

# Bäume

---

Martin Wirsing

in Zusammenarbeit mit  
Matthias Hözl, Piotr Kosiuczenko, Dirk Pattinson

06/03

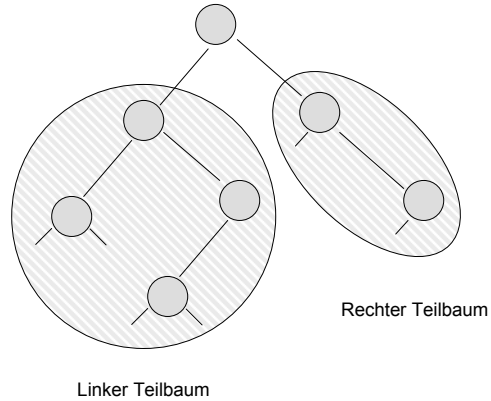
---

## Ziele

- Standardimplementierungen für Bäume kennenlernen
- Das Composite-Muster kennenlernen

## Bäume - 2-dimensionale Listen

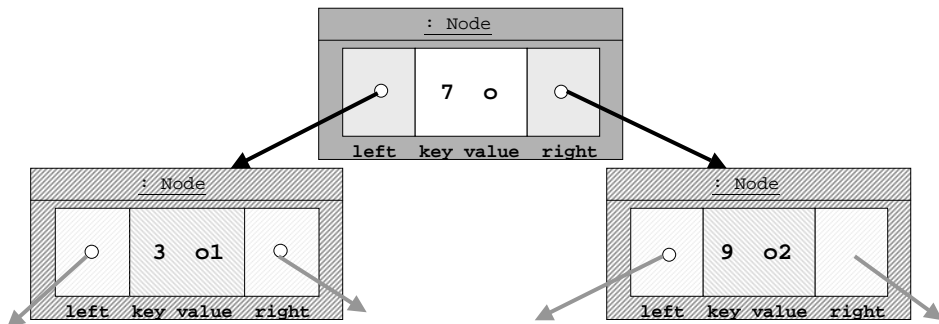
- Bäume sind hierarchische Strukturen
- Bäume bestehen aus
  - Knoten und
  - Teilbäumen
- Bei *Binärbäumen* hat jeder Knoten zwei Unterbäume:
  - den linken Unterbaum
  - den rechten Unterbaum
- In den Knoten kann Information gespeichert werden



## Baumknoten

```
class Node
{
    Node left;
    int key; Object value
    Node right; ...
}
```

- Wir implementieren einen Knoten als Objekt mit zwei Zeigern.
- Wir speichern **einen Schlüssel und ein Objekt** in den Knoten.



## BinTree

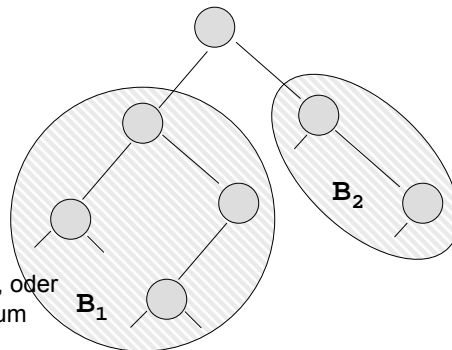
- Die Klasse `Node` ist nur eine Hilfsklasse für die Klasse `BinTree`

```
public class BinTree
{
    Node anchor; . . .
}
class Node
{
    Node left;
    int key; Object value;
    Node right;

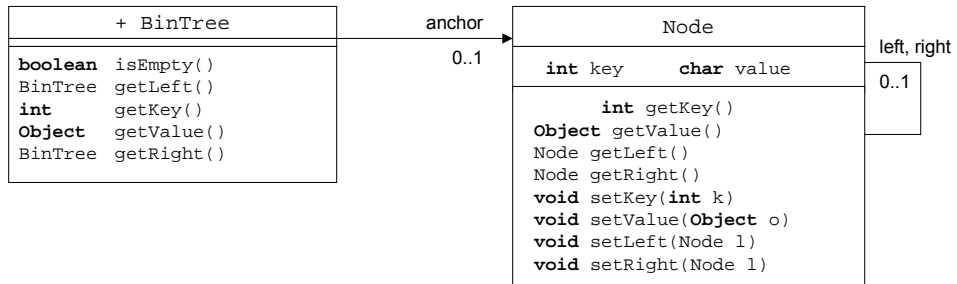
    // Konstruktor
    Node(Node b1, int k, Object o, Node b2)
    {
        left = b1; key = k; value = o; right = b2;
    }
    . . .
}
```

## Operationen auf BinTree

- Konstruktoren
  - `BinTree()`
    - der leere Baum
  - `BinTree( $B_1$ ,  $k$ ,  $o$ ,  $B_2$ )`
    - neuer `BinTree` mit
    - linkem Teilbaum  $B_1$
    - rechtem Teilbaum  $B_2$
    - Inhalt der Wurzel:  $k, o$
- Prädikat `isEmpty`
  - Testen, ob ein `BinTree` leer ist, oder einen rechten und linken Teilbaum enthält
- Selektoren `getLeft`, `getRight`, `getKey`
  - Falls der `BinTree` nicht leer ist, liefert
    - `getLeft()` den linken Teilbaum
    - `getRight()` den rechten Teilbaum
    - `getKey()` den Inhalt der Wurzel



## BinTree in UML



## Implementierung in Java

### Die Implementierung von BinTree verläuft analog zu Listen:

- BinTree repräsentiert Binärbäume über einem Integer-Schlüssel und Werten vom Typ Object.
  - BinTree selbst speichert nur einen Verweis auf die Wurzel des Baumes.
  - Die eigentliche Funktionalität wird von der Klasse Node realisiert;
- Um Funktionen auf BinTree zu definieren, verwenden wir folgende Fallunterscheidung:
  - leerer Baum: Berechnung des Resultats direkt in BinTree
  - nicht-leerer Baum: Weitergeben der Funktion an Node

## Implementierung BinTree: isEmpty & Zugriff auf linken Teilbaum

```

public class BinTree
{
    private Node anchor;

    BinTree(){}; // der leere Baum
    BinTree(BinTree b1, int k, Object o, BinTree b2)
    {
        anchor = new Node(b1.anchor, k, o, b2.anchor);
    }

    boolean isEmpty(){return anchor==null;}

    BinTree getLeft() throws NoSuchElementException
    {
        if (anchor == null) throw new NoSuchElementException();
        else
        {
            BinTree l = new BinTree();
            l.anchor = anchor.getLeft();
            return l;
        }
    }
}

```

...

- Implementierung verläuft analog zu Listen

## Implementierung BinTree: Summe der Knoten

```

int sumNodes()
{
    if (anchor == null) return 0;
    else return anchor.sumNodes();
}

where

class Node
{
    ...
    int sumNodes()
    {
        int suml = 0, sumr = 0;
        if (left != null) suml = left.sumNodes();
        if (right != null) sumr = right.sumNodes();
        return 1 + suml + sumr;
    }
}

```

...

## Geordnete Binärbäume

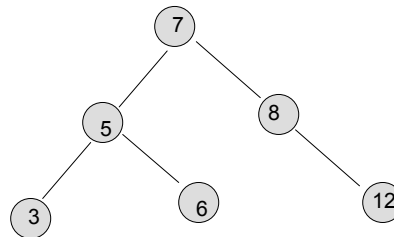
- Ein Binärbaum  $b$  heißt geordnet, wenn folgendes für alle nichtleeren Teilbäume  $t$  von  $b$  gilt:

Der Schlüssel von  $t$  ist

- größer (oder gleich) als alle Schlüssel des linken Teilbaums von  $t$  und
- kleiner (oder gleich) als alle Schlüssel des rechten Teilbaums von  $t$

- Beispiel: Geordnet sind:

Der leere Baum und der Baum:



## Suche im geordneten Binärbaum

```

public Object find(int key)
{ if (anchor == null) return null;
  else return anchor.find(key);
}

```

wobei

```

class Node
{ . . .
  Object find(int key)
  { Node current = this;
    while(current.key != key) // solange nicht gefunden,
    { if(key < current.key) // gehe nach links?
      current = current.left;
      else // oder gehe nach rechts?
        current = current.right;
    }
    if(current == null) return null; //nicht gefunden!
    return current.value; //gefunden; gib value zurück
  }
}

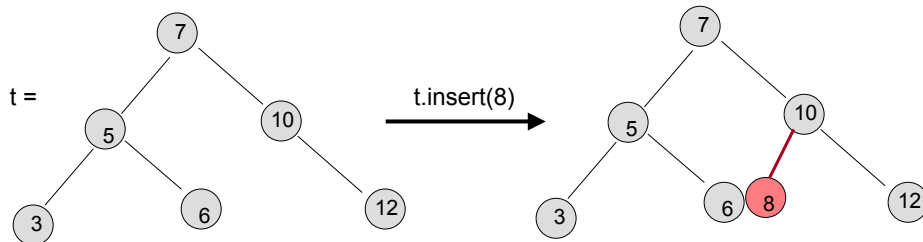
```

Gibt value zurück, wenn key im Baum; sonst wird null zurückgegeben

## Einfügen in geordneten Binärbaum

- Beim Einfügen in einen geordneten Binärbaum wird rekursiv die "richtige" Stelle gesucht, so daß wieder eine geordneter Binärbaum entsteht.

- Beispiel: `t.insert(8)` ergibt:



## Einfügen in geordneten Binärbaum

Fügt einen neuen Knoten mit Schlüssel id an der richtigen Stelle im geordneten Baum ein

```

public void insert(int id, Object o)
{
    if(anchor==null) // falls kein Knoten im anchor
        anchor = new Node(null,id,o,null); // neuer Knoten
    else anchor = anchor.insertKeyObj(id, o);
}
  
```

Wobei in `class Node`:

## Einfügen in geordneten Binärbaum

```

Node insertKeyObj(int id, Object o)
{
    Node current = this;    // starte bei this
    Node parent;
    while(true)            // terminiert intern
    {
        parent = current;
        if(id < current.key) // gehe nach links?
        {
            current = current.left;
            if(current == null) // am Ende füge links ein
            {
                parent.left = new Node(null, id, o, null);
                return this;
            }
        } // end if go left
        else // falls id > current.key, gehe nach rechts
        {
            current = current.right;
            if(current == null) // am Ende füge rechts ein
            {
                parent.right = new Node(null, id, o, null);
                return this;
            }
        } // end else gehe nach rechts
    } // end while
}

```

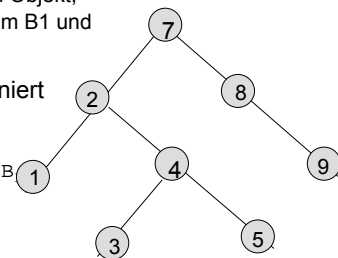
Fügt einen neuen Knoten passend ein  
**Achtung: id darf nicht im Baum vorkommen!**

## Rekursion auf Binärbäumen

- Binärbäume sind induktiv definiert
  - Der Leere Baum ist ein Binärbaum
  - Sind  $B_1$  und  $B_2$  Binärbäume, id ein Schlüssel und o ein Objekt, dann ist der Baum mit Wurzel id und o, linkem Teilbaum  $B_1$  und rechtem Teilbaum  $B_2$  ein Binärbaum

BinTree( $B_1, k, o, B_2$ )

- Operationen f auf Binärbäumen können rekursiv definiert werden (vgl. SML)
  - Falls isEmpty(B): gibt f(B) direkt an
  - Ansonsten beschreibe wie sich der Wert von f aus f( $B_1$ ), f( $B_2$ ) und id, o ergibt.



- Beispiel: sumNodes1

- Falls isEmpty(B): 0
- Ansonsten  
 getLeft().sumNodes1()+getRight().sumNodes1()+1

sumNodes: 8

- Beispiel: exist(int k) (ist k vorhanden?)

- Falls isEmpty(B): false
- Ansonsten: getLeft().exists(k)  
 || getKey()==k || getRight().exists(k)



## Einfache Baumoperationen

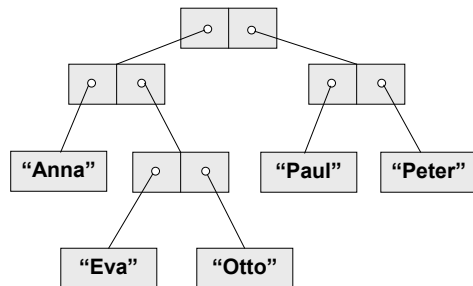
```
public class XBinTree extends BinTree
{
    public int sumNodes()
    {
        if(isEmpty()) return 0;
        else return
            ((XBinTree)getLeft()).sumNodes() +
            ((XBinTree)getRight()).sumNodes() +1;
    }

    public boolean exists(int k)
    {
        if(isEmpty()) return false;
        else return
            ((XBinTree)getLeft()).exists(k)
            || getKey()==k
            || ((XBinTree)getRight()).exists(k);
    }
}
```

- isEmpty(), left(), right() werden aus der Oberklasse geerbt
- left() liefert einen BinTree
- sumNodes() ist nur in der Unterklasse definiert - für XBinTrees
- Wir brauchen casts um die BinTrees in XBinTrees zu verwandeln

## Bäume mit Blättern

- Jeder Zweig soll in einem Blatt enden
- Die Information speichern wir in Blättern
- Jeder Knoten hat zwei Unterbäume



```
class Knoten
{
    Knoten links;
    Knoten rechts;
}
```

```
class Blatt
{
    String value;
}
```

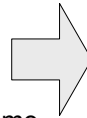
Wir haben ein Problem: Wir müssen auch zulassen:

```
Blatt links;
Blatt rechts;
```

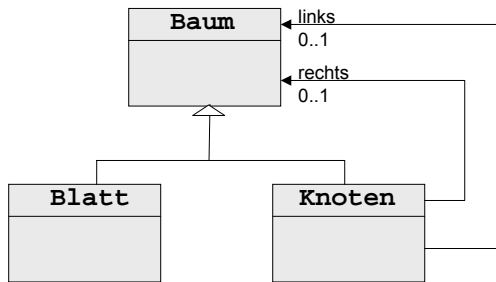
... aber ein Blatt ist kein Knoten!

## Baum in UML

- Wir wollen Blatt und Knoten zu einer Klasse Baum zusammenfassen.
- Blatt wird Unterklasse von Baum
- Knoten wird Unterklasse von Baum
- Viele Methoden müssen für alle Bäume funktionieren
  - `istBlatt()`
  - `istKnoten`
  - `sumNodes()`
  - `toString()`



```
class Blatt extends Baum
class Knoten extends Baum
```



## Default-Methoden redefinieren

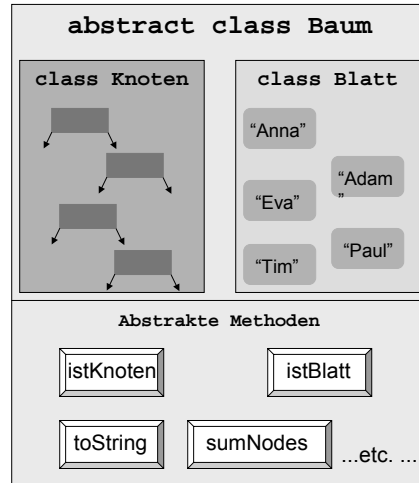
- In Baum definieren wir die Methoden irgendwie:
 

```
boolean istBlatt(){
  { return false; // äähm na ja...
}
void toString(){ // tu nix
```
- In den Unterklassen redefinieren wir sie wieder
 

```
// z.B. In Blatt:
boolean istBlatt ()
{ return true;
}
void toString()
{ System.out.println(info);
}
```

## Besser: Abstrakte Klassen

- Vereinigung von Unterklassen
- Gemeinsame Methoden
  - In der Oberklasse abstrakt erklärt
    - Nur die Signatur wird aufgeführt
  - In jeder nicht abstrakten Unterklasse implementiert
- Beispiel
  - Jedes Blatt ist ein Baum
  - Jeder Knoten ist ein Baum
  - Definiere Baum als abstrakte Klasse, die Blatt und Knoten umfasst



## Abstrakte Klasse Baum

- Klassen werden wechselseitig rekursiv, vgl. die Implementierung von Bäumen in SML.

```

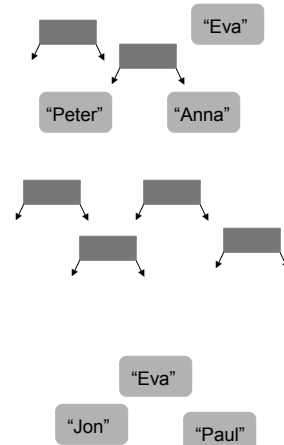
abstract class Baum
{
    ...
}

class Knoten extends Baum
{
    Baum links;
    Baum rechts;
    ...
}

class Blatt extends Baum
{
    String value;
}
    
```

Jeder Knoten ist ein Baum

Jedes Blatt ist ein Baum



## Implementierung

- Abstrakte Methoden müssen in (konkreten) Unterklassen implementiert werden
- Wird vom Computer geprüft

```
abstract class Baum{
    abstract boolean istBlatt();
    abstract int sumNodes();
    ...
}
```

```
class Knoten extends Baum{
    Baum links, rechts ;
    boolean istBlatt()
    { return false;}
    int sumNodes(){
    { return
        1+links.sumNodes()
        +rechts.sumNodes();
    }
}
```

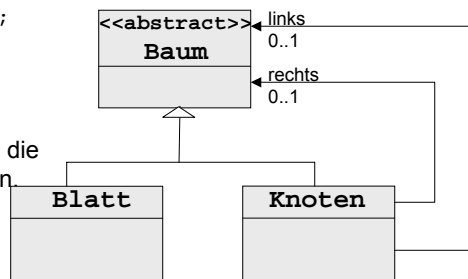
```
class Blatt extends Baum
{
    String value;
    boolean istBlatt(){
    { return true;}
    int sumNodes()
    { return 0; }
}
```

## Abstrakte Klassen

- Haben **keine** eigenen Objekte
  - Was sollte auch new Baum() liefern:
    - ein Blatt oder einen Knoten?
- Können abstrakte und konkrete Methoden enthalten:

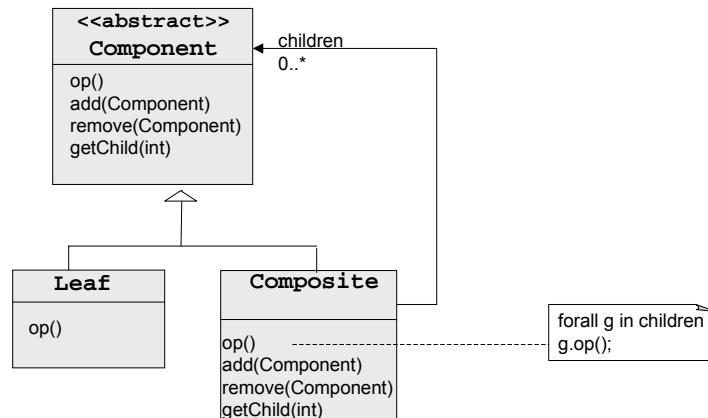
```
abstract boolean istBlatt();
boolean istKnoten()
{ return !istBlatt(); }
```

- Sobald eine Methode abstrakt ist, muss die ganze Klasse als abstrakt erklärt werden.
- Der Compiler achtet darauf, dass jede abstrakte Methode in jeder Unterklasse implementiert wird.



## Das Composite-Muster

- Das Composite-Muster dient zum Entwurf allgemeiner Baumstrukturen; es verallgemeinert das Muster für Binärbäume.



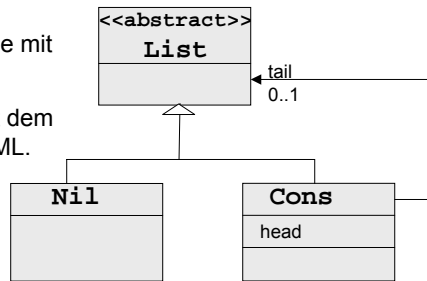
## Das Composite-Muster

Das Composite-Muster wird verwendet zur Implementierung von Objekthierarchien wie etwa hierarchischen Benutzeroberflächen, verschachtelten Diagrammen oder Parsebäumen in Übersetzern.

- Component**  
Bildet die Schnittstelle und implementiert das Standardverhalten für alle Klassen des Musters.
- Leaf**
  - repräsentiert die Blattobjekte. Ein Blatt hat keine Kinder.
  - Definiert das Verhalten der primitiven Objekte.
- Composite**
  - Definiert das Verhalten für Komponenten mit Kindern
  - Speichert die Kindkomponenten

## Listen als Abstrakte Klassen

- Listen sind leer oder nicht leer
  - Nichtleere Listen haben `head` und `tail`
  - Eine leere Liste hat **kein** Attribut
- Traditionell (in Lisp) heißt der Konstruktor einer nichtleeren Liste `cons`, die leere Liste heißt `nil`
- Wir definieren `List` als abstrakte Klasse mit Unterklassen `Cons` und `Nil`.
- Diese Listenimplementierung entspricht dem Ansatz rekursiver Datenstrukturen in SML.



## Implementierung

- Abstrakte Methoden müssen in (konkreten) Unterklassen implementiert werden
- Wird vom Computer geprüft

```

abstract class List
{
  abstract boolean isEmpty();
  abstract int length();
  abstract List addFirst();
  ...
}
  
```

```

class Cons extends List
{
  Object value; List tail;
  boolean isEmpty(){return false;}
  Cons(Object o, List l)
  { value = o; tail = l; }
  Object head() throws NoSuchElementException
  { return value; }
  List tail() throws NoSuchElementException
  { return tail; }
  int length()
  { return 1+tail.length(); }
  ...
}
  
```

```

class Nil extends List
{
  Nil(){ }
  boolean isEmpty(){return false;}
  int length()
  { return 0; }
  ...
}
  
```

## Zusammenfassung

- Binäre Bäume werden in Java implementiert:
  - als Verallgemeinerung der einfach verketteten Listen mit zwei Nachfolgerverweisen oder
  - durch eine abstrakte Oberklasse und zwei Unterklassen, einer Blatt- und eine Knotenklasse
- Eine Operation auf binären Bäume mit Knoten wird definiert:
  - durch Weitergeben der Operation an die Knotenklasse oder
  - durch Fallunterscheidung bzgl. des leeren Baums und rekursiven Aufruf der Selektoren getLeft() und getRight() von BinTree.
- Eine Operation auf beblätterten binären Bäume werden definiert:
  - durch Definition der Operation in beiden Unterklassen.

## Zusammenfassung

- Das Composite-Muster dient zur Beschreibung hierarchischer Objektstrukturen; es verallgemeinert die zweite Implementierung binärer Bäume auf Bäume mit endlich vielen Kindbäumen.
- Spezialisierung des Composite-Musters ergibt eine weitere rekursive Implementierung für Listen.