Übung 11

- Fragen ?
- Nächste Woche ist letzte Übung - Fragestunde

Übung 9

- Neues Basic.java wird benötigt !!!

Siehe Folien - Übung 8

- Ein package für die TicTacToe Klassen muß erstellt werden
  - -> Basic muß in diesen importiert werden:
    - import Basic;
  - -> Basic.java soll im Wurzelverzeichnis des packages liegen:
    - wenn c:\xyz\swe1\tictactoe -> Basic.java liegt in c:\xyz
  - -> Übersetzen in c:\xyz
    - javac swe1\tictactoe*.java
    - javac Basic.java
  - -> Exekution in c:\xyz
    - java swe1.tictactoe.TicTacToe

Übung 9 / Aufgabe 1d

- Idee ohne Vererbung:

```
class Player extends Basic {
   // available types of players
   final static int USER = 0, RANDOM = 1, COMPUTER_SIMPLE = 2;

   int type;      // type of player
   String name;   // name of player

   Player (int playerType, char playerName) {
      type = playerType; name = playerName;
   }

   boolean doMove (Field field) {
      switch type {
         case USER: ..... break;
         case RANDOM: ..... break;
         case COMPUTER_SIMPLE: .... break;
         default: .... break;
      }
}
```

Übung 9 / Aufgabe 1d

- Idee mit Vererbung (Bonuspunkte):

```
abstract class Player extends Basic {
   abstract void doMove(Field field);
}

class RandomPlayer extends Player {
   doMove(Field field) { ... make any legal move ...}
}

class HumanPlayer extends Player {
   doMove(Field field) { ... Interaction with human player ...}
}

class SimplePlayer extends RandomPlayer {
   doMove(Field field) {
       if opponent can win, prevent loss
       else super.doMove()  // RandomPlayer
   }
}
```

- Im Forum gibt es schon sehr viele Tips!

Rekursion - Kontrollfluß

Ralf Hauber:
1 für n=0
Undef. für n<0

✸ Fibonacci

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
     return fib(n-1) + fib(n-2);
}
```

$$fib(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ fib(n-1) + fib(n-2) & \text{für } n > 1 \end{cases}$$

```
fib(3)
```

Rekursion - Kontrollfluß

✸ Fibonacci

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
     return fib(n-1) + fib(n-2);
}
```

$$\text{fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{für } n > 1 \end{cases}$$

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
    return fib(n-1) + fib(n-2);
}
```

n=3

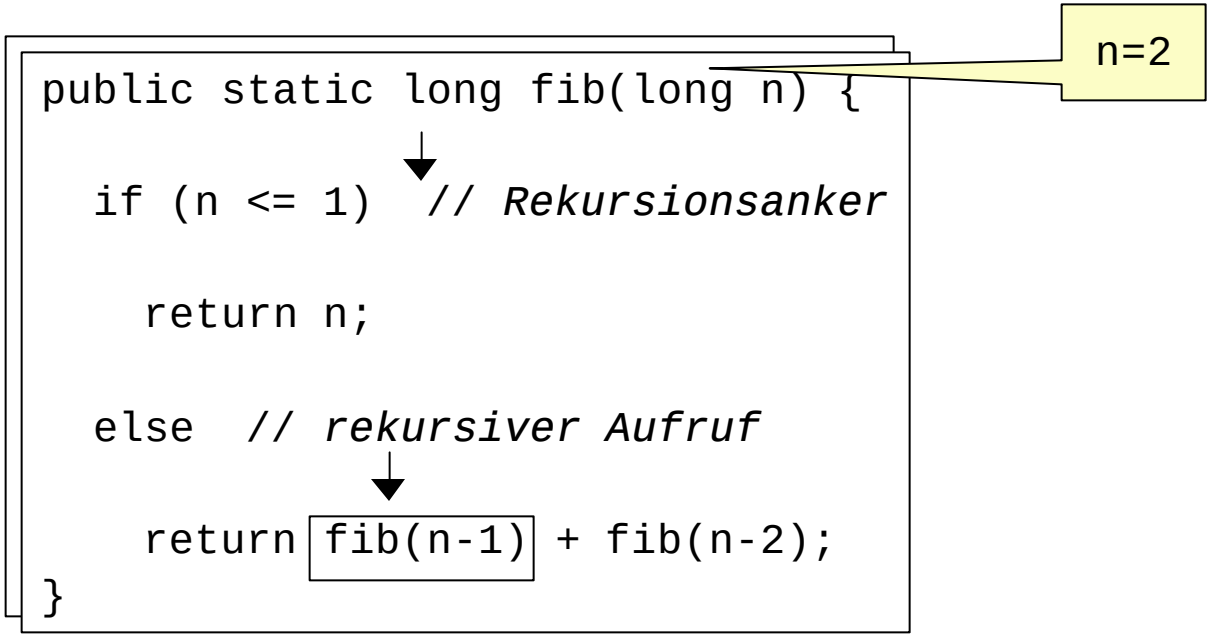
Ausführung von links nach rechts

Rekursion - Kontrollfluß

✸ Fibonacci

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
     return fib(n-1) + fib(n-2);
}
```

$$\text{fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{für } n > 1 \end{cases}$$

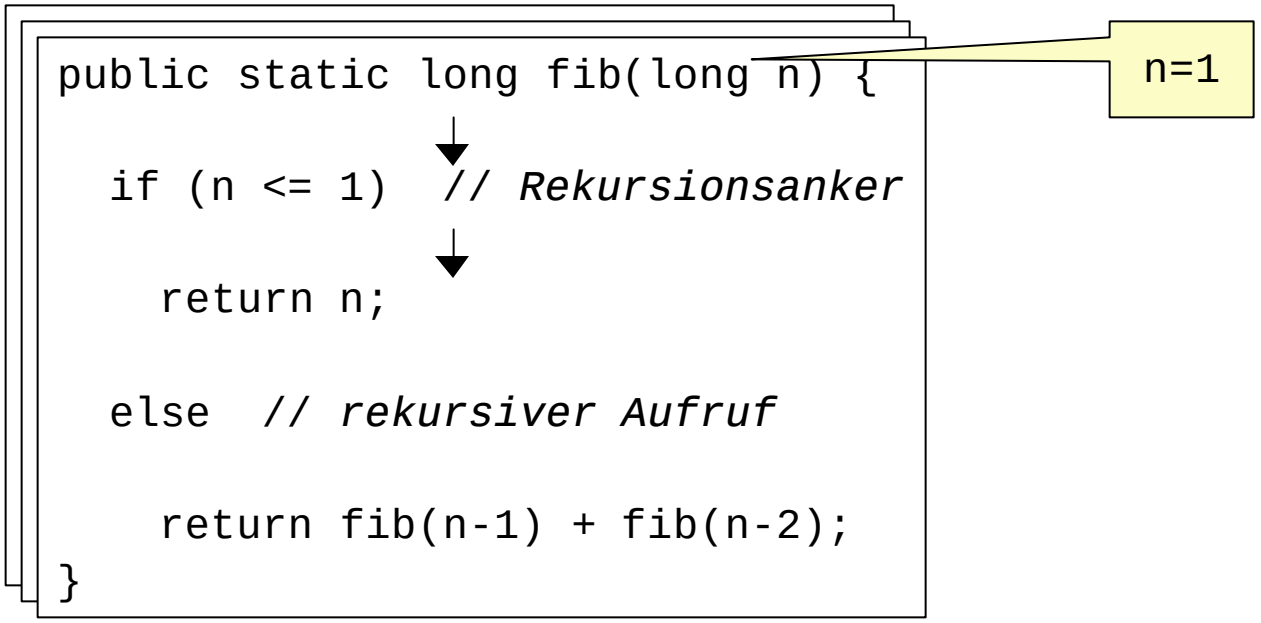


Rekursion - Kontrollfluß

✸ Fibonacci

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
     return fib(n-1) + fib(n-2);
}
```

$$\text{fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{für } n > 1 \end{cases}$$

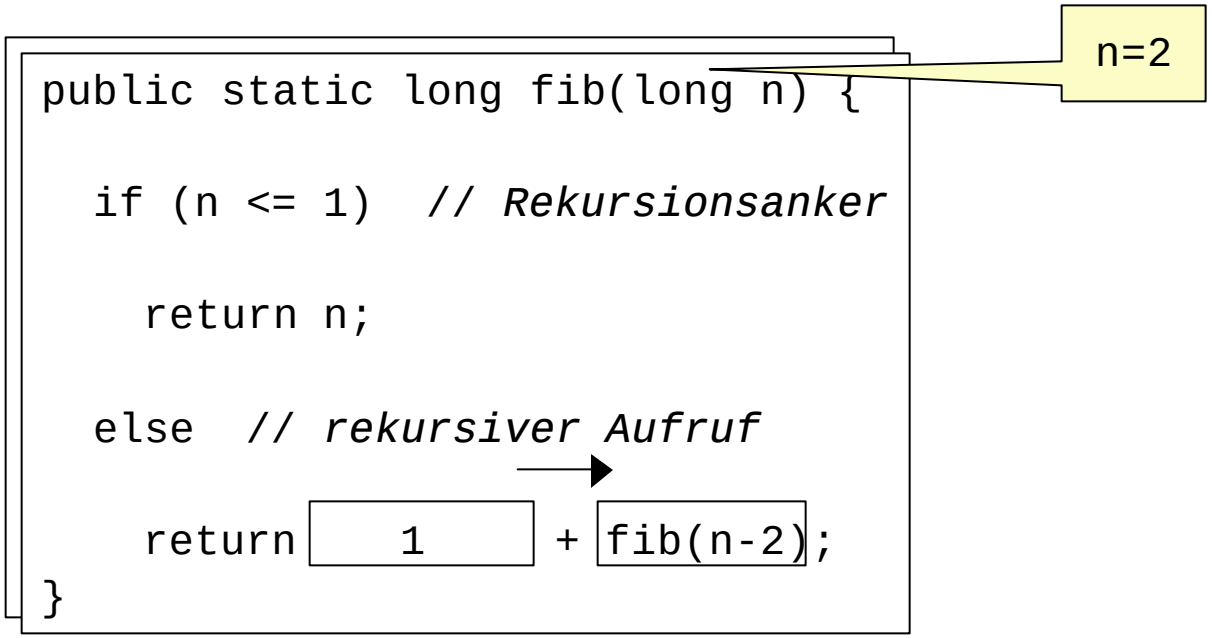


Rekursion - Kontrollfluß

✸ Fibonacci

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
     return fib(n-1) + fib(n-2);
}
```

$$\text{fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{für } n > 1 \end{cases}$$

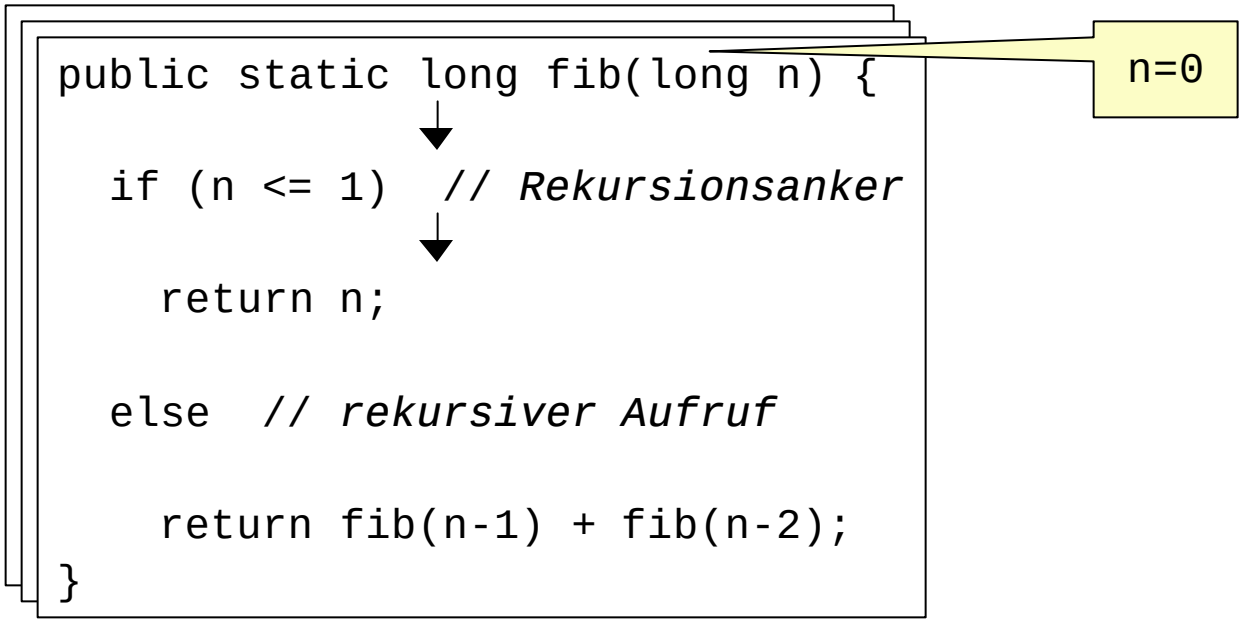


Rekursion - Kontrollfluß

✸ Fibonacci

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
     return fib(n-1) + fib(n-2);
}
```

$$\text{fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{für } n > 1 \end{cases}$$



Rekursion - Kontrollfluß

✸ Fibonacci

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
     return fib(n-1) + fib(n-2);
}
```

$$\text{fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{für } n > 1 \end{cases}$$

```
public static long fib(long n) {

  if (n <= 1)  // Rekursionsanker

    return n;

  else  // rekursiver Aufruf

    return [ 1 ] + [ 0 ];
}
```

Rekursion - Kontrollfluß

✸ Fibonacci

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
     return fib(n-1) + fib(n-2);
}
```

$$\text{fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{für } n > 1 \end{cases}$$

```
public static long fib(long n) {

  if (n <= 1)  // Rekursionsanker

    return n;

  else  // rekursiver Aufruf

    return    1    + fib(n-2);
}
```

n=3

Rekursion - Kontrollfluß

✸ Fibonacci

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
     return fib(n-1) + fib(n-2);
}
```

$$\text{fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{für } n > 1 \end{cases}$$

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
    return fib(n-1) + fib(n-2);
}
```

n=1

Rekursion - Kontrollfluß

✸ Fibonacci

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
     return fib(n-1) + fib(n-2);
}
```

$$\text{fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{für } n > 1 \end{cases}$$

```
public static long fib(long n) {

  if (n <= 1)  // Rekursionsanker

    return n;

  else  // rekursiver Aufruf

    return [   1   ] + [   1   ];
}
```

Rekursion - Kontrollfluß

✸ Fibonacci

```
public static long fib(long n) {
  if (n <= 1)  // Rekursionsanker
    return n;
  else  // rekursiver Aufruf
     return fib(n-1) + fib(n-2);
}
```

$$fib(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ fib(n-1) + fib(n-2) & \text{für } n > 1 \end{cases}$$

2

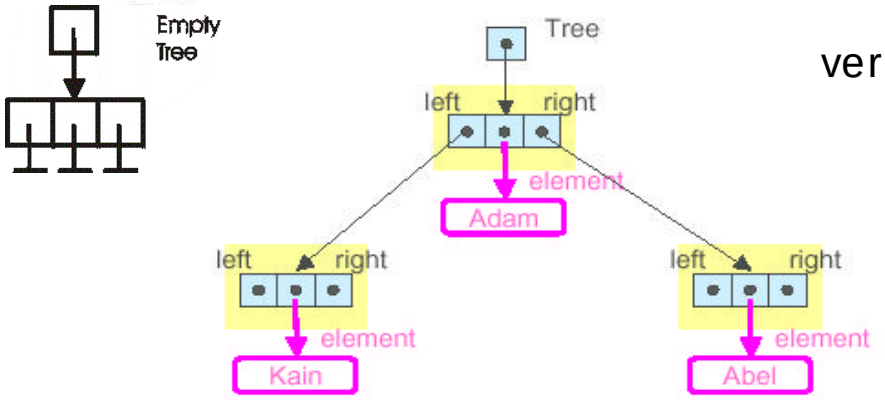
Wiederholung: Dynamische Datenstrukturen

- Um größere Anzahl von Datensätzen zu verwalten
- Dazu gibt's doch Arrays! Geht schon, ABER:
  - Größe muss vor ihrer Verwendung fest stehen
    - Zu klein, Inhalt muß zur Laufzeit kopiert werden
    - Zu groß, Speicher wird verschwendet
  - Einfügen & Entfernen erfordert Kopieren aller nachfolgenden Elemente
- Existieren in großer Formenvielfalt
  - Listen (einfach, doppelt, zyklisch)
  - Baum (natürlich, binär, balanciert)
  - Graphen
- Prinzip: **Verkettung**
  - Knoten beinhaltet neben „Daten" Verweis(e) zum nächsten Knoten
  - Beispiel:

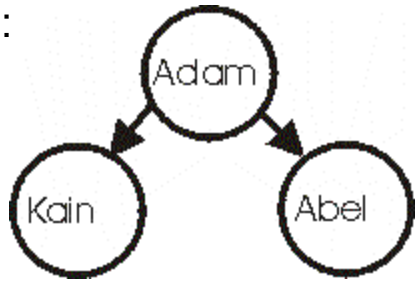
```
class Tree {
   Object element;
   Tree left, right;      // next Items
}
```

Binärer Baum

- Ein Binärbaum ist entweder leer oder er besteht aus einem Knoten, dem ein Element und zwei binäre Bäume zugeordnet sind (rekursive Definition).



vereinfacht:

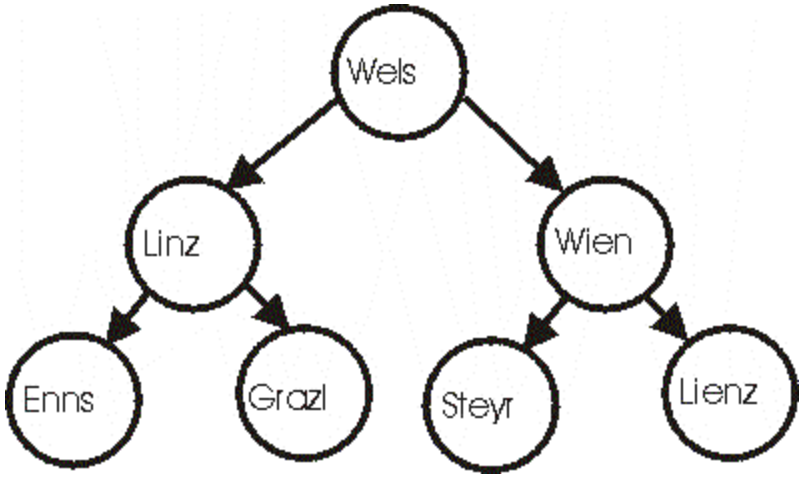


- Operationen (Auswahl):
  - empty () liefert Wahrheitswert true, wenn Baum leer ist, false sonst
  - left () liefert Verweis auf linken Teilbaum
  - right () liefert Verweis auf rechten Teilbaum
  - value () liefert Verweis auf Wurzelelement

Binärer Baum

- Rekursive Traversierung
  - Preorder (Wurzel, links, rechts)
  - Inorder (links, Wurzel, rechts)
  - Postorder (links, rechts, Wurzel)

- Beispiel



- Preorder: Wels, Linz, Enns, Graz, Wien, Steyr, Lienz
- Inorder: Enns, Linz, Graz, Wels, Steyr, Wien, Lienz
- Postorder: Enns, Graz, Linz, Steyr, Lienz, Wien, Wels

Selbsttest

Binärer Baum

```
public class Tree {
   Object element;
   Tree left, right;

   public Tree left() { .... }   // siehe VL-Skript
   public Tree right() { ... }
   public Object value() { ... }
}


public static void printPreOrder(Tree tree) {
   if (!tree.empty()) {
      writeLn(tree.value().toString());
      printPreOrder(tree.left());
      printPreOrder(tree.right());
   }
}
```

Binärer Baum

```
public static void printInOrder(Tree tree) {
   if (!tree.empty()) {
      printInOrder(tree.left());
      writeLn(tree.value().toString());
      printInOrder(tree.right());
   }
}

public static void printPostOrder(Tree tree) {
   if (!tree.empty()) {
      printPostOrder(tree.left());
      printPostOrder(tree.right());
      writeLn(tree.value().toString());
   }
}
```

- Vorgehensweise immer gleich (abgesehen von Reihenfolge)
  - rufe Methode rekursiv für linken und rechten Teilbaum auf
  - bearbeite Element

Wiederholung: Typcast

- Viele Methoden erwarten als Parameter Objekte der Klasse `Object` bzw. liefern solche (siehe Container-Beispiel). Möglich wegen Polymorphismus!
- Typcast nötig um wieder die spezialisierte „Sicht“ auf das Objekt zu bekommen.
- Beispiel

```
Container c = new LinkedList();
Person p = new Person();

c.add(p);
Person r = (Person)c.first();
```

Der Programmierer „weiß“, daß hier wieder eine Person geliefert wird.... doch wie sagt man‘s dem Compiler? (zur Erinnerung: `first()` liefert `Object`)

- sicherer:

```
Object o = c.first();
Person r = null;
if (o instanceof Person)
   r = (Person)o;

// Achtung r kann hier u.U. null sein !
```

Wrapper für primitive Datentypen

- Fast alles ist ein Objekt: Primitive Datentypen (`int`, `boolean`, `double`, ...) sind keine.

- Um sie wie Objekte verwenden zu können, gibt es Wrapper-Klassen, die die Daten in Objekte verpacken.
  - Die Klassen heißen wie die Datentypen, nur ausgeschrieben und mit großem Anfangsbuchstaben.

```
Integer wrappedInt = new Integer( 3);
int value = wrappedInt.intValue();
```

  - Siehe JFC: `java.lang`.

„Generische“ „Schnittstelle“

- Möglichst allgemeine Schnittstelle
  - problemunabhängig
  - verschiedene Implementierungen z.B.: Liste, Baum

- Ein Container, in dem alles verwaltet werden kann:

```
public abstract class Container {
   abstract public boolean add (Object object);
   abstract public boolean delete (Object object);
   abstract public boolean contains (Object object);
   abstract public Object first();
   abstract public Object last();
   abstract public boolean isEmpty();
}
```

Verkette Liste (Ansatz)

- Listenelement:

```
public class ListNode {
   Object element;
   ListNode next;

   public ListNode(Object element, ListNode next) {
      this.element = element;
      this.next = next;
   }
}
```

- Liste

```
public class LinkedList extends Container {
   ListNode head = null;

   public boolean add (Object element) {
      head = new ListNode(element,head);
      return true;
   }
   ...
```

Verkette Liste

```
public Object first() {
   if (head != null)
      return head.element;
   else
      return null;
}

public boolean contains(Object object) {
   ListNode tmp = head;
   if (object == null) return false;
   while (tmp != null && !object.equals(tmp.element) {
      tmp.next = tmp;
   }
   return tmp!=null;
}
```

Binärer Baum

```
public class Tree extends Container {
   Object element = null;
   Tree left = null, right = null;
   Compare comp;

   public Tree (Compare comp) {this.comp = comp;}

   public boolean add(Object x) {
      if (x == null) return false;
      if (empty()){
         element = x;
         left = new Tree(comp);
         right = new Tree(comp);
      } else if (comp.compare(x,value()) == 0)
         element = x;
      else if (comp.compare(x,value()) < 0) {
         return left().add(x);
      } else {
         return right.add(x);
      }
      return true;
   }
```

Idee Prof. Ferscha,
siehe Vergleichsklasse

Binärer Baum

```
// In Sinn „das größte Element bezüglich
// Vergleichsoperation“.
public Object last() {
   if (!right().empty())
      return right().last();
   else
      return value();
}

public boolean contains(Object x) {
   if (empty() || (x == null))
      return false;
   else if (comp.compare(x,value()) == 0)
      return true;
   else if (comp.compare(x,value()) < 0)
      return left().contains(x);
   else
      return right().contains(x);
}
```

Vergleichsklasse

zurück

```
// Problemabhängig (be)schreibt der Programmierer
// in Subklassen die Vergleichsoperation
// zweier Objekte.
public abstract class Compare {
   // 0, wenn o1 gleich o2 (im Sinn des Programmierers)
   // negativer Wert, wenn o1 < o2
   // positiver Wert, wenn o1 > o2
   public int compare(Object o1, Object o2);
}

// Z.B. Vergleich von Personen anhand ihres Alters ...
// negativer Wert, o1 jünger als o2, ...
public class PersonCompare extends Compare {
   public int compare(Object o1, Object o2) {
      if (o1 instanceof Person && o2 instanceof Person) {
          return ((Person) o1).age - ((Person) o2).age;
      } else {
          // Fehlermeldung, besser ClassCastException
      }
   }
}
```