TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Lernziele

- Verschiedene Listenarten zu implementieren (einfach-/doppelt-verkettet, Array-Listen).
- Konzept Information Hiding verstehen und in Java anwenden können
- Zugriffsmodifikatoren `private` und `public` verwenden
- getter-/setter-Methoden sinnvoll einsetzen
- Grenzen des Information Hiding in Java erläutern können

Datenstruktur Liste

Datenstruktur Liste

- Im Code-Tagebuch haben wir eine Liste zur Verwaltung der Einträge verwendet.
- Typische Operationen auf Listen:
  - Elemente hinzufügen und entfernen
  - über die Elemente iteriert
  - Anzahl der Elemente ermitteln
- Aber wie ist diese Datenstruktur eigentlich implementiert?

Was ist eine Liste?

Definition

Eine **Liste über einem Typ** T ist eine total geordnete Multimenge mit Elementen aus T (bzw. eine Folge, d.h. eine Abbildung $\mathbb{N} \rightarrow T$).
Eine Liste heißt **endlich**, wenn sie nur endlich viele Elemente enthält.

Drei Implementierungsvarianten in dieser Vorlesung:

1. als einfachverkettete Liste
2. als doppeltverkettete Liste
3. als Array-Liste

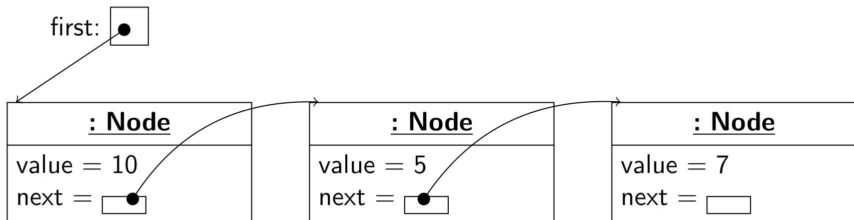
TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Einfachverkettete Listen

Bei einfachverketteten Listen wird für jedes Listenelement ein Knoten-Objekt mit zwei Attributen angelegt:

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

Die Liste mit den Elementen `[10,5,7]` erhält also folgende Repräsentation:



Die Knoten-Klassen

```
class Node {
    int value;
    Node next;

    Node(int value) {
        this.value = value;
        this.next = null;
    }
}
```

Mit dieser Implementierung von Node lässt sich die Liste [10,5,7] wie folgt erstellen:

```
Node n1 = new Node(10);
Node first = n1;
Node n2 = new Node(5);
n1.next = n2;
Node n3 = new Node(7);
n2.next = n3;
```

Begriffsklärung: Objektgeflecht

Eine Menge von Objekten, die sich gegenseitig referenzieren, nennen wir ein **Objektgeflecht**.

Bemerkungen:

- Objektgeflechte werden zur Laufzeit aufgebaut und verändert, sind also dynamische Entitäten.
- Klassendiagramme kann man als vereinfachte statische Approximationen von Objektgeflechten verstehen.

Die IntList-Klasse I

- Abstraktere Schnittstelle wünschenswert!

```
class IntList {
  Node first;

  IntList() {
    first = null;
  }

  void add(int element) { ... }
  int size()            { ... }
  int get(int index)    { ... }
}
```

Die IntList-Klasse II

Diese Liste kann dann wie folgt verwendet werden:

```
// Neue leere Liste erstellen:
IntList list = new IntList();
// Elemente einfuegen:
list.add(10);
list.add(5);
list.add(3);
list.add(7);
// Element an Index 2 ausgeben:
System.out.println(list.get(2));
// Groesse der Liste ausgeben:
System.out.println(list.size());
// Alle Zahlen in der Liste ausgeben:
Node n = list.first;
while (n != null) {
  System.out.println(n.value);
  n = n.next;
}
```

Hinzufügen von neuen Einträgen am Ende der Liste

```
void add(int element) {
  Node newNode = new Node(element);
  if (first == null) {
    first = newNode;
  } else {
    Node n = first;
    while (n.next != null) {
      n = n.next;
    }
    n.next = newNode;
  }
}
```

- Erstelle zunächst einen neuen Knoten (ohne Nachfolger!).
- Fallunterscheidung
  - List bisher leer: Füge das Element als erstes Element ein.
  - Sonst: Iteriere zum Ende der Liste und füge das Element dort an.

Länge der Liste

```
int size() {
  int res = 0;
  Node n = first;
  while (n != null) {
    res = res + 1;
    n = n.next;
  }
  return res;
}
```

Element an Position

```
/* Liefert das Element an Position index
   requires 0 <= index < Anzahl der Eintraege
 */
int get(int index) {
  Node n = first;
  for (int i = 0; i < index; i++) {
    n = n.next;
  }
  return n.value;
}
```

- Die Indizes sind, analog zu Arrays, ab 0 nummeriert.
- Zunächst iterieren wir über die Listeneinträge bis zum gewünschten Knoten. Danach wird der Eintrag, der in diesem Knoten abgelegt ist, zurückgeliefert.

Frage

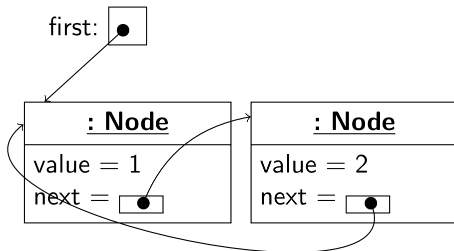
Was passiert bei der Ausführung des folgenden Codes?

```
IntList list = new IntList();
list.add(1);
list.add(2);
list.first.next.next = list.first;

int i = list.size();
```

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Zyklus in der Liste



Terminierung und Invarianten

- Klasse `IntList` hat also bestimmte Anforderungen an den Zustand der Liste, die immer gelten sollten.
- Solche Anforderungen nennt man auch **Invarianten**.

Wie können wir sicherstellen, dass die Implementierung nur so verwendet wird, wie gedacht,
d.h. dass die Invarianten nicht verletzt werden?

Kapselung und Strukturieren von Klassen

Schnittstellenbildung und Kapselung

Objekte stellen eine bestimmte Funktionalität/Dienste zur Verfügung:

- Aus Anwendersicht: Nachrichten schicken, Ergebnisse empfangen
- Aus Implementierungssicht: Realisierung der Zustände und Funktionalität durch
  - → objektlokale Attribute
  - → Referenzen auf andere Objekte
  - → Implementierung von Methoden

Ziel:

- Lege die Anwendungsschnittstelle genau fest.
- Verhindere Zugriff "von außen" auf Implementierungsteile, die nur für internen Gebrauch bestimmt sind.

Begriffsklärung: Anwendungsschnittstelle

Die **Anwendungsschnittstelle** eines Objekts bzw. eines Referenztyps besteht aus

- den Nachrichten, die es für Anwender zur Verfügung stellt;
- den Attributen, die für den direkten Zugriff von Anwendern bestimmt sind.

Bemerkung:

- Die Festlegung von Anwendungsschnittstellen ist eine Entwurfsentscheidung.
- Direkter Zugriff auf Attribute muss nicht gewährt werden, sondern kann mit Nachrichten/Methoden realisiert werden.

Begriffsklärung: Information Hiding (Geheimnisprinzip)

- Anwendern soll nur die Informationen zur Verfügung stehen, die zur Anwendungsschnittstelle gehören.
- Alle anderen Informationen und Implementierungsdetails sind für sie verborgen und möglichst nicht zugreifbar sind.

Gründe für Information Hiding:

- Vermeiden unsachgemäßer Anwendung
- Vereinfachen von Software durch Reduktion der Abhängigkeiten zwischen ihren Teilen
- Austausch von Implementierungsteilen

Beispiele: Postleitzahlen

```
class Addresse {
  String vorname, nachname;
  String strasse;
  int hausnummer;
  int postleitzahl;
  String stadt;
}
```

Beispiele: Postleitzahlen

```
class Addresse {
  String vorname, nachname;
  String strasse;
  int hausnummer;
  int postleitzahl;
  String stadt;
}
```

- Klasse `Addresse` erlaubt Zugang zu allen Attributen
- In (West-)Deutschland bis 1962 zweistellige, danach vierstellige, seit 1993 fünfstellige PLZ
- Es ist semantisch nicht sinnvoll mit Postleitzahlen zu rechnen.
- Andere Ländern: bis zu 10 Zeichen, auch Buchstaben und Bindestriche.

Information Hiding in Java

Java ermöglicht es für Programmelemente sogenannte *Zugriffsbereiche* zu deklarieren.

Vereinfachend gesagt kann ein Programmelement nur innerhalb seines Zugriffsbereichs verwendet werden.

Java unterscheidet vier Arten von Zugriffsbereichen:

- nur innerhalb der eigenen Klasse (Modifikator `private`)
- nur innerhalb des eigenen Pakets (ohne Modifikator)
- nur innerhalb des eigenen Pakets und in Subklassen (Modifikator `protected`)
- ohne Zugriffsbeschränkung (Modifikator `public`)

Generell können Klassen, Attribute, Methoden und Konstruktoren mit diesen Zugriffsmodifikatoren deklariert werden.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Beispiel: Vermeiden falscher Anwendung I

```
public class IntList {
  private Node first;

  public IntList() {
    first = null;
  }

  public void add(int element) { ... }
  public int size()            { ... }
  public int get(int index)    { ... }
}
```

Beispiel: Vermeiden falscher Anwendung II

Die Schnittstelle erlaubt keinen direkten Zugriff auf die Knoten der Liste.

- Es gibt keine Methode `getFirst` o.ä., und keine der Methoden liefert eine Referenz auf ein `Node`-Objekt.
- Außerhalb der Klasse `IntList` kann daher weder das Attribut `first` noch die Listenstruktur verändert werden.
- Damit wird vermieden, dass die Listen beispielsweise Zyklen enthält.
- Es erlaubt ausserdem die Implementierung der `Node`-Klasse beliebig abzuändern, ohne dass Code, der Objekte der Klasse `IntList` verwendet, abgeändert werden muss.

Beispiel: Austausch von Implementierungen I

```java
public class IntList {
    private Node first;
    private int size;

    public IntList() {
      first = null;
      size = 0;
    }

    // Liefert die Anzahl der Eintraege
    public int size() {
      return this.size;
    }

    // Fuegt einen neuen Eintrag hinzu
    public void add(int e) {
      ...
      this.size++;
    }
    ...
}
```

Beispiel: Austausch von Implementierungen II

Die Anwendungsschnittstelle erlaubt keinen direkten Zugriff auf die Länge der Liste.

- Außerhalb der Klasse `IntList` kann nur mittels `size()` die Anzahl der Listeneinträge abgefragt werden.
- Verwender der Klasse können das `size`-Attribut nicht beliebig verändern.
- Wie kann man die Erzeugung von ungültigen Datumswerten verhindern?

Zugriffsfunktionen

- Zugriff auf Attribute häufig über spezifische *getter*- / *setter*- Methoden realisiert.
- Kontrolliert Modifikationen, z.B.
  - das Überprüfen der Gültigkeit von neuen Attributwerten
  - das Benachrichtigen von anderen Objekten bei Änderungen ($\Rightarrow$ späteres Kapitel zu "Beobachtermuster")

Beispiel

```
class MitDirektemAttributZugriff {
   public int attr;
}

class AttributZugriffUeberMethoden {
   private int attr;
   public int getAttr() {
    return attr;
   }
   public void setAttr(int a) {
    attr = a;
   }
}
```

Aufgabe

1. Schreiben Sie die Klasse `Date` so um, dass die Attribute nicht direkt zugreifbar sind.
2. Fügen Sie nun Getter-Methoden für die einzelnen Attribute hinzu.
3. Warum ist es sinnvoll keine beliebigen Veränderungen bei den Datumsobjekten zu erlauben?

```
class Date {
    int day;
    int month;
    int year;

    Date(int day, int month, int year) {
      this.day = day;
      this.month = month;
      this.year = year;
    }
}
```

Lösungsvorschlag I

```
class Date {
    private int day;
    private int month;
    private int year;

    Date(int day, int month, int year) throws Exception {
      if (isValidDate(day,month,year)) {
        this.day = day;
        this.month = month;
        this.year = year;
      } else {
        throw new Exception("Kein gueltiges Datum!");
      }
    }
    public int getDay() {
      return this.day;
    }
    public int getMonth() {
      return this.month;
    }
    public int getYear() {
      return this.year;
    }
```

Lösungsvorschlag II

```
  private boolean isValidDate(int d, int m, int y){
    // Monate mit 31 Tagen
    if (m == 1  || m == 3 || m == 5 || m == 7 || m == 8 ||
        m == 10 || m == 12) {
      return (1 <= d && d <= 31);
    }
    // Monate mit 30 Tagen
    if (m == 4 || m == 6 || m == 9 || m == 11) {
      return (1 <= d && d <= 30);
    }
    // Februar
    if (m == 2) {
      boolean istSchaltjahr = ((y % 4 == 0) && (y % 100 != 0))
                              || (y % 400 == 0);
      return (istSchaltjahr && 1 <= d && d <= 29)
        || (!istSchaltjahr && 1 <= d && d <= 28);
    }
    // In allen anderen Faellen
    return false;
  }
}
```

Hinweis: Diese Implementierung ist nicht effizient und dient nur der Veranschaulichung.

In der Praxis sollten geeignete Java-Bibliotheken verwendet werden.

Beispiel: Web-Seiten I

Objekte zur Repräsentierung trivialer Web-Seiten mit Titelzeile und Inhalt

```
1  public class W3Seite {
2     private String titel;
3     private String inhalt;
4
5     public W3Seite(String t, String i) {
6        titel  = t;
7        inhalt = i;
8     }
9     public String getTitel() {
10     return titel;
11    }
12    public String getInhalt(){
13     return inhalt;
14    }
15 }
16
```

Beispiel: Web-Seiten II

Die obige Klasse kann ersetzt werden durch die folgende, ohne dass Anwender der Klasse davon betroffen werden:

```
1  public class W3Seite {
2    private String seite;
3    public W3Seite( String t, String i ) {
4      seite =  "<TITLE>" + t + "</TITLE>" + i ;
5    }
6    public String getTitel() {
7      int ix = seite.indexOf("</TITLE>") - 7;
8      return new String(seite.toCharArray(), 7, ix);
9    }
10   public String getInhalt() {
11     int ix = seite.indexOf("</TITLE>") + 8;
12     return new String(seite.toCharArray(), ix,
13                               seite.length() - ix );
14   }
15 }
16
```

Bemerkung

- Information Hiding erlaubt insbesondere:
  - konsistente Namensänderungen in versteckten Implementierungsteilen
  - Verändern versteckter Implementierungsteile, soweit sie keine Auswirkungen auf die öffentliche Funktionalität haben (kritisch)

    Beispiel:
    Die zweite Implementierung von W3Seite kann nur dann anstelle der ersten benutzt werden, wenn Titel den Substring `</TITLE>` nicht enthalten.

- Attribute sollten privat sein und nur in Ausnahmefällen öffentlich.

Grenzen der Zugriffskontrolle I

```
public class A {
   private int[] werte;

   public void setWerte(int[] ar){
      for (int i = 0; i < ar.length; i++) {
         if (ar[i] < 0) {
            // Repariere Eintrag
            ar[i] = 0;
         }
      }
      this.werte = ar;
   }
}
```

Geben Sie eine Beispielverwendung an, die eine negative Zahl in das `werte`-Array einträgt, ohne die Implementierung der Klasse `A` zu verändern.

Grenzen der Zugriffskontrolle II

```
public static void main (String[] args) {
  A aobj  = new A();
  int[] w = {4,7,1};
  aobj.setWerte(w);

  w[0] = -7;
}
```

Iteratoren

Iteratoren

Iteratoren erlauben es, schrittweise über sogenannte Datenstrukturen für Datenansammlungen (engl. *collections*) wie Listen zu laufen, so dass alle Elemente der Reihe nach besucht werden.

Im Zusammenhang mit Kapselung sind sie unverzichtbar.

```
public class IntListIterator {
    //prueft, ob es noch weitere Eintraege gibt
    public boolean hasNext(){ ... }

    //liefert den naechsten Eintrag
    public int next() {...}
}
```

Beispiel: Iterator für einfachverkettete Liste I

Wir reichern die Klasse `IntList` mit Iteratoren an:

```
public class IntList {
  // Erweiterung um Iteratoren
  private Node first;
  ...
  public IntListIterator iterator() {
     return new IntListIterator(first);
  }
}
```

Beispiel: Iterator für einfachverkettete Liste II

```
1  import java.util.NoSuchElementException;
2
3  public class IntListIterator {
4    private Node current;
5    public IntListIterator(Node e) {
6      this.current = e;
7    }
8    //prueft, ob es noch weitere Eintraege gibt
9    public boolean hasNext(){
10     return current != null;
11   }
12   //liefert den naechsten Eintrag
13   public int next() {
14     if (current == null) {
15        throw new NoSuchElementException();
16     }
17     int res = current.getValue();
18     current = current.getNext();
19     return res;
20   }
21 }
```

Beispiel: Iterator für einfachverkettete Liste III

```
1   public class IntListTest {
22    public static void main(String[] args) {
3
4       IntList l = new IntList();
5       l.add(2);
6       l.add(-7);
7       l.add(34);
8
9       IntListIterator iter = l.iterator();
10      while(iter.hasNext()) {
11        StdOut.println(iter.next());
12      }
13    }
14  }
15
```

Bemerkung:

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

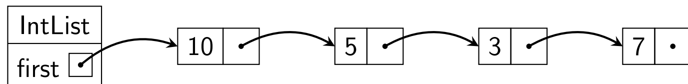
Weitere Listen-Implementierungen

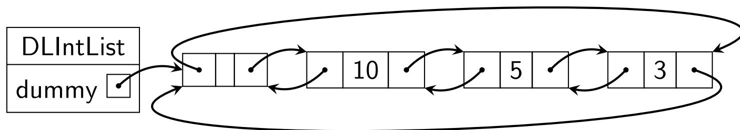
Doppeltverkettete Listen

- Einfachverkettete Listen kann man nur effizient in eine Richtung durchlaufen, nämlich vom ersten Element zum letzten.

Idee: Doppelte Verkettung
Jeder Knoten zeigt nicht nur auf den Nachfolgerknoten (`Node`-Attribute `next`), sondern auch auf den Vorgängerknoten (zusätzliches `Node`-Attribute `prev`).

Skizze





Bemerkungen

- Problem: Das Einfügen am Ende der Liste erfordert zuerst vollständige Durchlaufen der Liste.
  - Lösung 1: Verwaltete Referenz auf das letzte Element der Liste (siehe Übung)
  - Lösung 2: Füge ein Dummy-Element bei der doppeltverketteten Liste ein
    - `next`-Referenz zeigt auf ersten Knoten der Liste
    - `prev`-Referenz zeigt auf letztes Element der Liste
    - Dies vermeidet Spezialfälle bei Implementierung der Methoden (z.B. beim Einfügen am Anfang der Liste)

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Array-Listen

Grundidee: Array-Listen

- Elemente werden intern in einem Array gespeichert.
- Dazu wird zunächst ein Array mit einer bestimmten Anfangsgröße initialisiert.
- Das Array wird dann nach und nach mit neuen Elementen gefüllt.
- Wenn das Array zu klein für neue Elemente ist, wird ein größeres erstellt und die alten Elemente werden in das neue Array kopiert.

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Array-Liste: Konstruktor

```
public class IntArrayList {
    private int[] elementData;
    private int size;

    public IntArrayList(int initialArrayLength) {
      elementData = new int[initialArrayLength];
      size = 0;
    }
```

Array-Liste: Einfügen

```
// Element am Ende der Liste einfuegen
public void add(int element) {
    ensureCapacity(size + 1);
    elementData[size] = element;
    size++;
}

// stellt sicher, dass Array genug Platz hat
private void ensureCapacity(int minCapacity) {
    if (elementData.length < minCapacity) {
        // Groesse verdoppeln, mindestens auf minCapacity
        int newSize = Math.max(minCapacity,
                               2*elementData.length);
        elementData = Arrays.copyOf(elementData, newSize);
    }
}
```

Array-Liste: Element an Position

```
// liefert das Element an der gegebenen Position
public int get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    return elementData[index];
}
```

- Beachte: Operationen müssen sicher stellen, dass nur der Teil des Arrays benutzt wird, der gültig ist.

Array-Liste: Suchen eines Elements

```
/** gibt die erste Position eines Elements in der Liste
  *  oder -1, wenn das Element nicht in der Liste ist */
public int indexOf(int element) {
    for (int i=0; i<size; i++) {
        if (elementData[i] == element) {
            return i;
        }
    }
    return -1;
}
```

Array-Liste: Entfernen eines Elements

```
// entfernt das 1. Vorkommen eines Elements aus der Liste
public void remove(int element) {
    int indexOf = indexOf(element);
    if (indexOf < 0) {
        // Element nicht vorhanden
        return;
    }
    // alle Elemente nach dem ersten Vorkommen
    // nach links verschieben:
    for (int i=indexOf; i<size-1; i++) {
        elementData[i] = elementData[i+1];
    }
    size--;
}
```

- Anmerkung: Löschen ist im Vergleich zu verketteten Listen aufwendig, insbesondere das Löschen am Anfang der Liste.

Iterator für Array-Liste I

Idee: Der Iterator hat eine Referenz auf das interne Array der Array-Liste. Das Attribut `position` gibt den `index` des nächsten Elements an.

```
public class IntArrayListIterator {
    private int[] elementData;
    private int size;
    private int position;

    IntArrayListIterator(int[] elementData, int size) {
        this.elementData = elementData;
        this.size = size;
        this.position = 0;
    }
```

Iterator für Array-Liste II

```
    public boolean hasNext() {
        return position < size;
    }

    public int next() {
        int elem = elementData[position];
        position++;
        return elem;
    }

}
```

Iterator für Array-Liste III

Beim Erstellen des Iterators (in der Klasse `IntArrayList`) wird das interne Array an den Iterator übergeben:

```
// liefert einen Iterator fuer die Liste
public IntArrayListIterator iterator() {
    return new IntArrayListIterator(elementData, size);
}
```

Was passiert, wenn der Array-Liste Elemente hinzugefügt bzw. entfernt werden, während der Iterator noch über die Liste iteriert?