Grundlagen der Programmierung

Dr. Christian Herzog
Technische Universität München

Wintersemester 2016/2017

Kapitel 7: Reihungen und Listen

Überblick über dieses Kapitel

- Reihungen
  - ein- und mehrdimensionale Reihungen
  - Instantiierung von Reihungen
  - Reihungsvariable als Referenzvariable
  - sortierte Reihungen
- Geflechte
  - lineare Listen
  - sortierte lineare Listen
- Programmierbeispiele: Darstellung von Mengen
  - als Reihungen
  - als sortierte Reihungen
  - als sortierte lineare Listen

Reihungen

- Eine Reihung (*array*) ist eine Menge von Variablen, die Werte *desselben Typs* speichern.
- Eine einzelne Variable wird nicht durch ihren Namen benannt, sondern *durch ihre Position* innerhalb der Reihung.
- Beispiel: Ausgabe der Studentennamen, die in einem Verzeichnis gespeichert sind:

Ohne Reihung

```
System.out.println(student1);
System.out.println(student2);
System.out.println(student3);
System.out.println(student4);
System.out.println(student5);
System.out.println(student6);
System.out.println(student7);
```

Einzelne Variable

Mit Reihung

```
for (int k = 1; k <= 7; k++)
 System.out.println(student[k]);
```

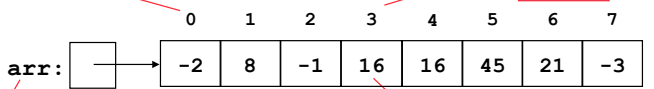
Der k-te Student

1-dimensionale Reihungen

- Eine Variable in einer Reihung wird durch ihre Position innerhalb der Reihung bezeichnet, nicht durch einen eigenen Bezeichner.
  - In einer Reihung von n Variablen mit dem Bezeichner `arr` werden die Variablen so benannt: `arr[0],arr[1],arr[2],…,arr[n-1]`.
- Die folgende Reihung umfasst 8 Variablen vom Typ `int`.

Die Indizierung von Reihungen beginnt in Java immer bei 0

Index



Reihungsname

Elementwert

Syntax einer Reihungsdeklaration:

```
typbezeichner[] arrayname;
```

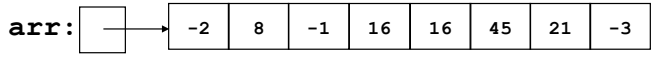
wobei

- `arrayname` der Name der Reihung ist, und
- `typbezeichner` der Typ der Reihungsvariablen.

Terminologie

- Die Variablen in einer Reihung heißen auch Reihungselemente oder kurz Elemente.
- Jedes Element in einer Reihung hat denselben Typ, auch Elementtyp genannt.
- Die Länge einer Reihung ist die Anzahl ihrer Elemente.
- Jedes Reihungsobjekt hat ein Attribut **`length`**, das die Länge der Reihung angibt.
  - **`arr.length`** ist im Beispiel der vorigen Folie gleich **`8`**
- Eine Reihung der Länge **`0`** (leere Reihung) enthält keine Variablen.
- Reihungselemente können von beliebigem Typ sein, einschließlich Reihungen und Klassen.
  - In Java: Eine 2-dimensionale Reihung ist eine Reihung von Reihungen von Elementen eines Typs

Zugriff auf Elemente in einer Reihung



Gegeben sei: **`int j = 5, k = 2;`**

Gültige Elementzugriffe sind:

```
arr[1]      // Bezeichnet den Wert 8
arr[0]      // Bezeichnet den Wert -2
arr[j + k]  // Ergibt arr[5+2], d.h. arr[7], also -3
arr[j % k]  // Ergibt arr[5%2], d.h. arr[1], also 8
```

Ungültige Elementzugriffe:

```
arr[5.0]   // 5.0 ist eine Gleitkommazahl (double),
           // kann also nicht Index sein.
arr['5']   // '5' ist vom Typ char, nicht von Typ int
arr[-1]    // Negative Indizes sind nicht möglich.
arr[8]     // Das letzte Element von arr hat Index 7
arr[j*k]   // j*k ist gleich 10, also außerhalb des
           // Indexbereiches
```


Deklaration und Instantiierung einer Reihung

- Um eine 1-dimensionale Reihung zu **deklarieren**, müssen wir den Namen der Reihung und auch den Typ der Elemente angeben.
- Um eine Reihung zu **instantiieren**, müssen wir den **`new`**-Operator benutzen, der den Elementtyp und die Länge der Reihung benötigt.

```
int[] arr;          // Deklaration einer Reihung
arr = new int[15]; // Instantiierung einer Reihung von 15
                    // Elementen
```

Wir können beide Aktivitäten in einer Anweisung vereinen:

```
int[] arr = new int[15];
```

Der Name der Reihung ist **`arr`**.

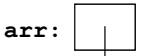
Die Reihung kann 15 Variablen vom Typ **`int`** enthalten.

Die 15 Variablen: **`arr[0], arr[1], .., arr[14]`**
Indexwerte von **`0`** bis **`arr.length-1`** zulässig

Was genau passiert bei Deklaration und Instantiierung einer Reihung?

Deklaration einer Reihungsvariable:

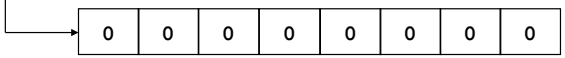
```
int[] arr;
```



Nach der Deklaration ist der Wert der Reihungsvariable **`null`.**

Die new-Operation instantiiert eine Reihung, deren Inhalte mit dem Default-Wert 0 vorbesetzt sind.

```
new int[8]
```



Die Zuweisung ersetzt den **`null`**-Wert durch einen Verweis auf das Reihungsobjekt.

```
arr = new int[8];
```

Initialisierung von Reihungen

- Bei der Deklaration werden Reihungen und Reihungselemente in Java mit **Voreinstellungswerten** (*default values*) initialisiert:
  - Reihungsvariablen werden auf den Wert **`null`** initialisiert.
  - Reihungselemente mit Typen **`int`**, **`boolean`**, … werden auf 0, false, ... initialisiert.
- Reihungselemente kann man bereits bei der Deklaration der Reihung mit initialen Werten versehen:

```
int[] arr = {-2,8,-1,-3,16,20,25,16,16,8,18,19,45,21,-2};
```

Regel: Wenn eine Reihung bereits bei der Deklaration initialisiert wird, dann brauchen wir den **`new`**-Operator nicht mehr, um die Reihung zu kreieren (die Instantiierung der Reihung hat bereits bei der Deklaration stattgefunden!).

Zuweisung und Benutzung von Reihungswerten

Indizierte Reihungsvariablen benutzen wir genauso wie andere Variablen:

```
arr[0] = 5;
arr[5] = 10;
arr[2] = 3;
```

Ein Programmstück, das die ersten 15 Fibonacci-Zahlen (0,1,1,2,3,5…) der Reihung **`arr`** zuweist:

```
arr[0] = 0;
arr[1] = 1:
for (int k = 2; k < arr.length; k++)
        arr[k] = arr[k-1] + arr[k-2];
```

Eine Schleife zum Drucken der Werte von **`arr`**:

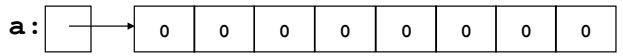
```
for (int k = 0; k < arr.length; k++)
        System.out.println(arr[k]);
```

Wichtig: **`length`** ist ein Attribut von **`arr`**, keine Methode.

Reihungsvariablen sind Referenzvariablen (Verweise)

Wir kennen schon die Situation nach folgender Deklaration und Instantiierung:

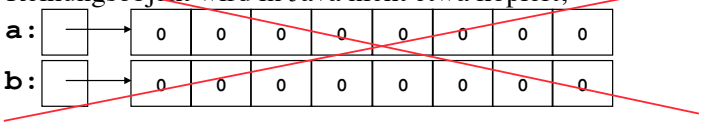
```
int[] a = new int[8];
```



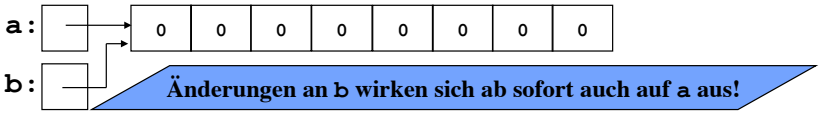
Was passiert nun, wenn wir eine zweite Reihungsvariable deklarieren und ihr den Wert von a zuweisen?

```
int[] b = a;
```

Das Reihungsobjekt wird in Java nicht etwa kopiert,



sondern es entsteht ein zweiter Verweis auf dasselbe Objekt!



Referenzvariablen in Java (siehe auch Folien 30-33 aus Kapitel 6)

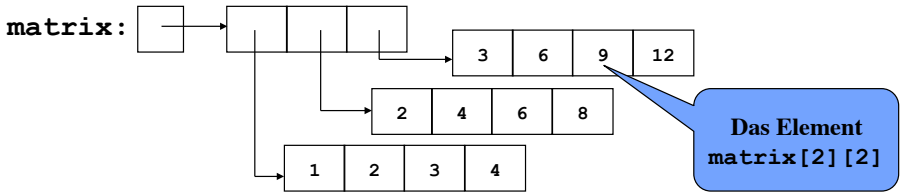
- In Java sind Variablen in der Regel **Referenzvariablen**!
  - Ausnahme: Variablen einfacher Typen wie **`int`**, **`double`**, **`boolean`**
- Bei der Zuweisung werden nicht die referenzierten Objekte kopiert sondern lediglich die Verweise auf diese Objekte.
  - Es entstehen mehrfache Verweise auf dasselbe Objekt.
  - Änderungen über den einen Verweis (über die eine Variable) beeinflussen also Objekte, die auch über den anderen Verweis (über die andere Variable) erreicht werden.
  - Es entsteht also das so genannte **Aliasnamen**-Problem.
- Auch bei der Parameterübergabe (*call by value*) werden nur Verweise kopiert, nicht jedoch die referenzierten Objekte.
- In Programmiersprachen wie Pascal dagegen werden bei der Zuweisung von Reihungsvariablen auch die Inhalte kopiert.
  - Das Referenzkonzept muss/kann dort explizit angefordert werden.
  - Z.B. bei der Parameterübergabe in Pascal mittels **`var`**.

Mehrdimensionale Reihungen

- Mehrdimensionale Reihungen sind in Java Reihungen, deren Elemente Reihungen sind, deren Elemente Reihungen sind, ...
- Eine zweidimensionale Matrix aus 3 Zeilen und 4 Spalten ganzer Zahlen wird in Java folgendermaßen deklariert, instantiiert und besetzt:

```
int[][] matrix = new int[3][4];
for (int row = 0; row < 3; row++)
   for (int col = 0; col < 4; col++)
      matrix[row][col] = (row+1) * (col+1);
```

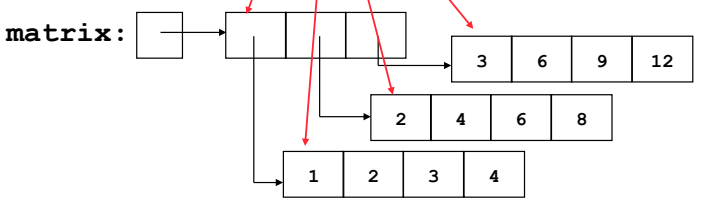
- Die entstehenden Reihungsobjekte lassen sich graphisch folgendermaßen veranschaulichen:



mehrdimensionale Reihungen (cont'd)

- Dasselbe Ergebnis bekommt man, wenn man die 3 Zeilen einzeln instantiiert:

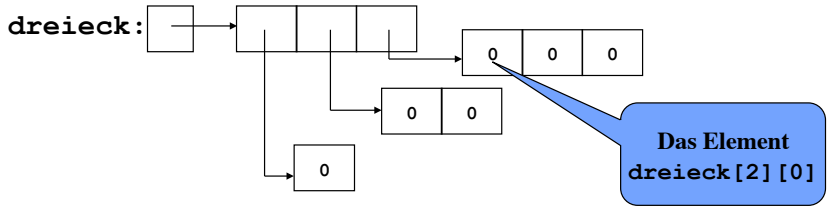
```
int[][] matrix = new int[3][];
for (int row = 0; row < 3; row++){
   matrix[row] = new int[4];
   for (int col = 0; col < 4; col++)
      matrix[row][col] = (row+1) * (col+1);
}
```



Dreiecksmatrizen

- Wenn die Zeilen 2-dimensionaler Reihungen einzeln instantiiert werden können, müssen sie auch nicht gleiche Länge haben.
- Damit lassen sich z.B. Dreiecksmatrizen darstellen:

```
int[][] dreieck = new int[3][];
dreieck[0] = new int[1];
dreieck[1] = new int[2];
dreieck[2] = new int[3];
```



Die for-each-Schleife (seit Java 5)

- Seit Java 5 gibt es eine komfortable Möglichkeit, die Elemente einer Reihung nacheinander aufzuzählen.
- Bisher:

```
double[] array;
...
double sum = 0;
for (int index = 0; index < array.length; index++)
   sum = sum + array[index];
```

- Mit der for-each-Schleife:

```
double[] array;
...
double sum = 0;
for (double element: array)
   sum = sum + element;
```

Die Variable `element` nimmt nacheinander die Werte `array[0],array[1],..., array[array.length-1]` an.

Wie geht es weiter?

- Wir werden uns nun ein Problem stellen und zur Lösung des Problems einige Java-Klassen komplett implementieren.
- Dies ist kein Spielbeispiel mehr, sondern könnte (mit einigen „professionellen“ Ergänzungen) innerhalb einer Klassenbibliothek zur Verfügung gestellt werden.
- Wir werden bei der Implementation immer wieder auf Probleme stoßen und zur Behebung dieser Probleme neue Konzepte kennen lernen.
- Wir werden bei der Implementation einige Programmiertechniken kennen lernen.
- Wir werden im Rahmen dieser Problemlösung
  - das bisher Gelernte vertiefen (z.B. Schleifen, Reihungen)
  - das Arbeiten mit Verweisstrukturen (Geflechte, Listen) kennen lernen
  - über dieses Kapitel hinaus objektorientierte Konzepte erarbeiten.

Ein größeres Beispiel: Darstellung von Mengen

- **Problemstellung:** Der Prüfungsausschuss der Fakultät für Informatik benötigt ein Studentenverwaltungssystem, das die anfallenden Arbeitsprozesse unterstützt.
- **Analyse:** Im Rahmen der Analyse kristallisiert sich u.a. heraus, dass Mengen ganzer Zahlen modelliert werden müssen (z.B. Mengen von Matrikelnummern als Vertretung von Mengen von Studenten).
- **Systementwurf:**
  - Eine Klasse wird die Modellierung von Mengen ganzer Zahlen übernehmen.
  - Die Schnittstelle bilden die üblichen Mengenoperationen (Einfügen, Löschen, Suchen, ...)
  - Nach unseren bisherigen Vorkenntnissen bieten sich Reihungen zur Modellierung von Mengen an.
  - Die Klasse soll entsprechend **`ArrayIntSet`** heißen.

Attribute und Methoden der Klasse `ArrayIntSet`

Eine mit `static final` gekennzeichnete Variable ist eine Konstante (keine weitere Zuweisung erlaubt).

```
class ArrayIntSet {
// Attribute (Datenstruktur)
 private static final int DEFAULT_CAPACITY = 10;
 private static final int DEFAULT_CAPACITY_INCREMENT = 5;
 private int currentSize;          // aktuelle Groesse der Menge
 private int[] array;              // speichert die Elemente der
                                   // Menge
// verschiedene Konstruktoren:
  ...

// sonstige Methoden:
 public boolean isEmpty(){…}       // ist Menge leer?
 public boolean contains(int i){…}// ist Element enthalten?
 public int size(){…}              // Groesse der Menge
 public void insert(int i){…}      // Einfuegen eines Elementes
 public void delete(int i){…}      // Entfernen eines Elementes
 public boolean isSubset(ArrayIntSet s){…}
                                   // ist Menge Teilmenge von s?
 public String toString() {…}      // Ausgabefunktion
}
```


Konstruktoren, die die leere Menge instantiiern

```
// verschiedene Konstruktoren fuer eine leere Menge:

    // Reihungskapazitaet richtet sich nach dem Default-Wert:
    public ArrayIntSet() {
      array = new int[DEFAULT_CAPACITY];
      currentSize = 0;
    }

    // gewuenschte Reihungskapazitaet wird uebergeben:
    public ArrayIntSet(int capacity) {
      array = new int[capacity];
      currentSize = 0;
    }
```

Die Auswahl des „passenden“ Konstruktors hängt von den Parametern beim Aufruf ab!

```
   // parameterloser Konstruktor:
ArrayIntSet s1 = new ArrayIntSet();
   // Konstruktor mit int-Parameter:
ArrayIntSet s2 = new ArrayIntSet(100);
```

Dieselben Konstruktoren, etwas „professioneller“ formuliert

```
// gewuenschte Reihungskapazitaet wird uebergeben:
public ArrayIntSet(int capacity) {
  array = new int[capacity];
  currentSize = 0;
}

// Reihungskapazitaet richtet sich nach dem Default-Wert:
public ArrayIntSet() {
  this(DEFAULT_CAPACITY);
}
```

this meint hier einen Konstruktor der eigenen Klasse! Dieser this-Aufruf ist nur als erste Anweisung im Rumpf erlaubt.

Der Konstruktor mit weniger Parametern stützt sich auf einen mit mehr Parametern ab.

Konstruktor, der die Menge als die Kopie einer anderen Menge instantiiert

```
// Konstruktor, der die Kopie einer anderen Menge liefert:

public ArrayIntSet(ArrayIntSet s) {
  currentSize = s.size();

            // die Reihungsgroesse wird so gewaehlt, dass
            // zusaetzliche Elemente Platz finden:
  if (currentSize < DEFAULT_CAPACITY)
    array = new int[DEFAULT_CAPACITY];
  else
    array = new int[currentSize + DEFAULT_CAPACITY_INCREMENT];

            // Die Elemente aus s werden uebertragen:
  for (int index=0; index<currentSize; index++)
    array[index] = s.array[index];
}
```

Aufruf :

```
ArrayIntSet s2 = new ArrayIntSet(s1);
```


Die Methoden `isEmpty()`, `contains()` und `size()`

```
            // Abfrage, ob Menge leer ist:
public boolean isEmpty() {
  return size() == 0;
}

            // Abfrage, ob Element enthalten ist:
public boolean contains(int i) {
  for(int index=0; index<currentSize; index++) {
      if (array[index] == i)
        return true;
  }
  return false;
}

            // Abfrage nach Groesse der Menge:
public int size() {
  return currentSize;
}
```


Die Methode `insert()`

```
            // Einfuegen eines Elementes:
public void insert(int i) {
            // i darf noch nicht enthalten sein:
  if (contains(i)) {
    System.out.println("insert: " + i + " schon enthalten!");
    return;
  }
            // wenn neues Element nicht hineinpasst:
  if (currentSize >= array.length) {
    System.out.println("insert: Kein Platz mehr fuer " + i);
    return;
  }
            // Sonst: Speichern von i auf erstem freien
            //        Platz:
  array[currentSize] = i;
            // Konsistente Erhoehung von currentSize:
  currentSize++;
}
```

Konkatenation auf dem Typ `String`

Die Methode `insert()` - Problem mit fester Reihungsgröße

Problem:
- **Diese Lösung ist völlig unflexibel.**
- **Die maximale Größe muss bekannt sein, bzw. vorausgeahnt werden können.**

```
public void inse
  if (contains(i
    System.out.println(                    + i + " schon enthalten!");
    return;
  }
                  // wenn neues Element nicht hineinpasst:
  if (currentSize >= array.length) {
    System.out.println("insert: Kein Platz mehr fuer " + i);
    return;
  }
                  // Sonst: Speichern von i auf erstem freien
                  //        Platz:
  array[currentSize] = i;
                  // Konsistente Erhoehung von currentSize:
  currentSize++;
}
```


Die Methode `insert()` - geht es auch anders?

```
                  // Einfuegen eines Elementes:
public void insert(int i) {
                  // i darf noch nicht enthalten sein:
  if (contains(i)) {
    System.out.println("insert: " + i + " schon enthalten!");
    return;
  }
                  // wenn neues Element nicht hineinpasst:
```

Alternative Lösungsmöglichkeit:
- **eine neue, größere Reihung generieren**
- **die alte Reihung dorthin umspeichern**
- **das neue Element hinzufügen**

(Diese Variante verwendet z.B. die vorgefertigte Klasse `java.util.Vector`.)

```
  array[currentSize] = i;
                  // Konsistente Erhoehung von currentSize:
  currentSize++;
}
```

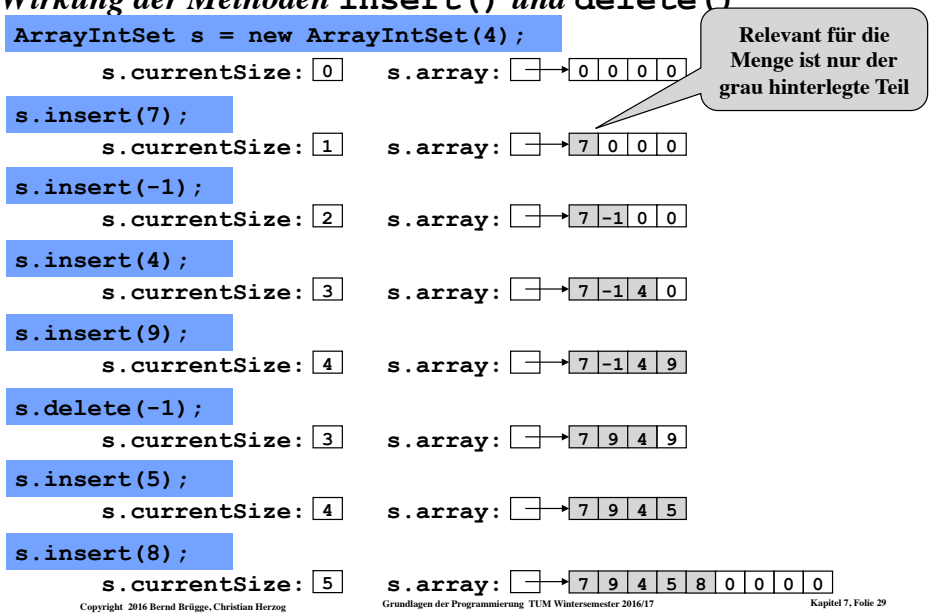

Die Methode `insert()` - eine flexiblere Lösungsvariante

```
                  // Einfuegen eines Elementes:
public void insert(int i) {
                  // i darf noch nicht enthalten sein:
  if (contains(i)) {
    System.out.println("insert: " + i + " schon enthalten!");
    return;
  }
                  // wenn neues Element nicht hineinpasst:
  if (currentSize >= array.length) {
                  // alte Reihung zwischenspeichern:
    int[] oldArray = array;
                  // array eine neue,groessere Reih. zuweisen:
    array = new int[1+currentSize+DEFAULT_CAPACITY_INCREMENT];
                  // Werte umspeichern:
    for (int index = 0; index < currentSize; index++)
      array[index] = oldArray[index];
  }
                  // Speichern von i auf erstem freien Platz:
  array[currentSize] = i;
                  // Konsistente Erhoehung von currentSize:
  currentSize++;
}
```


Die Methode `delete()`

```
                  // Entfernen eines Elementes:
public void delete(int i) {
                  // Indexposition von i ermitteln:
  for (int index = 0; index < currentSize; index++) {
    if (array[index] == i) {
                  // i steht auf Position index; i wird
                  // geloescht, indem das rechteste Element
                  // auf Position index verschoben wird:
      array[index] = array[currentSize-1];
                  // Konsistente Verminderung von currentSize:
      currentSize--;
      return;
    }
  }
                  // ansonsten i nicht in Menge enthalten
  System.out.println("delete: " + i + " nicht enthalten!");
}
```

Wirkung der Methoden `insert()` *und* `delete()`



Die Methode `isSubset()`

```
                    // Abfrage nach Teilmengeneigenschaft:

  public boolean isSubset(ArrayIntSet s) {

                    // Bei jedem Element der Menge wird
                    // ueberprueft, ob es in der anderen
                    // Menge s enhalten ist:

    for (int index=0; index < currentSize; index++) {
        if (! s.contains(array[index]))
                    // Teilmengeneigenschaft verletzt:
          return false;
    }

                    // Teilmengeneigenschaft nie verletzt:
    return true;
  }
```


Die Methode `toString()`

- Falls in einer Klasse eine Methode
  `public String toString() {…}`
  definiert ist, dann wird sie zur (automatischen) Konvertierung von Objekten dieser Klasse in Zeichenreihen verwendet.
  - z.B. bei `System.out.println(…)`

```
                    // Ausgabefunktion:
  public String toString() {
    String result = "{";
    for(int index=0; index<currentSize; index++) {
        result += array[index];
        if (index < currentSize - 1)
            result += ", ";
    }
    return result + "}";
  }
```

Konkatenation auf dem Typ `String`

Hier wird (wie schon häufig) die automatische Konvertierung von `int` in `String` verwendet!

Rückbetrachtungen zur Klasse ArrayIntSet

- Behandlung fehlerhafter Aufrufe:
  - Die Methoden `insert(i)` und `delete(i)` setzen voraus, dass `i` in der dargestellten Menge noch nicht vorhanden bzw. vorhanden ist.
  - In beiden Fällen hätte man auch reagieren können, indem man „nichts tut“.
    - Der Anwender würde dann nichts davon erfahren, dass die Aufrufe wirkungslos geblieben sind.
  - Andererseits: Ist der Ausdruck einer Fehlermeldung die adäquate Methode, den Anwender aufmerksam zu machen???
    - Was geschieht, wenn der Anwender nicht „lesen“ kann, weil er z.B. ein anderes Informatiksystem ist?
  - Man hätte in beiden Fällen auch das Programm abbrechen können.
    - Sind die Fehler so schwerwiegend, dass dies nötig ist?
  - Wie werden später mit dem Konzept der **Ausnahmen** (*exceptions*) andere Möglichkeiten der Fehlerbehandlung kennenlernen.
    - Exceptions informieren den Anwender mittels einer definierten Schnittstelle über das Auftreten einer außergewöhnlichen Sitation.
    - Der Anwender kann entscheiden, wie er reagiert.

Weitere Rückbetrachtungen zur Klasse ArrayIntSet

- Die Implementation der Mengendarstellung ist nicht effizient, da die Elemente ungeordnet in der Reihung abgelegt sind.
  - Man muss den relevanten Teil der Reihung vollständig durchlaufen, um festzustellen, dass ein Element nicht enthalten ist.
  - Falls die Elemente der Größe nach angeordnet wären, könnte man die Suche beenden, sobald man auf ein größeres als das zu suchende Element trifft.
  - Der durchschnittliche Laufzeit-Aufwand für die Methode **`contains()`** würde sich dann verringern.
  - Auch die Methode **`isSubset()`**, die bisher für jedes Element der Menge einmal die Methode **`contains()`** aufruft, ließe sich wesentlich effizienter implementieren.
  - Allerdings müssen die Methoden **`insert()`** und **`delete()`** dafür sorgen, dass die Reihenfolge der Elemente korrekt ist.

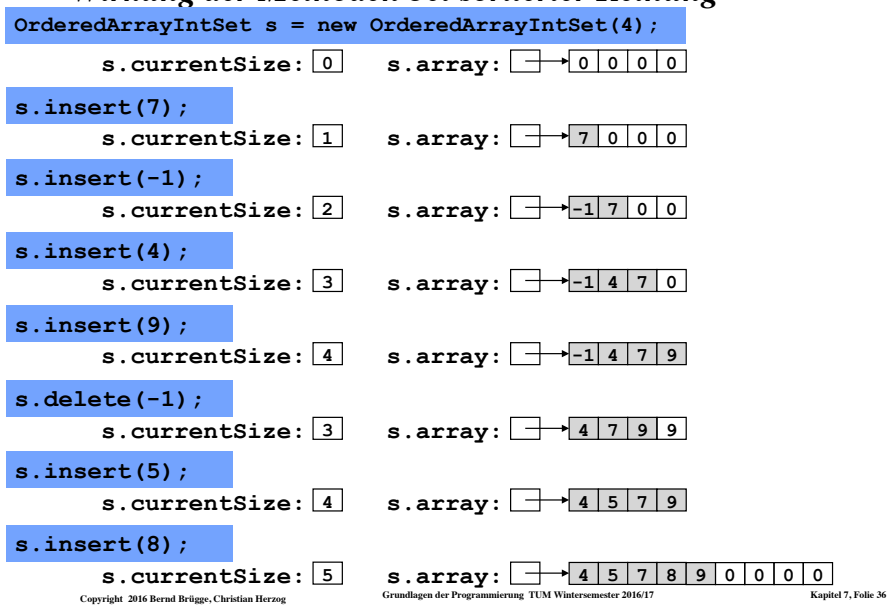
Sortierte Reihungen

- **Definition sortierte Reihung**: Ist **`arr`** eine Reihung über einem Typ, auf dem eine Ordnung definiert ist (z.B. **`int`**), dann heißt **`arr`** **sortiert**, wenn die Elemente in **`arr`** in aufsteigender Reihenfolge angeordnet sind:
  - **`arr[i] <= arr[i+1]`** für **`0 <= i < arr.length-1`**

Mengendarstellung auf sortierter Reihung: die Klasse OrderedArrayIntSet

- Wir werden nun mit der Klasse **`OrderedArrayIntSet`** eine alternative Implementation von Mengen ganzer Zahlen auf sortierten Reihungen angeben.
- Vorgehensweise:
  - Wir kopieren die Datei **`ArrayIntSet.java`** in eine Datei **`OrderedArrayIntSet.java`** und ersetzen konsistent die Bezeichnung **`ArrayIntSet`** durch **`OrderedArrayIntSet`** .
  - Die Konstruktoren und die Methoden **`isEmpty()`**, **`size()`** und **`toString()`** können danach unverändert erhalten bleiben, da sie weder die Sortiertheit der Reihung verletzen noch die Sortiertheit zur Effizienzsteigerung ausnutzen können.
  - Für die Methoden **`contains()`** , **`insert()`** , **`delete()`** und **`isSubset()`** werden neue Fassungen geschrieben.

Wirkung der Methoden bei sortierter Reihung



Die Methode `contains()` *bei sortierter Reihung*

- Die Reihung wird durchlaufen, bis einer der folgenden Fälle zutrifft:
  - das Element wird gefunden: Ergebnis **`true`**
  - (neu:) ein größeres Element wird erreicht: Ergebnis **`false`**
  - das Ende der Reihung wird erreicht: Ergebnis **`false`**

```
public boolean contains(int i) {
  for(int index=0; index<currentSize; index++) {
    if (array[index] == i) // Element gefunden
      return true;
    if (array[index] > i)  // größeres Element erreicht
      return false;
  }
  return false;      // Ansonsten Element nicht enthalten
}
```


Die Methode `insert()` *bei sortierter Reihung*

- Gesucht wird die „Nahtstelle“ zwischen den vorderen Elementen, die kleiner als **`i`** sind, und den hinteren, die größer sind.
- Dort wird dann Platz für das neue Element **`i`** geschaffen.

```
public void insert(int i) {
                    // Zunaechst Überspringen der kleineren
                    // Elemente bei der Suche nach der
                    // Einfuegestelle:
  int index = 0;
  while (index < currentSize && array[index] < i)
    index++;
                    // i darf noch nicht enthalten sein:
  if (index < currentSize && array[index] == i) {
    System.out.println("insert: " + i + " schon enthalten!");
    return;
  }
    // auf array[index] wird nun Platz für
    // i geschaffen
    // dieser Teil steht auf der nächsten Folie
  ...
  array[index] = i; // Speichern von i auf Position index
  currentSize++;    // Konsistente Erhoehung von currentSize
}
```

Sequentieller Operator **`&&`** verhindert unzulässige Reihungszugriffe

Die Methode `insert()` *bei sortierter Reihung (cont‘d)*

- Es fehlt noch der Teil der Methode, der für das neue Element i Platz auf der ermittelten Einfüge-Position index schafft:

```
                    // wenn ein neues Element noch hineinpasst:
  if (currentSize < array.length)
                    // Verschieben der restlichen Elemente nach
                    // rechts:
    for (int k=currentSize-1; k>=index; k--)
      array[k+1] = array[k];
  else {            // der Fall, dass groessere Reihung noetig:
                    // alte Reihung zwischenspeichern:
    int[] oldArray = array;
                    // Neue Reihung anlegen:
    array = new int[1+currentSize+DEFAULT_CAPACITY_INCREMENT];
                    // Umspeichern der vorne liegenden Elemente:
    for (int k=0; k<index; k++)
      array[k] = oldArray[k];
                    // Umspeichern der hinten liegenden Elemente
                    // mit Luecke bei index:
    for (int k=index; k<currentSize; k++)
      array[k+1] = oldArray[k];
  }
```


Die Methode `delete()` *bei sortierter Reihung*

```
public void delete(int i) {
                    // Indexposition von i ermitteln:
  int index = 0;
  while (index < currentSize && array[index] < i)
    index++;
                    // Falls dabei Reihenende oder groesseres
                    // Element erreicht, ist i nicht enthalten:
  if (index >= currentSize || array[index] > i) {
    System.out.println("delete: " + i + " nicht enthalten!");
    return;
  }
                    // Sonst steht i auf Position index; i wird
                    // geloescht, indem die Elemente rechts von
                    // Position index nach links umgespeichert
                    // werden
  for (int k=index+1; k<currentSize; k++)
    array[k-1] = array[k];
                    // Konsistente Verminderung von currentSize:
  currentSize--;
}
```

Ein „Schmankerl“: Implementation der Methode `isSubset()` *bei sortierter Reihung*

```
public boolean isSubset(OrderedArrayIntSet s) {
  int index = 0;        // Index der Menge selbst
  int indexS = 0;       // Index der anderen Menge s

  while (index < currentSize && indexS < s.size()) {
    if (array[index] < s.array[indexS])
              // Element der Menge kann nicht auch in s sein
      return false;
    if (array[index] > s.array[indexS])
              // s weiterschalten
      indexS++;
    else {
              // Element der Menge ist auch in s; beide
              // Indizes weiterschalten:
      index++;
      indexS++;
    }
  }
      // Teilmengeneigenschaft ist genau dann erfuellt, wenn
      // index die gesamte Menge durchlaufen hat:
  return index >= currentSize;
}
```

Zugriff auf `private`-Attribut ist erlaubt! Sichtbarkeit ist auf Klassen, nicht auf Objekten definiert.

Rückbetrachtungen zur Klasse OrderedArrayIntSet

- Um die Sortiertheit zu erhalten, müssen bei den Methoden `insert(i)` und `delete(i)` jeweils alle Elemente, die größer sind als `i`, um eine Position nach rechts bzw. nach links verschoben werden.
- Hier wäre eine *flexiblere* Datenstruktur wünschenswert, die es erlaubt
  - an beliebiger Stelle Platz für ein neues Element einzufügen und
  - an beliebiger Stelle den für ein Element vorgesehenen Platz zu entfernen.
- Diese Flexibilität erreicht man, wenn Elemente nicht starr einer Indexposition zugeordnet sind, sondern selbst Verweise auf benachbarte Elemente beinhalten.
  - Diese *Verweise* können dann geeignet umgelenkt werden.
  - Die entstehenden Verweisstrukturen nennt man **Geflechte**.
  - Für die Mengendarstellung werden wir nun lineare Geflechte betrachten, in der jedes Element einen Verweis auf seinen Nachfolger beinhaltet, sogenannte **Listen**.

Beispiel einer Liste

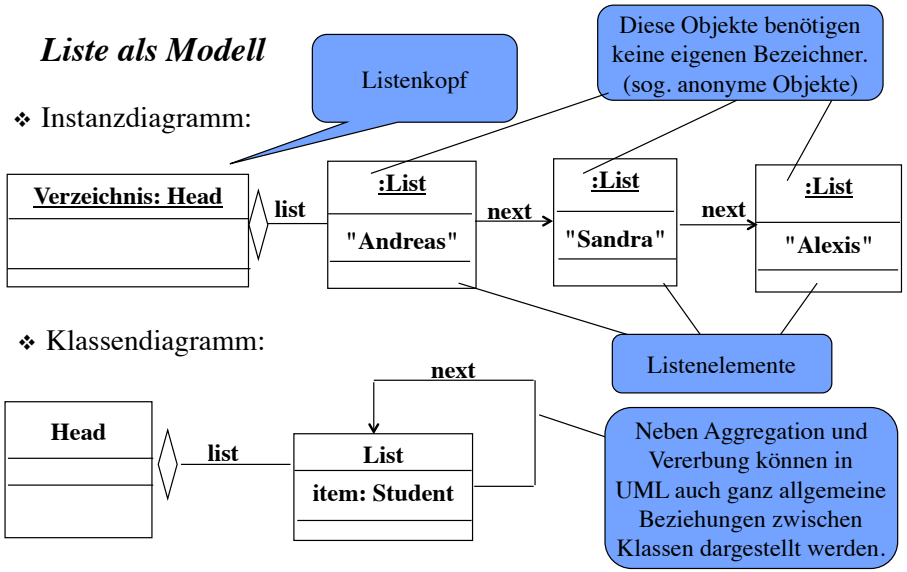
- Beispiel:
  - Wir sind auf einer Party, auf der auch Andreas, Helmut, Sandra, Opa und Barbie sind.
    - Andreas weiß, wo Helmut ist.
    - Helmut weiß, wo Sandra ist.
    - Sandra weiß, wo Opa ist
    - Opa weiß, wo Barbie ist.
- Um Sandra etwas zu sagen, muss ich es Andreas sagen. Der sagt es dann Helmut und der Sandra.
- Um Barbie zu finden, muss ich ...

Modellierung von Listen

- Grundbaustein einer verketteten Liste ist das Listenelement.
  - Ein Listenelement enthält immer folgende zwei Attribute:
    - Eine Referenz auf das nächste Listenelement (next)
    - Anwendungsspezifische Daten (item)
- Beispiel:
  - Das Studentenverzeichnis aller Erstsemester an der TUM
  - Die Immatrikulation hat gerade begonnen.
    - Das Studentenverzeichnis besteht aus 3 Studenten:
    - Andreas, Sandra, Alexis
- Wir können das Verzeichnis als verkettete Liste modellieren.
  - Andreas, Sandra und Alexis sind dann die anwendungsspezifischen Daten (vom Typ Student).

Liste als Modell

- Instanzdiagramm:



- Klassendiagramm:

Implementation des Listenelementes in Java

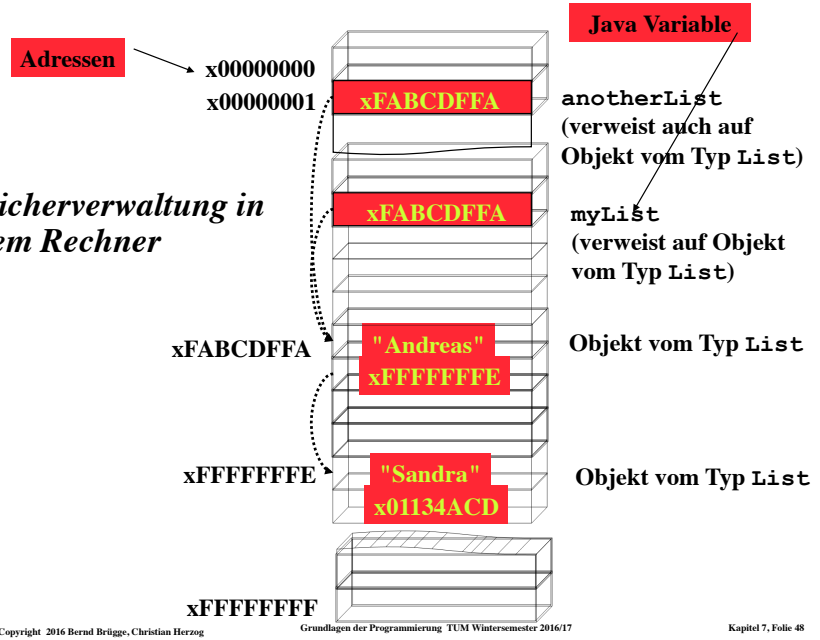
```
class List {
   private List next;
   private Student item;
   ...
}
```

Eine derartige Klassendefinition heißt auch rekursiv (self-referential), da sie ein Attribut enthält (in diesem Fall namens **`next`**), das von demselben Typ ist wie die Klasse selbst.

Noch einmal: Referenzvariablen (Verweise) in Java

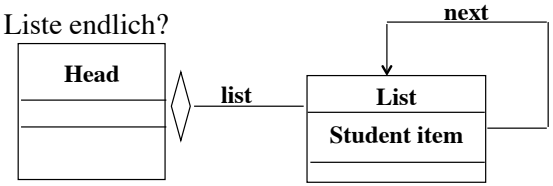
- Kann man ein Attribut vom Typ **`List`** in einer Klassendefinition vom Typ **`List`** verwenden?
  - Wird dadurch das Listenelement nicht unendlich groß?
- Das **`List`** -Attribut enthält kein weiteres **`List`** -Objekt, auch wenn es so aussieht. Das **`List`** -Attribut ist eine *Referenzvariable*:
  - Es enthält als Wert einen Verweis auf ein Objekt vom Typ **`List`**.
  - Referenzen sind Adressen im Speicher des Rechners.
    - Alle Adressen in einem Rechner haben die gleiche Größe.
    - Eine Referenzvariable belegt nur soviel Speicher, wie zur Speicherung einer Adresse benötigt wird.
    - Daher ist es kein Problem für den Java-Compiler, die Größe von rekursiv definierten Listenelementen zu berechnen.

Speicherverwaltung in einem Rechner



Ein besonderer Verweis: `null`

- Ist eine Liste endlich?

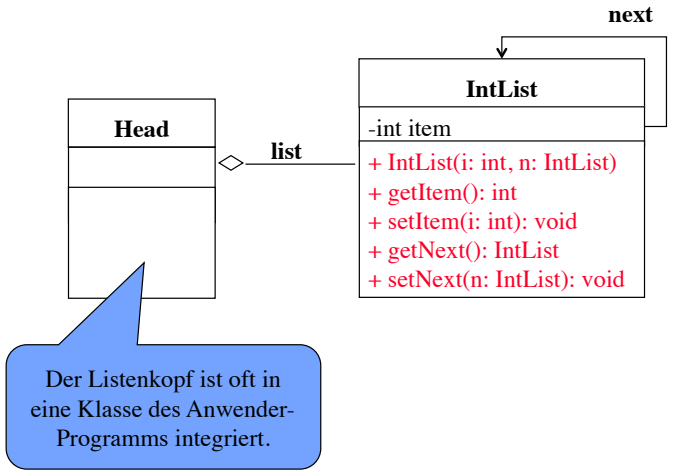


- Es sieht so aus, als könnte die Rekursion über den next-Verweis niemals zum Ende kommen.
- Um auszudrücken, dass eine Referenzvariable auf kein (weiteres) Objekt verweisen soll, wird ein spezieller Verweis (eine spezielle Adresse) eingeführt: der **`null`**-Verweis.
  - Der **`null`**-Verweis unterscheidet sich von jedem anderen Verweis (von jeder zulässigen Adresse).
  - Insbesondere unterscheidet er sich auch von fehlerhaften Verweisen auf zulässige Speicheradressen.
- Die **leere Liste** wird dargestellt, indem die Referenzvariable **`list`** im Listenkopf mit **`null`** besetzt wird.

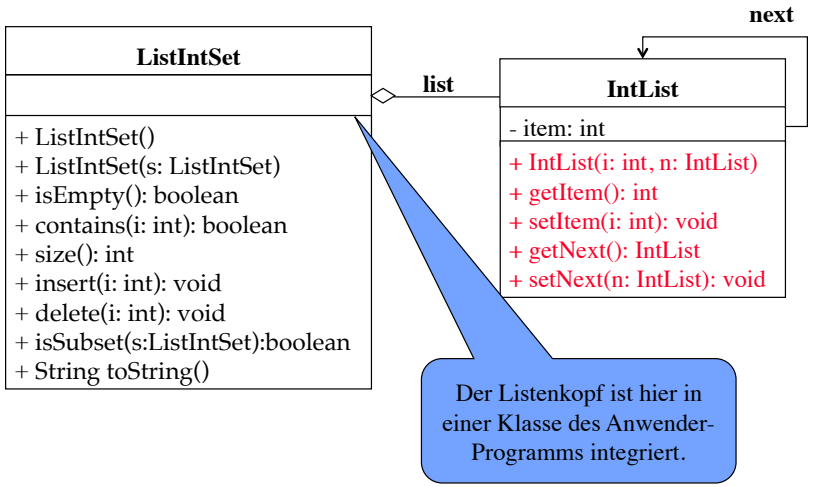
Java-Implementation von `int`-Listenelementen

```
class IntList {
  // Inhalt des Listenelements:
  private int item;
  // Naechstes Listenelement:
  private IntList next;

  // Konstruktor:
  public IntList (int i, IntList n)
    // Initialisiere Inhalt:
    item = i;
    // Initialisiere next-Verweis:
    next = n;
  }

  // Methoden:
  public int getItem () {
    return item;
  }

  public IntList getNext () {
    return next;
  }

  public void setItem (int i) {
    item = i;
  }

  public void setNext (IntList n)
  {
    next = n;
  }
} // end class IntList
```


Modell der Implementation von IntList



Beispiel: Mengendarstellung durch Listen:

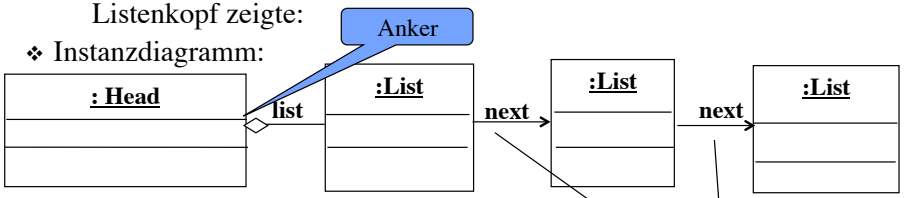


Typen von Listen

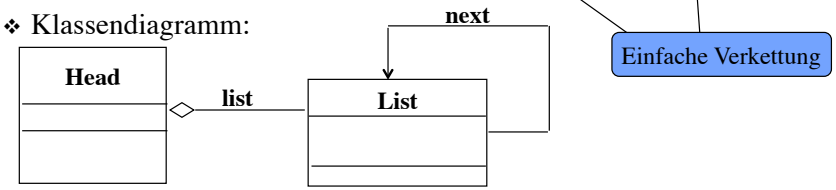
- Es gibt verschiedene Typen von Listen:
  - Einfach verkettete Listen
  - Doppelt verkettete Listen
  - Listen mit einem oder mit zwei Ankern im Listenkopf
  - Sortierte Listen

Einfach verkettete (lineare) Liste mit einem Anker

- Bisher haben wir nur Listen betrachtet,
  - deren Elemente über einen einzigen Verweis (next) verbunden (verkettet) waren,
  - Bei denen nur auf das erste Element ein Verweis (Anker) aus dem Listenkopf zeigte:
- Instanzdiagramm:

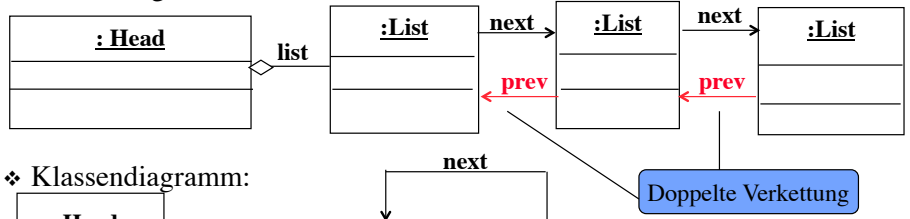


- Klassendiagramm:

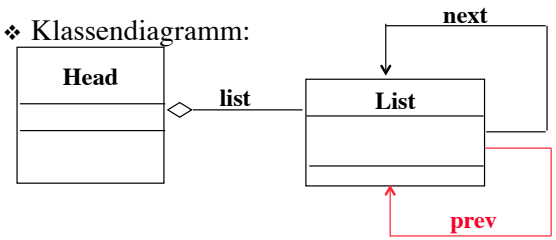


Doppelt verkettete (lineare) Liste mit einem Anker

- Oft ist es nötig, nicht nur vom Listenanfang zum Ende „laufen“ zu können, sondern auch in die umgekehrte Richtung.
- Dazu wird eine zweite Verkettung über einen **`prev`**-Verweis eingeführt.
- Instanzdiagramm:

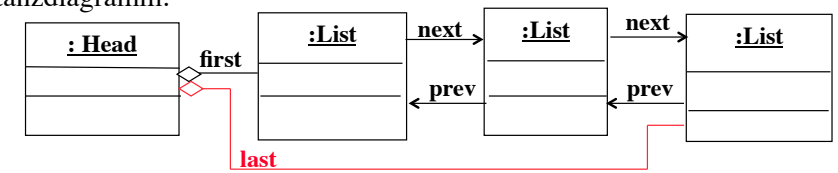


- Klassendiagramm:

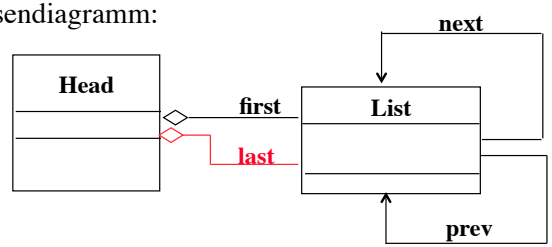


Doppelt verkettete (lineare) Liste mit zwei Ankern

- Vollständig symmetrisch in Listenanfang und Listenende wird die doppelt verkettete Liste, wenn vom Listenkopf auch noch ein zweiter Verweis auf das letzte Element geführt wird.
- Instanzdiagramm:



- Klassendiagramm:

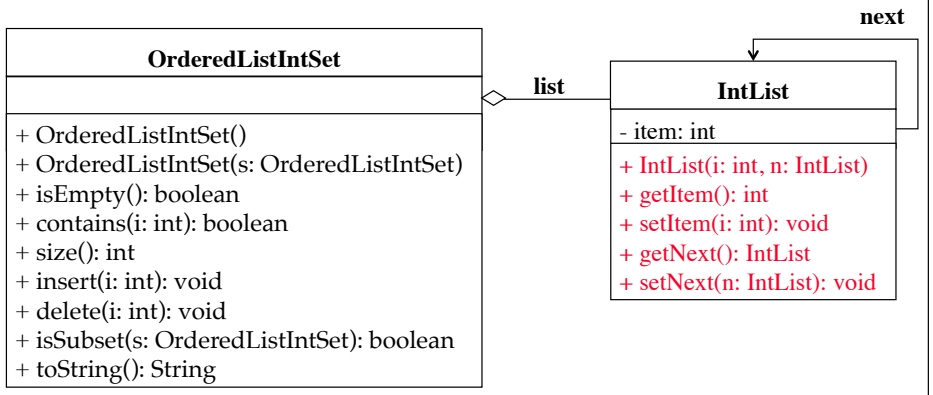


Sortierte lineare Liste

- Wie bei Reihungen ist es zum schnellen Auffinden von Elementen oft günstiger, wenn die Inhalte der Listenelemente beim Durchlauf vom Listenanfang zum Listenende in aufsteigender Reihenfolge angeordnet sind.
- Wir sprechen dann von einer **sortierten linearen Liste**.

- Für unsere ursprüngliche Aufgabenstellung, Mengen ganzer Zahlen darzustellen, wollen wir nun für den Rest dieses Kapitels *einfach verkettete, sortierte lineare Listen mit einem Anker* verwenden.

Mengendarstellung durch sortierte lineare Listen: Modellierung (1. Variante)



Diskussion dieser ersten Modellierungsvariante

- Ganz analog zur Klasse OrderedArrayIntSet können mit dieser Modellierung die Algorithmen für die benötigten Methoden implementiert werden.
  - Bei dieser Variante bleibt die Verantwortung für die Sortiertheit der Liste, d.h. für die Integrität der Daten, bei der Anwendungsklasse **`OrderedListIntSet`**
  - Dafür müssen die Methoden **`setItem()`** und **`setNext()`** der Klasse **`IntList`** öffentlich sein.
  - Von außen kann also jederzeit die Sortiertheit zerstört werden.
- Alternative Modellierungsvariante:
- Die sortierte Liste wird in einer Klasse **`OrderedIntList`** zur Verfügung gestellt, deren Schnittstelle Integrität der Daten sicherstellt.
  - Von außen kann die Sortiertheit dann nicht mehr zerstört werden.

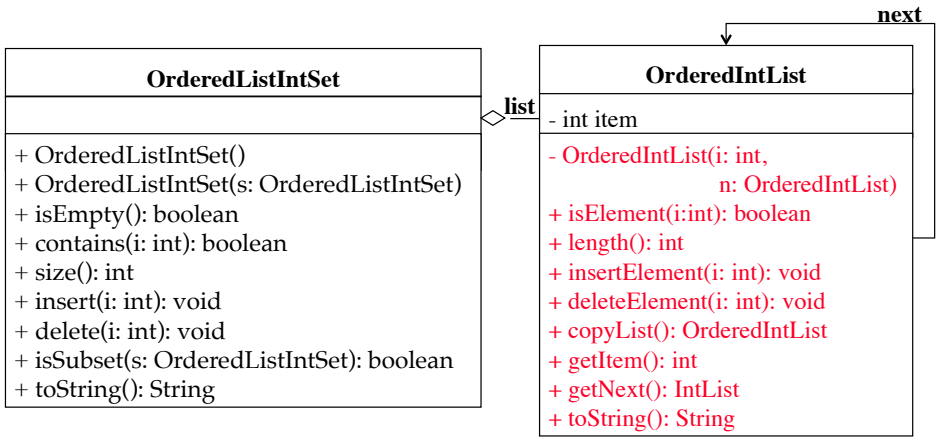
Benötigte Schnittstelle der Klasse `OrderedIntList`

- Die Schnittstelle von **`OrderedIntList`** muss die Implemantation aller Methoden von **`OrderedListIntSet`** unterstützen, die den Inhalt oder den Aufbau der Liste verändern.
- Dazu stellen wir in **`OrderedIntList`** folgende Methoden bereit:
  - **`insertElement()`** für die Methode **`insert()`**
  - **`deleteElement()`** für die Methode **`delete()`**
  - **`copyList()`** für den Copy-Konstruktor
- Auch die anderen Methoden, die die Datenstruktur nur lesen, nicht jedoch verändern, lassen sich direkt in **`OrderedIntList`** eleganter (weil rekursiv) realisieren.

Benötigte Schnittstelle der Klasse `OrderedIntList` (cont'd)

- Die Schnittstelle von **`OrderedIntList`** enthält deshalb auch folgende Methoden:
  - **`isElement()`** für die Methode **`contains()`**
  - **`length()`** für die Methode **`size()`**
  - **`toString()`** für die Methode **`toString()`**
- Lediglich den Algorithmus für **`isSubset()`** wollen wir (um auch diese Variante zu üben) direkt in **`OrderedListIntSet`** implementieren.
  - Dafür benötigen wir in der Schnittstelle noch die Methoden **`getItem()`** und **`getNext()`**

Mengendarstellung durch sortierte lineare Listen: Modellierung (2. Variante)



Die Klasse `OrderedIntList` für sortierte lineare Listen

```
class OrderedIntList {

// Attribute:
  private int item;          // Inhalt des Listenelements
  private OrderedIntList next;     // Verweis auf nächstes Element

// Konstruktor:

  // Inhalt und Nachfolgeelement werden als Parameter übergeben:
  private OrderedIntList (int item, OrderedIntList next) {
    this.item = item;
    this.next = next;
  }

// Methoden:
  ...
}
```

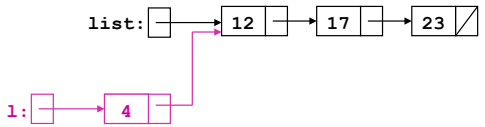
Nach der Gültigkeitsregel für Variable:
`item` und **`next`** sind als Parameter lokale Variable der Methode und verschatten die entsprechenden Instanzvariablen.
Diese können aber über **`this`** angesprochen werden.

`this` ist immer ein Verweis auf das Objekt selbst.

Wirkungsweise des Konstruktors der Klasse `OrderedIntList`

```
// Konstruktor:
private OrderedIntList (int item, OrderedIntList next) {
  this.item = item;
  this.next = next;
}
```

- Gegeben sei bereits eine Liste:
  - Auf das erste Element dieser Liste verweise die Variable **`list`**.



- Dann ist dies die Wirkungsweise der Anweisung **`OrderedIntList l = new OrderedIntList(4, list);`**
- **Achtung**: der Konstruktor sorgt nicht selbst für die Sortiertheit der Liste.
  - Diese ist nur gewährleistet, wenn alle „alten" Elemente größer als das „neue" sind.
  - Deshalb ist der Konstruktor nicht **`public`** sondern **`private`**.
  - Er gehört also nicht zur Schnittstelle.

Die Methode `length()` *für die Länge einer Liste*

```
// length liefert die Laenge einer Liste:
public int length () {
   if (next == null)
      return 1;
   return 1 + next.length();
}
```

Eine rekursive Methode für die rekursive Datenstruktur!

- Diskussion dieser Lösung:
  - Die Methode `length()` liefert niemals den Wert `0`, da die leere Liste nicht als Objekt sondern als `null`-Verweis dargestellt wird.
  - `length()` ist aber Methode eines Objekts.
- Eine Alternative wäre es, `length()` nicht als sog. **Instanz-Methode** sondern als **Klassen-Methode** zu realisieren.
- Klassen-Methoden sind nicht Merkmale von Objekten sondern *Dienste* einer Klasse.
  - Sie haben kein `this`-Objekt, auf dessen Attribute sie direkt zugreifen können.
  - Klassen-Methoden können benutzt werden, ohne vorher ein Objekt zu instantiiern.

Die Methode `length()` *als Klassen-Methode*

- Klassenmethoden werden durch das Wortsymbol `static` gekennzeichnet.
- Da ihnen kein Objekt direkt zugeordnet ist, müssen Objekte als Parameter übergeben bzw. als Ergebnis ausgeliefert werden.

```
// length als Klassen-Methode:
public static int length (OrderedIntList l) {
   if (l == null)
      return 0;
   return 1 + length(l.next);
}
```

- Wir werden uns für diese Alternative entscheiden.
- Beim Aufruf (außerhalb der Klasse) wird nicht ein Objektbezeichner sondern der Klassenname vorangestellt:
  - `int size = OrderedListInt.length(list);`

Klassen-Variable

- Neben Klassen-*Methoden* gibt es auch Klassen-*Variable*.
  - Auch sie werden durch `static` gekennzeichnet.
  - Eine Klassen-Variable gibt es **einmal pro Klasse**.
  - Eine Instanz-Variable gibt es **einmal pro Objekt**.
- Eine Klassen-Variable kann z.B. zählen, wie viele Objekte der Klasse instantiiert werden:

```
class MitNummer {
            // Klassenvariable laufendeNummer wird bei jeder
            // Instantiierung eines Objekts erhoeht:
   private static int laufendeNummer = 0;
            // Instanzvariable meineNummer enthält für jedes Objekt
            // eine andere, eindeutige Nummer:
   private int meineNummer;
   ...      // weitere Attribute

// Konstruktor:
   public MitNummer () {
            // Erhöhen der laufenden Nummer:
      laufendeNummer++;
            // Nummerierung des neuen Objekts:
      meineNummer = laufendeNummer;
      ...
   }
...
```

Gibt es nur einmal!

Für jede Instanz wird ein eigenes Exemplar generiert!

Klassen-Methoden und Klassen-Variable

- Instanz-Methoden
  - haben Zugriff auf die Klassen-Variablen
  - und die Instanz-Variablen des zugeordneten Objekts (`this`);
  - dürfen Klassen- und Instanz-Methoden aufrufen.
- Klassen-Methoden
  - haben nur Zugriff auf die Klassen-Variablen
  - dürfen nur Klassen-Methoden aufrufen
- Instanzvariable und -methoden sind nur nach Instantiierung eines Objekts mittels <*Objekt*>.<*Variable*> bzw. <*Objekt*>.<*Methode*> erreichbar

Klassen-Variable: Beispiel

- Beispiel für eine Klassenvariable:

  `System.out.println("Hello World");`

  - **`System`** ist eine vom Java-System bereit gestellte Klasse.
  - **`out`** ist eine Klassen-Variable der Klasse **`System`**.
  - **`System.out`** bezeichnet ein Objekt (der Klasse **`PrintStream`**).
  - **`println()`** ist ein Merkmal (eine Instanz-Methode) von Objekten der Klasse **`PrintStream`**.

Zum Begriff `static`

- Ist eine Variable in einer Java-Klasse statisch (**`static`**) deklariert, dann gibt es diese Variable nur einmal, egal wie viele Objekte von dieser Klasse existieren.
  - Der Name statisch bezieht sich auf die Bereitstellung des nötigen Speicherplatzes, die für Klassenvariablen statisch (zur Übersetzungszeit) geschehen kann, und nicht dynamisch (zur Laufzeit), wenn Objekte instantiiert werden.
  - Als statisch deklarierte Variablen haben nichts "Statisches", ihre Werte können während der Laufzeit variieren, genauso wie die Werte anderer Variablen (Instanzvariablen, lokale Variablen).

Weiteres Vorgehen

- Die Methoden, die wir in **`OrderedIntList`** noch bereit stellen müssen, sind:
  - **`isElement()`** für die Methode **`contains()`**
  - **`insertElement()`** für die Methode **`insert()`**
  - **`deleteElement()`** für die Methode **`delete()`**
  - **`copyList()`** für den Copy-Konstruktor
  - **`getItem()`** und **`getNext()`** für die Methode **`isSubset()`**
  - **`toString()`** für die Methode **`toString()`**
- Wegen der geeigneten Behandlung der leeren Liste werden wir die ersten vier dieser Methoden wieder als **Klassen-Methoden** realisieren.

Die Klassen-Methode `isElement()` für die Klasse `OrderedIntList`

```
// Test, ob ein Element in sortierter Liste enthalten ist:
public static boolean isElement (int i, OrderedIntList l) {

   // Falls die Liste leer ist oder nur groessere Elemente enthaelt:
   if (l == null || l.item > i)
      return false;

   // Falls das erste Listenelement i enthaelt:
   if (l.item == i)
      return true;

   // Ansonsten arbeite rekursiv mit der Nachfolger-Liste:
   return isElement(i, l.next);
}
```

- Man beachte wieder, wie „elegant“ sich die rekursive Methode an die rekursive Datenstruktur „anlehnt“!

Die Klassen-Methode `insertElement()` *für die Klasse* `OrderedIntList`

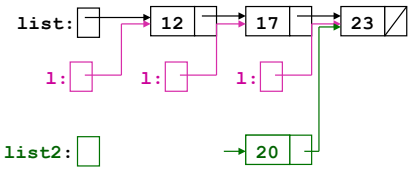
```
// Einfuegen eines Elementes in sortierte Liste:
public static OrderedIntList insertElement (int i, OrderedIntList l) {

   // Falls die Liste leer ist oder nur groessere Elemente enthaelt:
   if (l == null || l.item > i)
      return new OrderedIntList(i, l);

   // Falls das erste Listenelement i enthaelt:
   if (l.item == i) {
      System.out.println("insertElement: " + i + " schon vorhanden.");
      return l;  // l wird unveraendert zurueckgeliefert
   }

   // Ansonsten arbeite rekursiv mit der
   // Nachfolger-Liste:
   l.next = insertElement(i, l.next);
   return l;
}
```

Da einer Klassenmethode kein Objekt zugeordnet ist, auf der sie direkt operiert, muss das gewünschte Objekt als Parameter übergeben bzw. als Ergebnis abgeliefert werden!

Wirkungsweise der Methode `insertElement()`

```
public static OrderedIntList insertElement (int i, OrderedIntList l) {
  if (l == null || l.item > i)
      return new OrderedIntList(i, l);
  if (l.item == i) {
      System.out.println("insertElement: " + i + " schon vorhanden.");
      return l;
   }
   l.next = insertElement(i, l.next);
   return l;
}
```

- Gegeben sei wieder eine Liste:



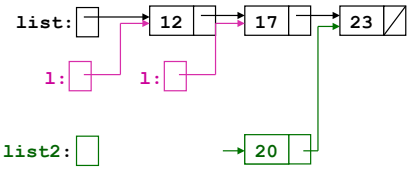
- Dann ist dies die Wirkungsweise der Anweisung

```
OrderedIntList list2 = OrderedIntList.insertElement(20,list);
```


Wirkungsweise der Methode `insertElement()`

```
public static OrderedIntList insertElement (int i, OrderedIntList l) {
  if (l == null || l.item > i)
      return new OrderedIntList(i, l);
  if (l.item == i) {
      System.out.println("insertElement: " + i + " schon vorhanden.");
      return l;
   }
   l.next = insertElement(i, l.next);
   return l;
}
```

- Gegeben sei wieder eine Liste:



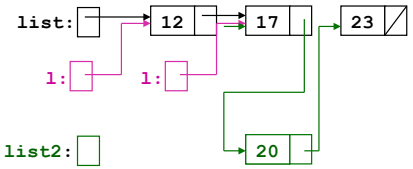
- Dann ist dies die Wirkungsweise der Anweisung

```
OrderedIntList list2 = OrderedIntList.insertElement(20,list);
```


Wirkungsweise der Methode `insertElement()`

```
public static OrderedIntList insertElement (int i, OrderedIntList l) {
  if (l == null || l.item > i)
      return new OrderedIntList(i, l);
  if (l.item == i) {
      System.out.println("insertElement: " + i + " schon vorhanden.");
      return l;
   }
   l.next = insertElement(i, l.next);
   return l;
}
```

- Gegeben sei wieder eine Liste:



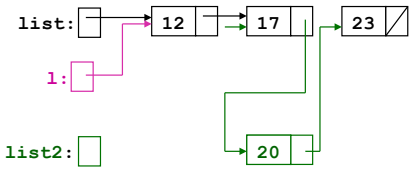
- Dann ist dies die Wirkungsweise der Anweisung

```
OrderedIntList list2 = OrderedIntList.insertElement(20,list);
```

Wirkungsweise der Methode `insertElement()`

```
public static OrderedIntList insertElement (int i, OrderedIntList l) {
  if (l == null || l.item > i)
      return new OrderedIntList(i, l);
  if (l.item == i) {
      System.out.println("insertElement: " + i + " schon vorhanden.");
      return l;
   }
   l.next = insertElement(i, l.next);
   return l;
}
```

❖ Gegeben sei wieder eine Liste:



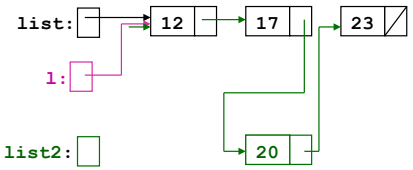
❖ Dann ist dies die Wirkungsweise der Anweisung

```
OrderedIntList list2 = OrderedIntList.insertElement(20,list);
```


Wirkungsweise der Methode `insertElement()`

```
public static OrderedIntList insertElement (int i, OrderedIntList l) {
  if (l == null || l.item > i)
      return new OrderedIntList(i, l);
  if (l.item == i) {
      System.out.println("insertElement: " + i + " schon vorhanden.");
      return l;
   }
   l.next = insertElement(i, l.next);
   return l;
}
```

❖ Gegeben sei wieder eine Liste:



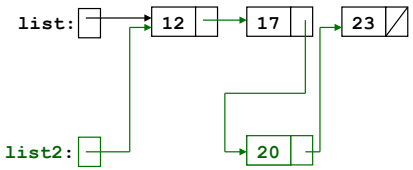
❖ Dann ist dies die Wirkungsweise der Anweisung

```
OrderedIntList list2 = OrderedIntList.insertElement(20,list);
```


Wirkungsweise der Methode `insertElement()`

```
public static OrderedIntList insertElement (int i, OrderedIntList l) {
  if (l == null || l.item > i)
      return new OrderedIntList(i, l);
  if (l.item == i) {
      System.out.println("insertElement: " + i + " schon vorhanden.");
      return l;
   }
   l.next = insertElement(i, l.next);
   return l;
}
```

❖ Gegeben sei wieder eine Liste:



❖ Dann ist dies die Wirkungsweise der Anweisung

```
OrderedIntList list2 = OrderedIntList.insertElement(20,list);
```

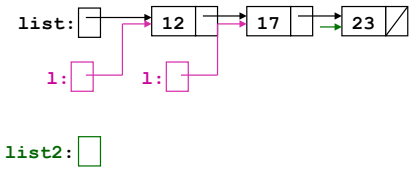

Die Klassen-Methode `deleteElement()` *für die Klasse* `OrderedIntList`

```
// Loeschen eines Elements aus sortierter Liste:
public static OrderedIntList deleteElement (int i, OrderedIntList l) {

   // Falls die Liste leer ist oder nur groessere Elemente enthaelt:
   if (l == null || l.item > i) {
      System.out.println("deleteElement: " + i + " nicht vorhanden.");
      return l;        // l wird unveraendert zurueckgeliefert
   }

   // Falls das erste Listenelement i enthaelt:
   if (l.item == i) {
      return l.next;  // hier wird i durch "Umleitung" gelöscht
   }

   // Ansonsten arbeite rekursiv mit der Nachfolger-Liste:
   l.next = deleteElement(i, l.next);
   return l;
}
```

Wirkungsweise der Methode `deleteElement()`

```
public static OrderedIntList deleteElement (int i, OrderedIntList l) {
   if (l == null || l.item > i) {
      System.out.println("deleteElement: " + i + " nicht vorhanden.");
      return l;
   }
   if (l.item == i) {
      return l.next;  // hier wird i durch "Umleitung" gelöscht
   }
   l.next = deleteElement(i, l.next);
   return l;
}
```

❖ Gegeben sei wieder eine Liste:



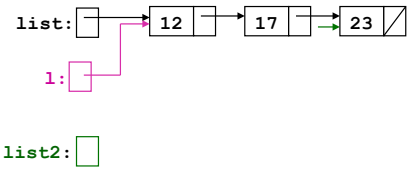
❖ Dann ist dies die Wirkungsweise der Anweisung

```
OrderedIntList list2 = OrderedIntList.deleteElement(17,list);
```


Wirkungsweise der Methode `deleteElement()`

```
public static OrderedIntList deleteElement (int i, OrderedIntList l) {
   if (l == null || l.item > i) {
      System.out.println("deleteElement: " + i + " nicht vorhanden.");
      return l;
   }
   if (l.item == i) {
      return l.next;  // hier wird i durch "Umleitung" gelöscht
   }
   l.next = deleteElement(i, l.next);
   return l;
}
```

❖ Gegeben sei wieder eine Liste:



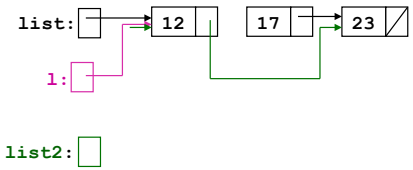
❖ Dann ist dies die Wirkungsweise der Anweisung

```
OrderedIntList list2 = OrderedIntList.deleteElement(17,list);
```


Wirkungsweise der Methode `deleteElement()`

```
public static OrderedIntList deleteElement (int i, OrderedIntList l) {
   if (l == null || l.item > i) {
      System.out.println("deleteElement: " + i + " nicht vorhanden.");
      return l;
   }
   if (l.item == i) {
      return l.next;  // hier wird i durch "Umleitung" gelöscht
   }
   l.next = deleteElement(i, l.next);
   return l;
}
```

❖ Gegeben sei wieder eine Liste:



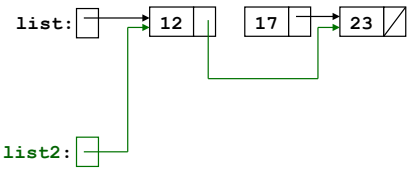
❖ Dann ist dies die Wirkungsweise der Anweisung

```
OrderedIntList list2 = OrderedIntList.deleteElement(17,list);
```


Wirkungsweise der Methode `deleteElement()`

```
public static OrderedIntList deleteElement (int i, OrderedIntList l) {
   if (l == null || l.item > i) {
      System.out.println("deleteElement: " + i + " nicht vorhanden.");
      return l;
   }
   if (l.item == i) {
      return l.next;  // hier wird i durch "Umleitung" gelöscht
   }
   l.next = deleteElement(i, l.next);
   return l;
}
```

❖ Gegeben sei wieder eine Liste:



❖ Dann ist dies die Wirkungsweise der Anweisung

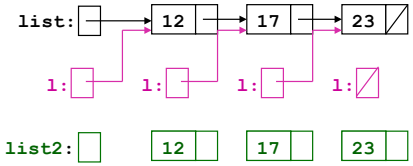
```
OrderedIntList list2 = OrderedIntList.deleteElement(17,list);
```

Die Klassen-Methode `copyList()` *für die Klasse* `OrderedIntList`

```
// Liefert Kopie einer sortierten Liste:
   public static OrderedIntList copyList (OrderedIntList l) {
      if (l == null)
         return null;
      return new OrderedIntList(l.item, copyList(l.next));
   }
```

Wirkungsweise der Methode `copyList()`

❖ Gegeben sei wieder eine Liste:



❖ Dann ist dies die Wirkungsweise der Anweisung

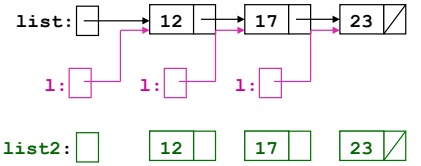
```
OrderedIntList list2 = OrderedIntList.copyList(list);
```


Die Klassen-Methode `copyList()` *für die Klasse* `OrderedIntList`

```
// Liefert Kopie einer sortierten Liste:
   public static OrderedIntList copyList (OrderedIntList l) {
      if (l == null)
         return null;
      return new OrderedIntList(l.item, copyList(l.next));
   }
```

Wirkungsweise der Methode `copyList()`

❖ Gegeben sei wieder eine Liste:



❖ Dann ist dies die Wirkungsweise der Anweisung

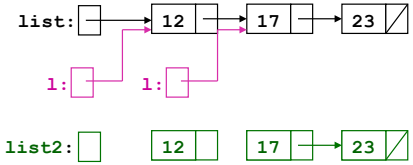
```
OrderedIntList list2 = OrderedIntList.copyList(list);
```


Die Klassen-Methode `copyList()` *für die Klasse* `OrderedIntList`

```
// Liefert Kopie einer sortierten Liste:
   public static OrderedIntList copyList (OrderedIntList l) {
      if (l == null)
         return null;
      return new OrderedIntList(l.item, copyList(l.next));
   }
```

Wirkungsweise der Methode `copyList()`

❖ Gegeben sei wieder eine Liste:



❖ Dann ist dies die Wirkungsweise der Anweisung

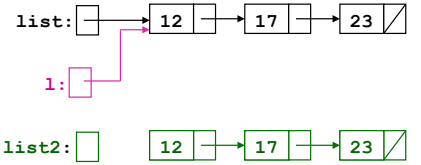
```
OrderedIntList list2 = OrderedIntList.copyList(list);
```


Die Klassen-Methode `copyList()` *für die Klasse* `OrderedIntList`

```
// Liefert Kopie einer sortierten Liste:
   public static OrderedIntList copyList (OrderedIntList l) {
      if (l == null)
         return null;
      return new OrderedIntList(l.item, copyList(l.next));
   }
```

Wirkungsweise der Methode `copyList()`

❖ Gegeben sei wieder eine Liste:



❖ Dann ist dies die Wirkungsweise der Anweisung

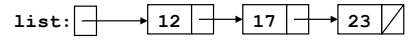
```
OrderedIntList list2 = OrderedIntList.copyList(list);
```

Die Klassen-Methode `copyList()` *für die Klasse* `OrderedIntList`

```
// Liefert Kopie einer sortierten Liste:
   public static OrderedIntList copyList (OrderedIntList l) {
      if (l == null)
         return null;
      return new OrderedIntList(l.item, copyList(l.next));
   }
```

Wirkungsweise der Methode `copyList()`

- Gegeben sei wieder eine Liste:



- Dann ist dies die Wirkungsweise der Anweisung

```
OrderedIntList list2 = OrderedIntList.copyList(list);
```

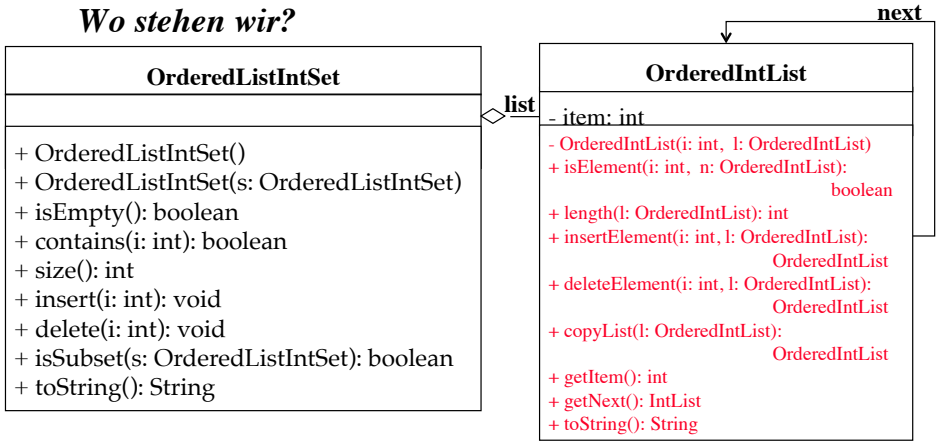

Die Instanz-Methoden `getItem()`, `getNext()` *und* `toString()` *für die Klasse* `OrderedIntList`

```
   // liefert Inhalt des Listenelements
   public int getItem() {
      return item;
   }

   // liefert Verweis auf Nachfolger des Listenelements
   public OrderedIntList getNext() {
      return next;
   }

   // Ausgabefunktion:
   public String toString() {
      if (next == null)
         return "" + item;
      return item + "," + next;
   }
```

Dies ist eine „Kunstgriff“, um den Compiler zu überzeugen, dass dies ein Ausdruck vom Typ **`String`** ist, **`item`** also nach **`String`** zu wandeln ist!

Hier wird **`next`** zu **`String`** gewandelt. Implizit wird also **`toString()`** rekursiv aufgerufen!

Wo stehen wir?



- Wir haben die Klasse **`OrderedIntList`** vollständig implementiert.
- Es fehlt noch die Klasse **`OrderedListIntSet`**.
- Bei der Realisierung von **`OrderedListIntArray`** können wir uns jedoch weitgehend auf **`OrderedIntList`** abstützen.
  - Ausname: Methode **`isSubset()`**)

Die Klasse `OrderedListIntSet` *für Mengen über sortierten linearen Listen*

```
class OrderedListIntSet {

// Attribute (Datenstruktur):
    private OrderedIntList list; // speichert die Elemente der Menge

// Konstruktoren:

   // Konstruktor fuer eine leere Menge:
    OrderedListIntSet() {
       list = null;
    }
   // Konstruktor, der die Kopie einer Menge liefert:
    public OrderedListIntSet(OrderedListIntSet s) {
       // stützt sich auf entsprechende Methode von OrderedIntList:
       list = OrderedIntList.copyList(s.list);
    }

// Sonstige Methoden:
   ...
}
```

Aufruf einer Klassen-Methode innerhalb eines Konstruktors

Die Methoden `isEmpty()`, `contains()` *und* `size()` *für die Klasse* `OrderedListIntSet`

- Die Methode **`isempty()`** stützt sich genau wie in den bisherigen Implementierungen auf die Instanz-Methode **`size()`**:

```
// Abfrage, ob Menge leer ist:
public boolean isEmpty() {
   return size() == 0;
}
```

- Die Methoden **`contains()`** und **`size()`** stützen sich auf die entsprechenden Klassen-Methoden **`isElement()`** und **`length()`** der Klasse **`OrderedIntList`**:

```
// Abfrage, ob Element enthalten ist:
public boolean contains(int i) {
   return OrderedIntList.isElement(i, list);
}
```

```
// Abfrage nach Groesse der Menge:
public int size() {
   return OrderedIntList.length(list);
}
```

Aufruf von Klassen-Methoden innerhalb von Instanz-Methoden

Die Methoden `insert()`, `delete()` *und* `toString()` *für die Klasse* `OrderedListIntSet`

- Die Methoden **`insert()`** und **`delete()`** stützen sich wieder auf die entsprechenden Methoden der Klasse **`OrderedIntList`**:

```
// Einfuegen eines Elementes:
public void insert(int i) {
   list = OrderedIntList.insertElement(i, list);
}
```

```
// Entfernen eines Elementes:
public void delete(int i) {
   list = OrderedIntList.deleteElement(i, list);
}
```

- Die Methode **`toString()`** muss nur noch für die Mengen-Klammern sorgen:

```
public String toString() {
   String result = "{";
   if (list != null)
      result += list;
   return result + "}";
}
```


`isSubset()`*: von sortierter Reihung zu sortierter Liste*

```
public boolean isSubset(OrderedListIntSet s) {
  OrderedIntList l = list;      // Pegel der Menge selbst
  OrderedIntList lS = s.list;   // Pegel der anderen Menge s
  while (l != null && lS != null) {
    if (l.getItem() < lS.getItem())
            // Element der Menge kann nicht auch in s sein
      return false;
    if (l.getItem() > lS.getItem())
            // s weiterschalten
      lS = lS.getNext();
    else {
            // Element der Menge ist auch in s; beide
            // Pegel weiterschalten:
      l = l.getNext();
      lS = lS.getNext();
    }
  }
     // Teilmengeneigenschaft ist genau dann erfuellt, wenn
     // l die gesamte Menge durchlaufen hat:
  return l == null;
}
```


Zusammenfassung: Listen

- Das Referenzkonzept von Java erlaubt es, rekursive Datenstrukturen über Verweise zu realisieren.
  - Ein Objekt vom Typ **`List`** enthält in einem Attribut einen Verweis auf ein Objekt vom selben Typ.
- Der besondere **`null`**-Verweis erlaubt es auszudrücken, dass eine Referenzvariable **nicht** auf ein Objekt verweist.
- Listen haben gegenüber Reihungen den Vorteil, dass ohne Verschieben an beliebiger Stelle ein neues Element eingefügt bzw. entfernt werden kann.
  - Dieses Mehr an Flexibilität wird durch ein Mehr an Speicherbedarf für den **`next`**-Verweis „erkauft“.
  - Ein Zugriff über den Index ist nicht mehr möglich.
- Für verschiedene Anwendungsfälle können Listen auf verschiedene Arten realisiert werden
  - einfach/doppelt verkettet, ein/zwei Anker, sortiert
- Das Programmieren mit Verweisstrukturen ist fehleranfällig und erfordert deshalb hohe Sorgfalt
  - z.B. Vermeidung von Zyklen in linearen Listen

Zusammenfassung: Reihungen und Listen

- Reihungen machen es möglich, eine Menge von Variablen desselben Typs über ihre Indexposition zu identifizieren
  - Mit einer **`for`**-Schleife kann damit eine beliebige Anzahl von Variablen „durchlaufen“ werden.
- Das Referenzkonzept erlaubt den Aufbau flexibler Datenstrukturen (Geflechte, z.B. Listen).
- Am Programmierbeispiel der Mengendarstellung wird deutlich, dass die Erfüllung nichtfunktionaler Anforderungen (z.B. Effizienz) entscheidend von der Wahl der Datenstruktur beeinflusst wird.