe

- Bei doppelt verketteten Listen sind
  - anfang und ende
  - next und prev

  gleichberechtigt
- Wir verwenden einen Hilfscursor, damit
  - toString den richtigen Cursor nicht verändert, denn ...
  - Testroutinen, die toString() benutzen, sollen den Zustand des Systems nicht beeinflussen !!!

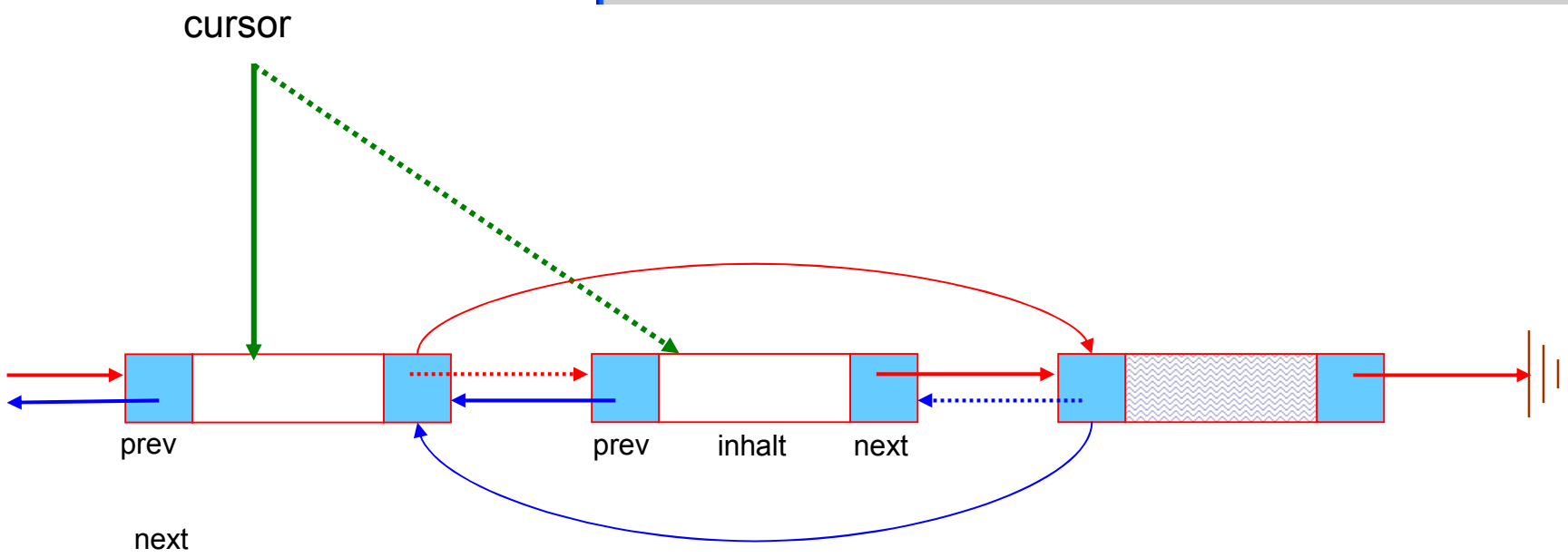
```
/* Cursornavigation */
 public void anfang(){ cursor = anfang;}
 public void next(){
     if(cursor == ende)cursor = anfang;
     else cursor = cursor.next;
 }
 public void prev(){
     if(cursor==anfang) cursor = ende;
     else cursor = cursor.prev;
 }
 /* Nützlich zum Testen */
 public String toString(){
   DCell hilfsCurs = anfang.next;
   StringBuffer listStr = new StringBuffer("( ");
   while (hilfsCurs != ende){
     listStr.append(hilfsCurs.inhalt().toString()+" ");
     hilfsCurs=hilfsCurs.next;
   }
   listStr.append(")");
   return new String(listStr);
 }
}
```



Löschen

- Vorgänger und Nachfolger sind direkt zugreifbar
- Löschen ist ganz einfach, nur ...
  - anfang und ende-Zellen bitte nicht löschen !

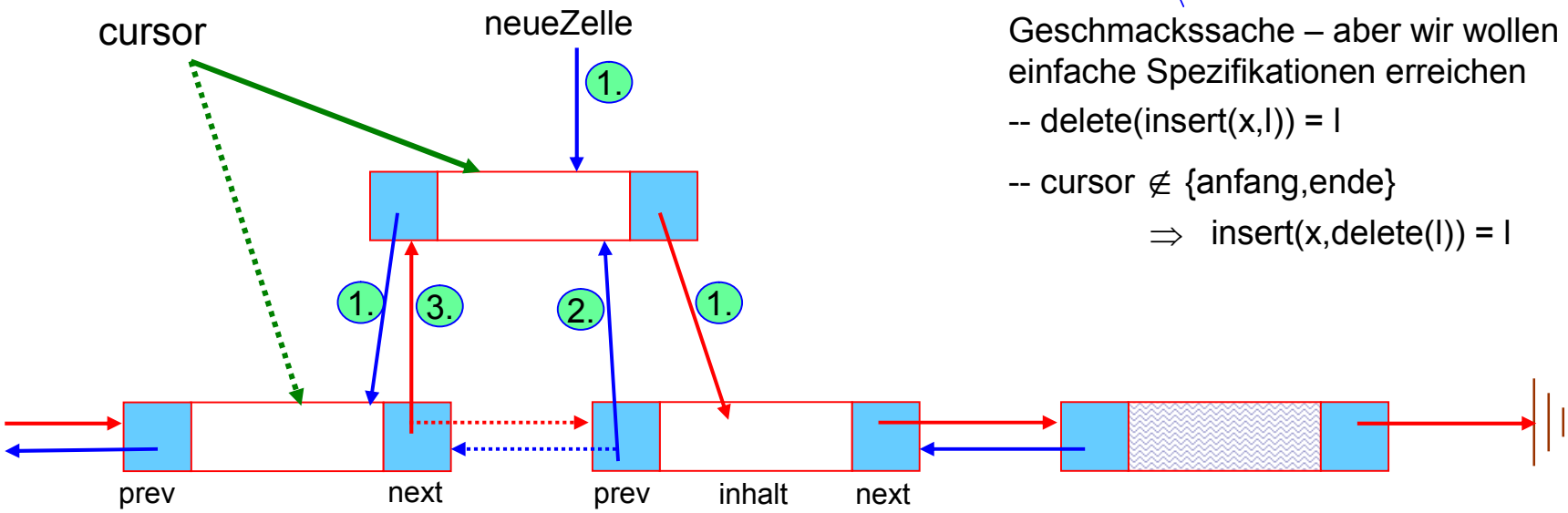
```
/* Entferne Zelle, auf die der Cursor zeigt*/
  public void delete(){
      if (cursor != anfang && cursor != ende){
      cursor.prev.next = cursor.next;
      cursor.next.prev = cursor.prev;
      cursor=cursor.prev;
      }
}
```



Einfügen

- Insert ähnlich wie für einfach verkettete Listen
- Reihenfolge des „*Zeigerverbiegens*“ wichtig

```
/* Füge Objekt rechts vom Cursor ein */
public void insert(Object wert){
    DCell neueZelle =
        new DCell(cursor,wert,cursor.next);
1.  cursor.next.prev = neueZelle;
2.  cursor.next = neueZelle;
3.  cursor = cursor.next;
}
```



Geschmackssache – aber wir wollen einfache Spezifikationen erreichen

-- delete(insert(x,l)) = l

-- cursor $\notin$ {anfang,ende}

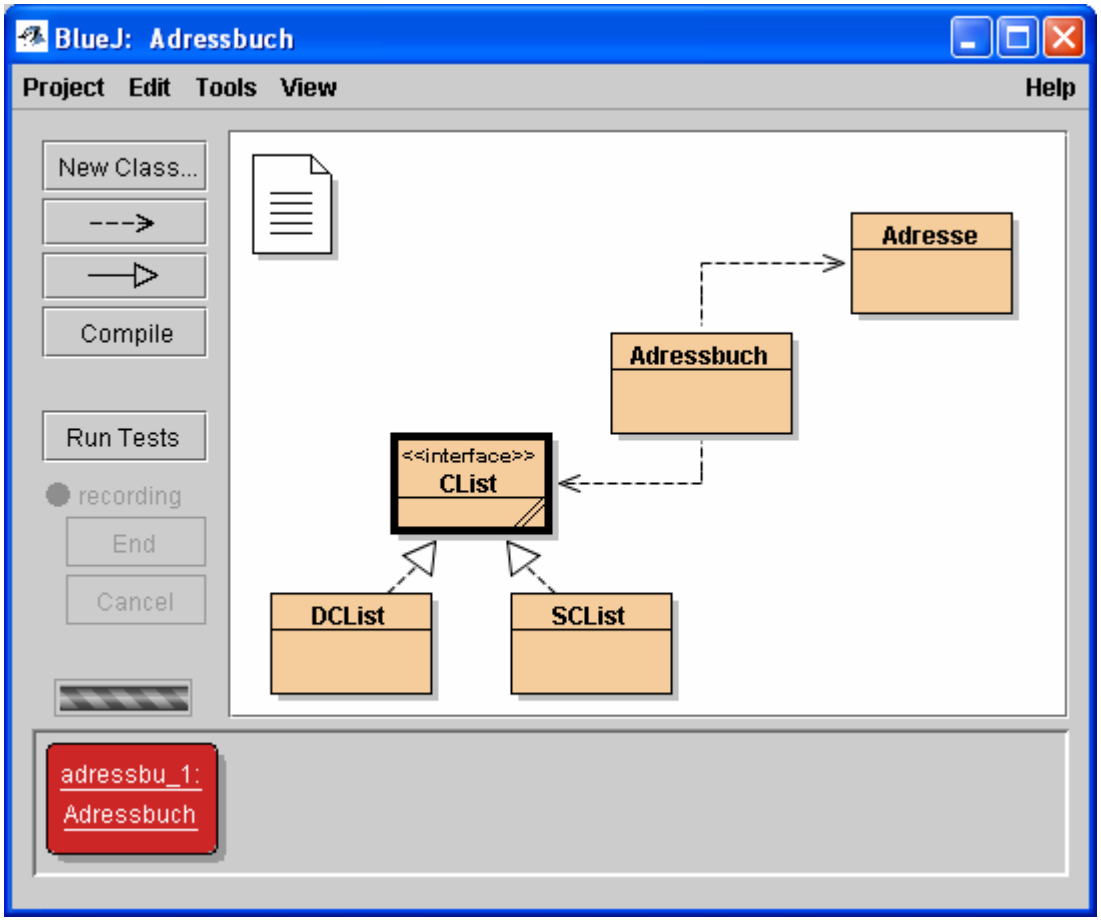
$\Rightarrow$ insert(x,delete(l)) = l

Eine Java-Anwendung

- Wir wollen ein Adressbuch mit einem GUI implementieren.
  - Der Benutzer soll Adressen speichern, löschen und nachschlagen können
  - Außerdem soll er durch die Adressen blättern können
- Was steckt „hinter“ dem GUI ?
  - Eine Klasse „Adresse“
  - Eine Liste mit Cursor
  - Ein javax.swing Fenster
- Was soll das GUI darstellen
  - Das Element der Adressliste, auf dem der Cursor steht, wird in den Textfeldern angezeigt
  - ==> und <== stehen rufen next() und prev()
  - Löschen: löscht die aktuelle Adresse
- Clear leert die TextFelder und ermöglicht Eingabe
  - Mit Einfügen wird dieser gespeichert: insert()
  - Man kann nur einige Felder eingeben und mit Suche eine Adresse finden, die damit übereinstimmt

Die Bestandteile

- Wir wollen verschiedene Arten von Listen mit Cursor ausprobieren
- Wir schreiben zunächst ein Interface
  - CList : Interface für Listen mit Cursor
- Dann implementieren wir das Interface
  - einmal mit einer einfach verketten Liste : SCList
  - mit einer doppelt verketteten Liste : DCList
- Die Klasse Adressbuch enthält das GUI



Das Interface

- Interface spezifiziert mindestens die benötigten Operationen
  - Hier auch prev(), um vor-und zurück zu blättern
- Es soll sowohl mit einfach als auch mit einer doppelt verketteten Liste implementiert werden
- Kommentare
  - hier: Javadoc

wichtig

```
public interface CList{

  /** Test, ob die Liste leer ist */
  public boolean isEmpty();

  /** Test, ob ein Element schon vorhanden */
  public boolean contains(Object o);

  /** Finde eine Zelle. Falls sie gefunden ist,
   * Seiteneffekt: Positioniere den Cursor dort */
  public boolean locate (Object o);

  /** Inhalt der aktuellen Zelle; */
  public Object item();

  /** Zum nächsten Element, sofern existiert */
  public void next();

  /** Zum vorherigen Element */
  public void prev();

  /** Fügt Object nach Cursor ein
   *  post: cursor steht auf dem neuen Element */
  public void insert(Object o);

  /** Entfernt Element, auf dem der Cursor steht */
  public void delete();
```

Das Graphical User Interface

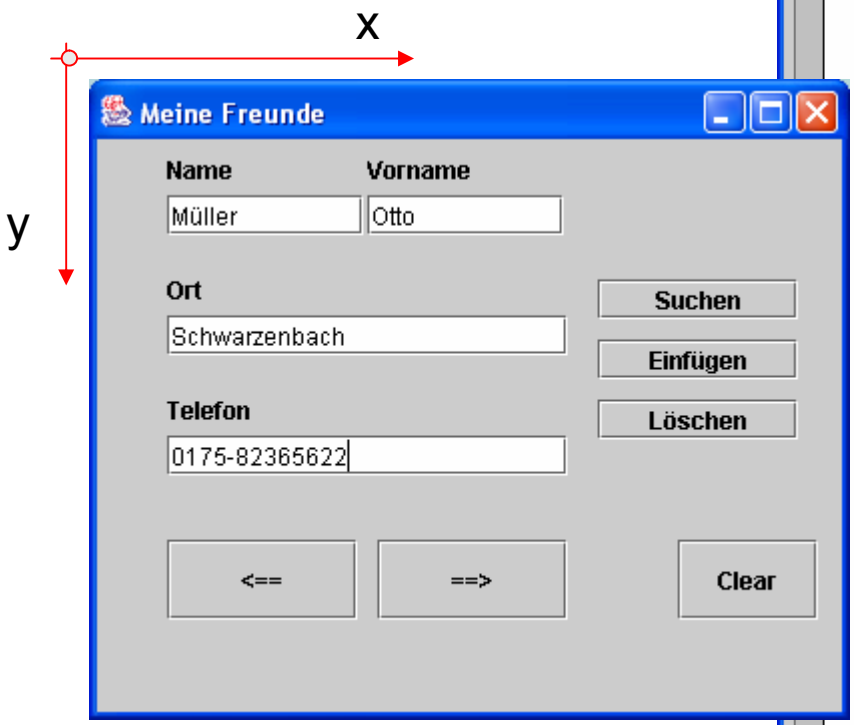
- Bestandteile
  - Beschriftungen (labels)
  - Formularfelder (text fields)
  - Schaltflächen (buttons)
- Benutzung von Paketen aus
  - javax.swing.*;
    - JLabel
    - JTextField
    - JButton
  - java.awt.event.*;
    - ActionListener – bewacht die Buttons


```
import java.awt.event.*;
import javax.swing.*;
public class Adressbuch implements ActionListener {
// Unsere Anwendung besteht aus :
// einem Fenster
   private JFrame window;
// 4 Textfeldern (Labels)
   private JLabel nameLbl, vornameLbl, ortLbl, telLbl;
```

Nur eine Methode zu implementieren:

actionPerformed(ActionEvent)

Das Fenster




```
/* Der Konstruktor baut die Benutzeroberfläche  */
public Adressbuch() {
  // Fenster erzeugen, Größe setzen
  window = new JFrame("Meine Freunde");
  window.setBounds(10, 10, 380, 320);
  // Wir brauchen keinen Layout-Manager:
  window.getContentPane().setLayout(null);

// Passe Labels, Textfelder und Buttons ins Fenster ein *
  nameLbl  = setLabel(35,5,150,20,"Name");
  nameFeld = setTextField(35,28,98,20);
  vornameLbl  = setLabel(135,5,150,20,"Vorname");
  vornameFeld = setTextField(135,28,98,20);
  ortLbl  = setLabel(35,65,100,20,"Ort");
  ortFeld = setTextField(35,88,200,20);
  telLbl  = setLabel(35,125,100,20,"Telefon");
  telFeld = setTextField(35,148,200,20);

  // Setze die Buttons
  vorBtn     = setButton( 140, 200, 95, 40, "==>");
  zurückBtn  = setButton( 35, 200, 95, 40, "<==");
  clearBtn   = setButton( 290, 200,70,40,"Clear");
  suchBtn    = setButton(250, 70, 100, 20, "Suchen");
  einfügeBtn = setButton(250, 100, 100, 20, "Einfügen");
  löschBtn   = setButton(250, 130, 100, 20, "Löschen");
  // Zeigs mir ...
  window.show();
}// Ende des Konstruktos Adressbuch
```

- Kontrollelemente haben x-y-Position, breite (width) und Höhe (height)
- *setLabel*, *setTextField*, *setButton*: selber geschrieben – siehe nächste Folie

Verankern und bewachen

- Ein neuer Button, Label oder TextFeld wird erzeugt
- Position und Größe werden gesetzt (setBounds)
- Er wird auf die Fensterscheibe (contentPane) geklebt (add)
- setLabel und setTextField sind völlig analog
- Buttons benötigen zusätzlich Bewacher (ActionListener), die im Falle des Knopfdrucks eine Methode aufrufen
- ActionListener ist ein Interface mit einer Methode: actionPerformed
- Der Ursprung des Ereignis e wird bestimmt (e.getSource)
- Dementsprechend wird eine gewünschte Methode aufgerufen
- Hier sind dies:
  - Adresse suchen
  - Adresse in Liste einfügen
  - Adresse anzeigen, auf der der Cursor steht.
  - etc.

```
  private JButton setButton(int x, int y, int w, int h, String s){
    JButton myButton = new JButton(s);
    myButton.setBounds(x,y,w,h);
    myButton.addActionListener(this);
    window.getContentPane().add(myButton);
    return myButton;
  }

  /* Setze einen Wächter der die Knöpfe bewacht: Unser
  * Fensterwächter. Er sagt jedem Knopf, was er zu tun hat */

public void actionPerformed(ActionEvent e){
   if (e.getSource() == suchBtn)
      suchen();
   else if (e.getSource() == einfügeBtn)
      neuAufnahme();
   else if (e.getSource() == vorBtn){
      freundListe.next();
      anzeigen();
   }else if (e.getSource() == zurückBtn){
      freundListe.prev();
      anzeigen();
   }else if (e.getSource() == löschBtn){
      freundListe.delete();
      anzeigen();
   }else if (e.getSource() == clearBtn){
      eingabeLöschen();
   }
      window.repaint();
} // end of actionPerformed
```

Objekt nach Textfeld – und zurück

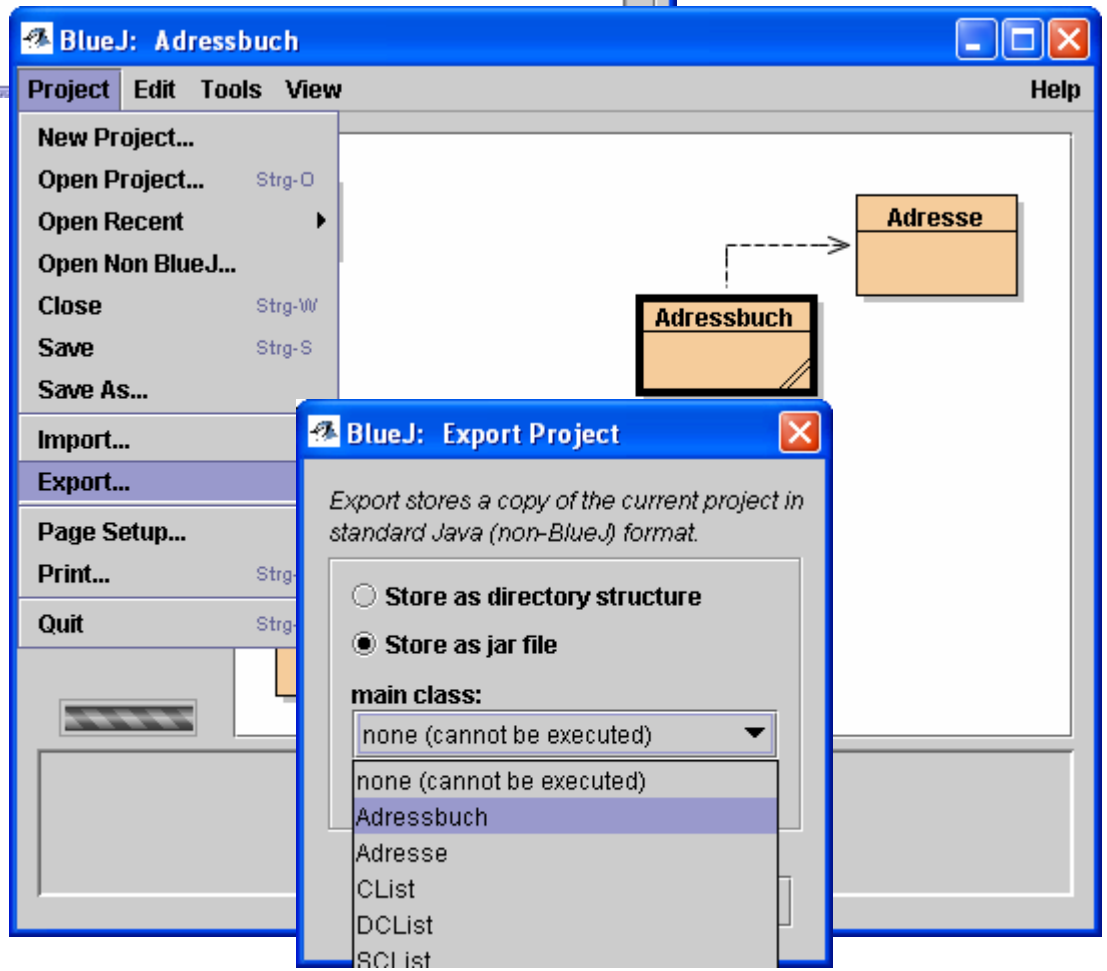
- Inhalte von Textfeldern werden
  - mit getText abgefragt und
  - durch setText gesetzt
- freundListe war eine Liste von Objekten
- um eine Adresse zu entnehmen:
  - Cast nach Adresse notwendig

```
/* Die Routinen, die vom Wächter aufgerufen werden */
  private Adresse eingabe(){
    Adresse adr = new Adresse(
          nameFeld.getText(),
          vornameFeld.getText(),
          "", ortFeld.getText(), "","");
    return adr;
  }

  private void anzeigen(){
    if (freundListe.isEmpty()) eingabeLöschen();
    else{
      Adresse ad = (Adresse)freundListe.item();
      vornameFeld.setText(ad.vorname);
      nameFeld.setText(ad.name);
      ortFeld.setText(ad.ort);
      telFeld.setText(ad.telefon);
    }
  }
```

Ein stand-alone-Programm

```
/* Die main-routine konstruiert ein Adressbuch */
   public static void main(String args[]){
      new Adressbuch();
   }
```

- benötigt main-Methode
- Diese muss nur den Konstruktor aufrufen
- Mit BlueJ kann man eine jar-Datei erzeugen
  - Project->Export
  - Store as jar
  - Hauptklasse (mit main) wählen
- Diese startet mit Doppelklick

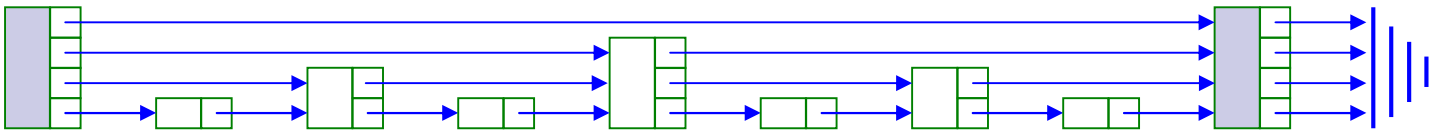


Adaptive Listen

- Elemente am Anfang der Liste findet man am schnellsten
  - Platziere häufig gesuchte Elemente möglichst weit vorn
- Zugriffshäufigkeit
  - anfangs unbekannt
  - kann sich mit der Zeit ändern
  - Liste soll sich adaptieren
- Strategien:
  - MoveToFront
    - jedesmal wenn ein Element gesucht wurde, entferne es und füge es vorne wieder an
  - Beförderung (Transpose)
    - jedesmal wenn ein Element gefunden wurde, vertausche es mit seinem Vorgänger
  - Belohnung
    - Speichere in dem Element die Anzahl der Zugriffe und ordne (gelegentlich) die Liste danach

Geordnete Listen – Skip Listen

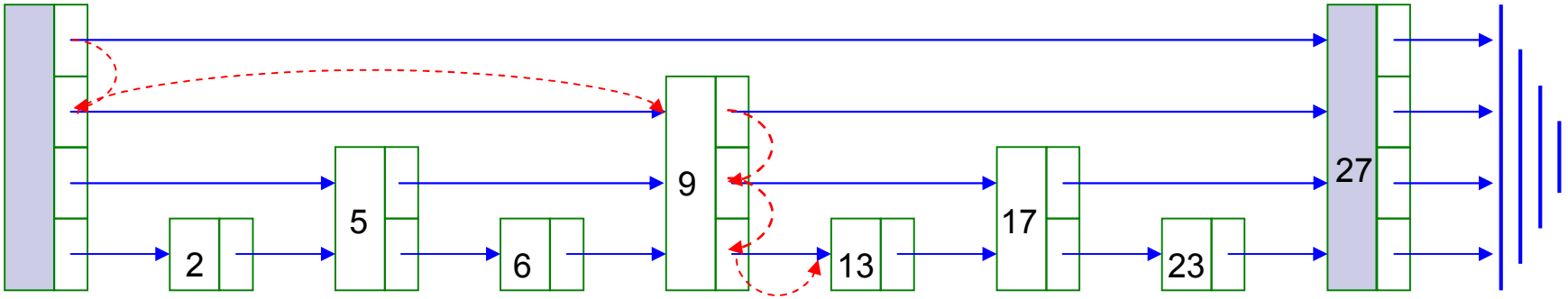
- Daten mit Ordnungsrelation kann man geordnet in Listen speichern
  - Einfügen und Suchen ist jeweils linear
  - Binäre Suche nicht möglich
- Binäre Suche nur möglich mit zusätzlichen Zeigern
  - zur Mitte
    - zur Mitte der ersten Hälfte
    - zur Mitte der zweiten Hälfte
      - zur Mitte des 1. Viertels
      - zur Mitte des 2 Viertels
      - zur Mitte des 3. Viertels
      - zur Mitte des 4. Viertels
        - zur Mitte des 1. Achtels
        - zur Mitte .....
- Verkette Listen mit einer solchen Zeigerstruktur heißen Skip-Listen
  - Zellen am Anfang und in der Mitte bekommen zusätzliche Zeiger
  - Aus Symmetriegründen setzen wir auch eine Dummy-Zelle ans Ende



4-stufige Skip-Liste

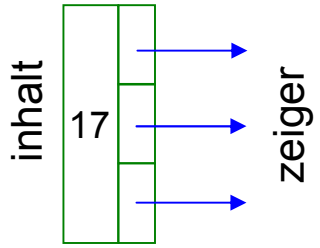
Suche in einer Skip-Liste

- Gesucht: x=13
  - Beginne mit dem obersten Zeiger der ersten Zelle
  - x < 27 $\Rightarrow$ eine Stufe absteigen
  - x $\geq$ 9 $\Rightarrow$ Skip-Zeiger folgen und absteigen
  - x < 17 $\Rightarrow$ eine Stufe absteigen
  - x = 13 Gefunden



4-stufige Skip-Liste

Zellenstruktur



- Eine SkipZelle benötigt einen array von Zeigern

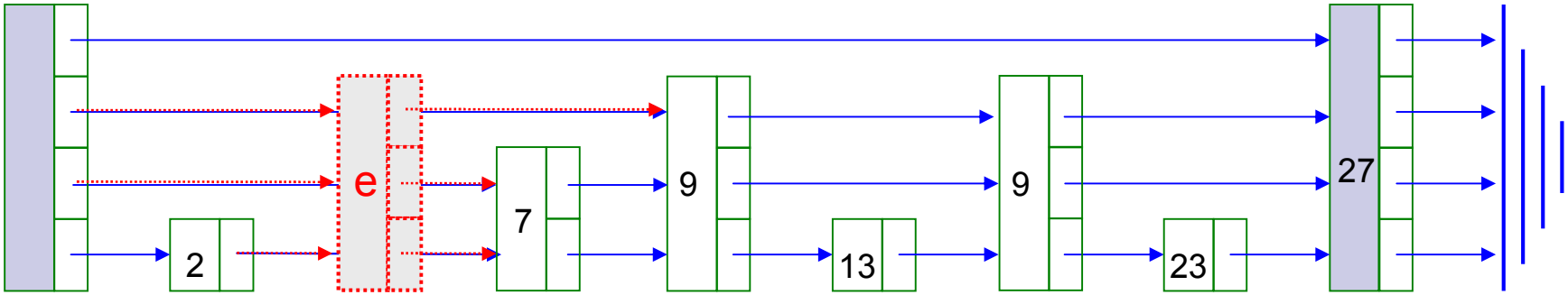
```
public class SkipList{
  int maxStufe;
  SkipZelle anfang;

  protected class SkipZelle{
    Object inhalt;
    SkipZelle[] zeiger;

    /** @param stufen Anzahl der Zeiger
     *  @param inhalt Inhalt der Zelle */

    SkipZelle(Object inhalt, int stufen){
      this.inhalt = inhalt;
      zeiger = new SkipZelle[stufen];
    }
  }
```

Randomisierte Skip-Liste

- Aufteilung nicht unbedingt genau in der Mitte
  - Suche funktioniert wie vorhin
  - evtl. nicht mehr ganz so effizient

- Beim Einfügen einer Zelle wird deren Höhe zufällig bestimmt
  - Höhe h mit Wahrscheinlichkeit $1/2^h$
  - $h < max$
  - max ungefähr log(n) wobei n die Anzahl der erwarteten Daten

- Einfügen eines Elementes e , hier z.B.: e=5
  - Zelle der zufälligen Höhe h konstruieren
  - Zeiger zur neuen Zelle setzen:
    - „so tun als ob“ man das neue Element suchen würde
    - wenn Höhe ≤ h ist und man bei der Suche absteigen müsste, setze Zeiger zur neuen Zelle
  - setze Zeiger von der neuen Zelle auf jeweils die nächste Zelle der Höhe $i \leq h$



Mutatoren, Operationen

- Für Methoden zur Manipulation von Daten gibt es zwei Möglichkeiten

  1. Die Methode verändert das Objekt
     - Mutator
     - Mutatoren sind Kennzeichen des imperativen Programmierens
     - void Methoden sind immer Mutatoren
     - Mutatoren sind manchmal effizienter

  2. Die Methode lässt das Objekt unverändert. Resultat ist neues Objekt
     - Solche Methoden heißen *Operationen*
     - Operationen sind Kennzeichen des *deklarativen Programmierens*
     - Mit Operationen kann man leichter umgehen
     - Operationen sind weniger fehleranfällig

- `insert` ist ein *Mutator*
- `removeNth` ist ein Mutator

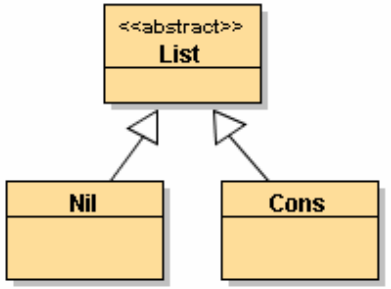
- Eine `insert` entsprechende *Operation* müsste
  - eine neue Liste konstruieren und
  - die alte unversehrt lassen

- Eine `removeNth` entsprechender *Operation* müsste
  - eine neue Liste konstruieren
  - die alte unversehrt lassen

- Wie soll das gehen ?
- Das soll einfacher sein ?

Listen als Ergebnistypen

- Objektorientiert (bekannt)
  - abstract class Liste
    - definiert abstrakt:
      boolean isEmpty, Liste insert(E e), ...
  - class Null extends Liste
    - implementiert z.B.
      boolean isEmpty(){ return true; }
  - class Cons extends Liste
    - implementiert z.B.
      boolean isEmpty(){ return false; }



- Funktional
  - nur Ergebnistypen
  - keine void-Methoden
  - leere Liste (null) kein Problem

Liste statisch und als Ergebnistyp

- Grundmenge:
  - E* = Endliche Folgen von Elementen einer Menge E
- Konstruktoren:
  - nil : *List*
  - cons : $E \times ConsList \rightarrow ConsList$
- Prädikat:
  - isEmpty : $ConsList \rightarrow boolean$
- Selektoren:
  - head : $ConsList \rightarrow E$
  - tail : $ConsList \rightarrow ConsList$
    - *jeweils nur definiert, falls not isEmpty*
- Gleichungen:
  - head(cons(e,l)) = e
  - tail(cons(e,l)) = l

ConsList

«Konstruktoren»
nil( )
cons (E,List)

«Predicate»
boolean isEmpty (List)

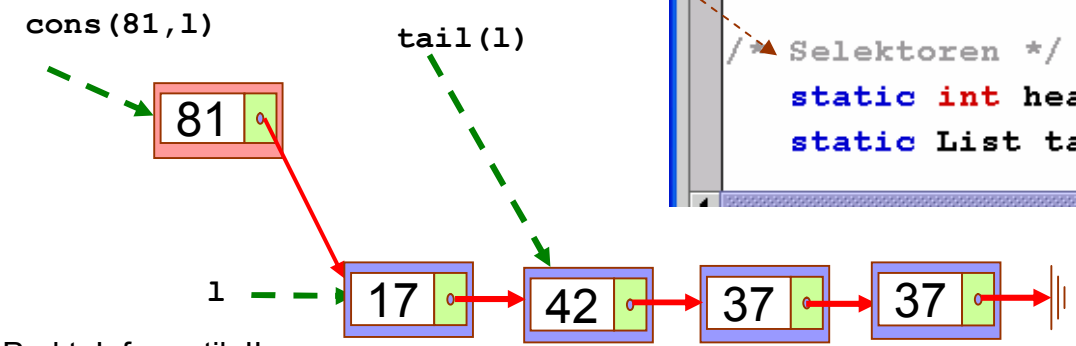
«Selektoren»
E head(List)
List tail(List)

Wir implementieren die Listenoperatoren als Klassenmethoden (static)

Liste als Ergebnistyp - in Java

- Rekursiver Datentyp
- Liefert leere Liste
- Liefert neue Liste mit Kopf h und Rest l
- Prädikat
- Selektoren
  - Laufzeitfehler bei leerer Liste

```
public class List{
    int head;
    List tail;

    List(int wert, List l){
        head = wert;
        tail = l;
    }
/* Die leere Liste */
    static List nil(){ return null;}

/* Die Liste mit head h, tail l */
    static List cons(int h, List l){
        return new List(h,l);
    }
/* Prädikat */
    static boolean isEmpty(List l){ return l==null;}

/* Selektoren */
    static int head(List l){ return l.head;}
    static List tail(List l){ return l.tail;}
```



Induktive Definition - Kochrezept

- Listen sind induktiv definiert
  - Eine Liste l ist leer
  - `isEmpty(l)`

  oder l besteht Kopf und Restliste

  - `l = cons(head(l),tail(l))`

- Methoden werden rekursiv definiert
  - Basisfall: leere Liste
    - Meist trivial
  - Rekursiver Fall:
    - Zur Verfügung stehen:
    - `head(l)`
    - `Ergebnis der Methode auf tail(l)`

```
/* Länge einer Liste */
  static int length(List l){
      if (isEmpty(l))
         return 0;
      else
         return 1+length(tail(l));
  }
```

Zusätzliche nützliche Operatoren

```
/* Konkatenieren zweier Listen */
  static List append (List l1, List l2){
    if ( isEmpty(l1))
      return l2;
    else return cons(head(l1),append(tail(l1),l2));
  }
/* String-Darstellung */
  public static String toString(List l){
    if (isEmpty(l)) return " ";
    else return head(l)+" "+toString(tail(l));
  }

/* enthält l das Element e ?? */
  static boolean enthält(List l, int e){
    if (isEmpty(l)) return false;
    else if(head(l)==e) return true;
    else return enthält(tail(l),e);
  }
```

- Strategie
  - löse das Problem für die leere Liste
  - beschreibe wie aus
    - head(l)
    - der Lösung für tail(l)
  - die Lösung für l gewonnen wird

Gemischte Implementierungen

- Feste Menge (z.B. Array) von Eimern (buckets),
  - Jeder Eimer ist ein Behälter
  - z.B. Liste oder Menge, ...

- Vorteil
  - suchen:
    - berechne den Eimer, in dem das gesuchte Element sein muss
    - durchsuche den Eimer
  - einfügen:
    - berechne den Eimer in den das Element gehört
    - wirf es hinein
- Berechnung des Eimers nennt man Hashing

Map Interface

- Eine Map ordnet
  - einem Argument (key)
  - einen Wert (value) zu
- Eine Map entsteht durch
  - einfügen (insert)
  - löschen(remove)

  eines (key,value)-Paares
- Selektoren:
  - isEmpty
  - containsKey
  - lookup
- Meist will man contains-Key mit lookup verbinden:
  - Vereinbare „Sentinel“-Wert (z.B. null):
    containsKey(k) ←→ lookup(k) == null

Map Interface

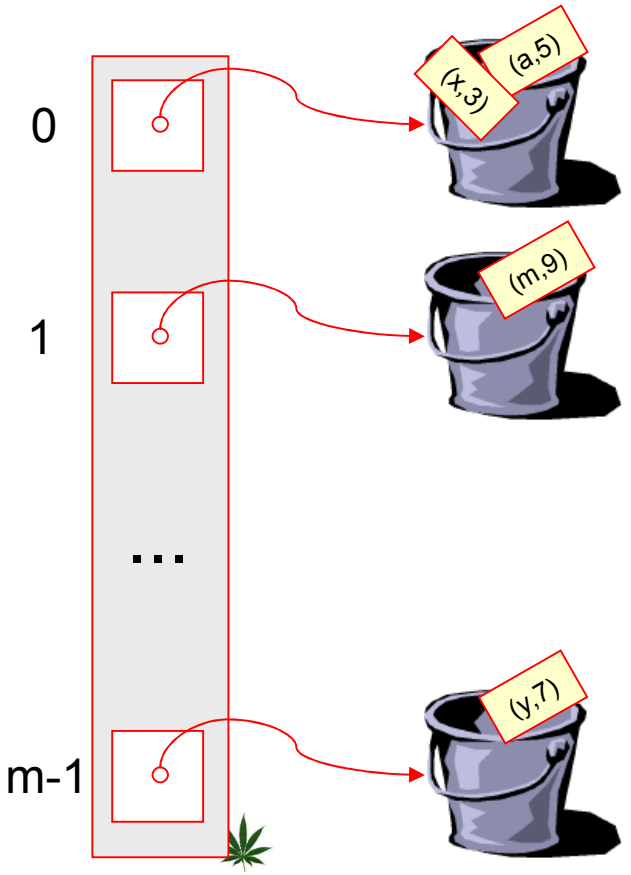
«updates»
+ insert (*Object, Object*)
+ void remove(Object)

«query»
+ boolean isEmpty()
+ boolean containsKey(Object)
+ Object lookup(Object)

Hash-Tabelle

```
Bucket[] buckets = new Bucket[4]
```

- Verteilung der key-value-Paare auf m Eimer (buckets)
  - Jeder bucket kann als Liste implementiert werden
- Hash-Funktion h berechnet für jeden Schlüssel k einen Bucket h(k)
  - dort wird (k,v) gespeichert
  - dort wird (k,v) gesucht
  - Ziel: gleichmäßige Verteilung der key-value-Paare auf die m buckets
- Ziel
  - h: Keys $\rightarrow$ {0,…, m-1}
  - h surjektiv
  - h effizient zu berechnen
  - $\forall i < m : |h^{-1}(i)| \approx |Keys| / m$
- Gewinn: Beschleunigung der Suche um Faktor m



Hash-Funktionen

- Ziel: Gleichmäßige Verteilung beliebiger Schlüssel auf m Eimer
  - Wir nehmen an, dass jeder Schlüsselwert k durch eine Zahl n(k) charakterisiert ist
    - z.B.falls k ein String: n(k) = Summe der ASCII Werte der Zeichen von k
- Divisionsmethode
  - h(k) = n(k) mod m
  - Vorsicht: Eine Systematik in den Daten kann zu einer Überbelegung in einem bestimmten Eimer führen
  - Beispiel: 10 Eimer, fast alle n(k) enden mit 0
    - $\Rightarrow$ Ein Bucket wird voll, alle anderen bleiben fast leer.
- Mittel-Quadrat-Methode
  - Bilde $n(k)^2 = (d_r\, d_{r-1} \ldots d_0)_{10}$
    - Beispiel: m = n(k) = 128, $n(k)^2 = 16384$
  - Wähle einen Block von mittleren Ziffern
    - Beispiel: 1**638**4
  - Reduziere modulo(m)
- Vorteil: Die mittleren Ziffern hängen von allen Ziffern in n(k) ab.
  - Dadurch werden die Hash-Werte besser gestreut

Hashing in Java

- Jedes Java-Objekt hat eine Methode *hashCode()*
- Wichtigste Eigenschaft:
  - o_1.equals(o_2)
    $\Rightarrow$ o_1.hashCode() == o_2.hashCode()
  - Sowohl equals als auch hashCode kann und sollte man in eigenen Klassen überschreiben.
  - Auf jeden Fall soll die obige Eigenschaft erfüllt bleiben
  - Wünschenswert ist noch, dass verschiedene Objekte meist auch verschiedenen hashCode haben
    - möglichst wenige Kollisionen
- Nutzen
  - Bei der Suche kann man erst auf gleichen hashCode testen
  - Sind die hashCodes verschieden, ist das Element nicht gefunden
- In *java.util* gibt es Behälter, die hashing verwenden
  - *HashMap*
  - *HashSet*
  - *Hashtable*
  - *LinkedHashMap*
  - *LinkedHashSet*