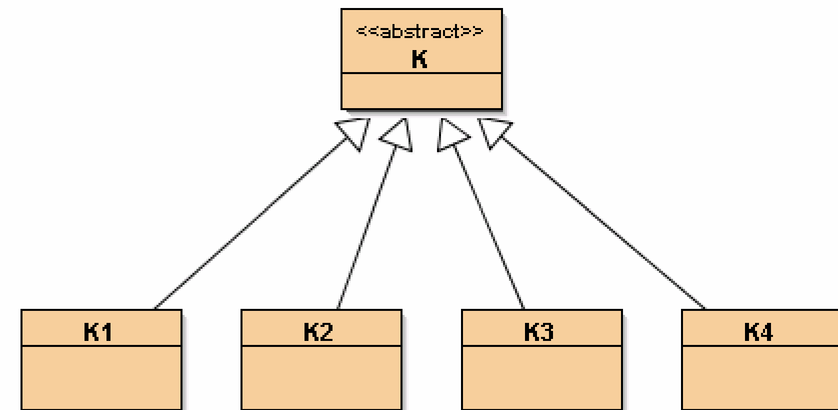
Abstrakte Klassen und Induktive Datenbereiche

Abstrakte Klassen, Induktive Datenbereiche, Bäume, Binärbäume, Bäume mit Blättern, Listen, Konstruktoren, Prädikate, Selektoren, Mutatoren, Operationen.

Abstrakte Klassen



- fassen mehrere Klassen

  $K_1 , \ldots , K_n$

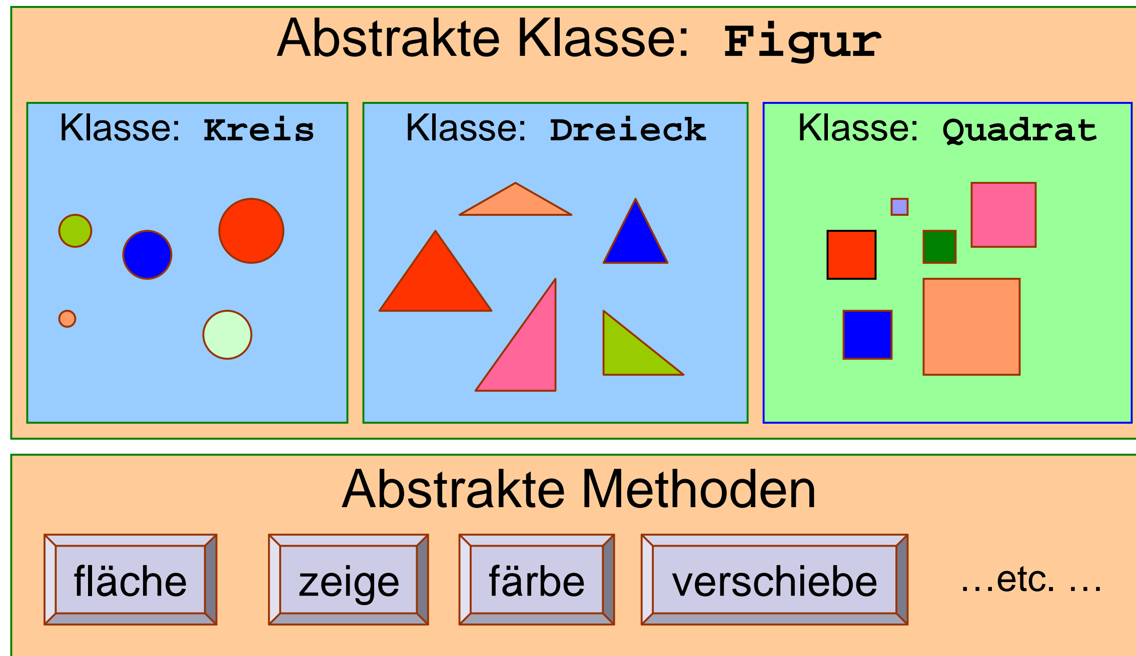
  unter gemeinsamem Aspekt zusammen.

- Mathematisch:
  - disjunkte Vereinigung $K = K_1+K_2+\ldots+K_n$
- Jedes Objekt der abstrakten Klasse gehört zu genau einer der Unterklassen
  - nur die (konkreten) Unterklassen können instanziiert werden
- Methoden können in der Oberklasse angegeben werden
  - abstrakte Methoden müssen in den Unterklassen implementiert werden
  - sonstige Methoden können sich auf abstrakten Methoden abstützen

©H. Peter Gumm, Philipps-Universität Marburg

Abstrakte Klasse : Figur

- Was haben Kreise, Dreiecke, Quadrate gemeinsam?
  - Methoden:
    - `flaeche()`
    - `umfang()`
    - `verschiebe(float,float)`
    - `färbe()`
    - `zeige()`
    - …
  - Felder:
    - `farbe`
    - …
- Zusammenfassung zu einer abstrakten Klasse **`Figur`**
  - `abstract class Figur`



- Warum abstrakt ?
  - weil ein Objekt der Klasse **`Figur`** keinen Sinn macht
  - es muss schon eine der konkreten Formen **`Kreis`**, **`Dreieck`**, **`Quadrat`**, … haben

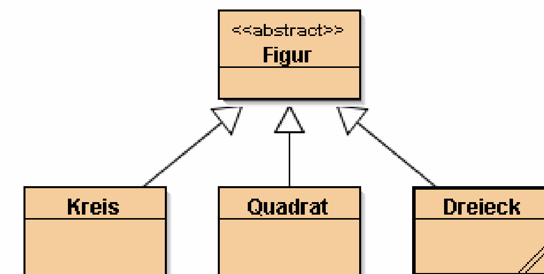
©H. Peter Gumm, Philipps-Universität Marburg

Abstrakte Klassen – nur konkret instanziierbar

- Man kann
  - .. keine Instanzen erzeugen
    - `Figur f = new Figur();` // nicht erlaubt !!!

Error: Figur is abstract; cannot be instantiated

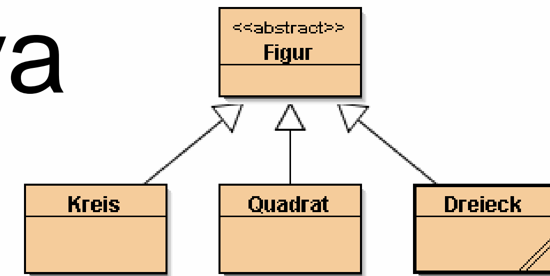
  - … nur Instanzen konkreter Unterklassen erzeugen
    - `Kreis k = new Kreis(mitte,radius)`
  - Konstruktoren der abstrakten Klasse nur aus einer Unterklasse mittels `super(...)` aufrufen.
- Beispiele für abstrakte Klassen in Java
  - Number
    mit konkreten Unterklassen
    - BigInteger, Byte, Integer, Long, Double, ...



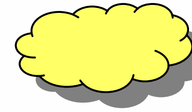
- Entspricht in anderen Programmiersprachen
  - Pascal: *Variantenrecord*
  - ML: *Abstract Data Type*

©H. Peter Gumm, Philipps-Universität Marburg

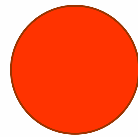
Erste Modellierung in Java



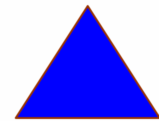

```java
public abstract class Figur {

    abstract float flaeche();
    abstract float umfang();
    abstract void verschiebe(float x, float y);
    abstract void faerbe(String farbe);
    abstract void zeige();
}
```



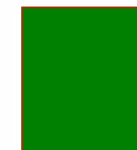
```java
public class Kreis extends Figur{
// Objektfelder
  Punkt mitte;
  float radius;
  String farbe="weiss";

// Konstruktor
  Kreis(Punkt mitte, float radius){
      this.mitte = mitte; this.radius=rad
  }

// Objektmethoden
  float flaeche(){
      return radius*radius*(float)Math.PI
  }

  float umfang(){
      return 2*radius*(float)Math.PI;
   }
```



```java
public class Dreieck extends Figur{
// Objektfelder
     Punkt ecke1, ecke2, ecke3;
     String farbe="weiss";
...
ecke3){
```



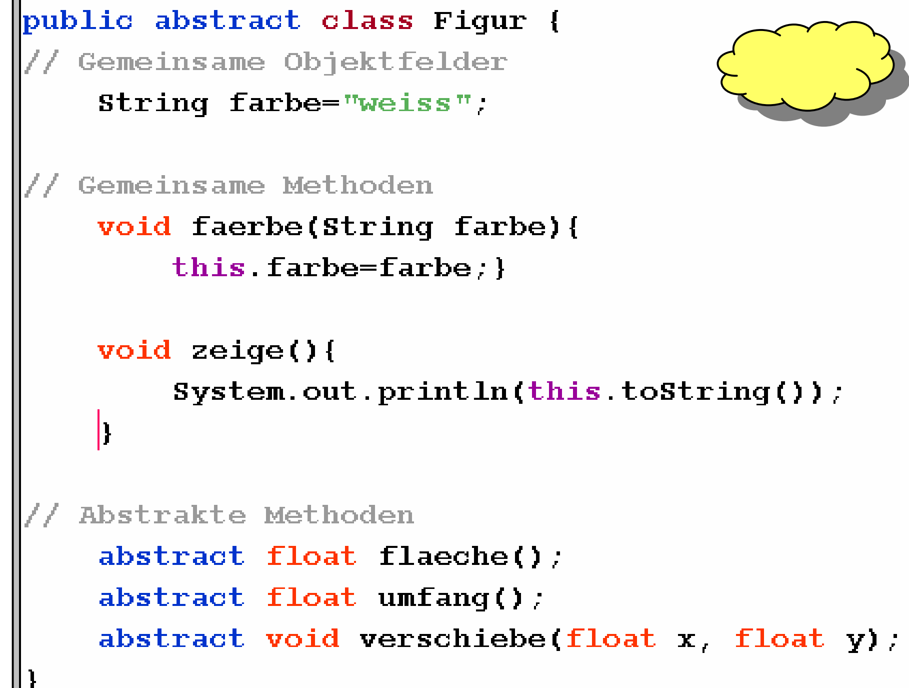
```java
public class Quadrat extends Figur{
// Objekt-Felder
     Punkt eckpunkt;
     float länge;
     String farbe="weiss";

// Konstruktor
     Quadrat(Punkt p, float länge){ this.eckpunkt=p; }
//Objektmethoden
     float flaeche(){return länge*länge;}
     float umfang(){return 4*länge;}
     void verschiebe(float x, float y){
          eckpunkt.add(new Punkt(x,y));
     }
     void faerbe(String farbe){this.farbe=farbe;}
     void zeige(){System.out.println(this.toString());}
```



©H. Peter Gumm, Philipps-Universität Marburg

Gemeinsames Verhalten

- gleiche Felder
  - **farbe**
- gleiche Methoden
  - nicht nur Name, sondern auch Code:
    - **faerbe()**
    - **zeige()**
- … können in der abstrakten Klasse implementiert werden
- Warum also nicht gleich eine konkrete Klasse ?
  - das einzige Objektfeld ist „farbe“
  - ein solches Objekt kann man kaum als Figur bezeichnen
- die abstrakte Klasse fasst Klassen zusammen, die ein gemeinsames Verhalten zeigen
  - Figuren haben
    - Umfang
    - Fläche
  - man kann sie
    - verschieben
    - färben
    - -…

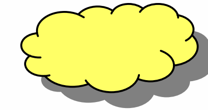


```
public abstract class Figur {
// Gemeinsame Objektfelder
    String farbe="weiss";

// Gemeinsame Methoden
    void faerbe(String farbe){
        this.farbe=farbe;}

    void zeige(){
        System.out.println(this.toString());
    }

// Abstrakte Methoden
    abstract float flaeche();
    abstract float umfang();
    abstract void verschiebe(float x, float y);
}
```

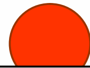

©H. Peter Gumm, Philipps-Universität Marburg

Abstrakte Klasse mit Unterklassen

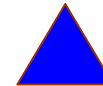
```
public abstract class Figur {
// Gemeinsame Objektfelder
    String farbe="weiss";

// Gemeinsame Methoden
    void faerbe(String farbe){ this.farbe=farbe;}
    void zeige(){System.out.println(this.toString());}

// Abstrakte Methoden
    abstract float flaeche();
    abstract float umfang();
    abstract void verschiebe(float x, float y);
} // Ende der abstrakten Klasse
```



```
public class Kreis extends Figur{
// Objektfelder
  Punkt mitte;
  float radius;

// Objektmethoden
  float flaeche(){
  float umfang(){ r
  void verschiebe(f

  public String toS

// Konstruktor
  Kreis(Punkt mitte
```



```
public class Dreieck extends Figur{
// Objektfelder
    Punkt ecke1, ecke2, eck

//Objektmethoden
    float flaeche(){ return
    float umfang(){ return
    void verschiebe(float x
    public String toString(

// Konstruktor
    Dreieck(Punkt ecke1, Pu
        this.ecke1=ecke1;
        this.ecke2=ecke2;
```



```
public class Quadrat extends Figur{
// Objekt-Felder
    Punkt eckpunkt;
    float länge;

//Objektmethoden
    float flaeche(){return länge*länge;}
    float umfang(){return 4*länge;}
    void verschiebe(float x, float y){
        eckpunkt.add(new Punkt(x,y));
    }
    public String toString(){ return
        "Quadrat mit Eckpunkt "+eckpunkt.to
// Konstruktor
```

©H. Peter Gumm, Philipps-Universität Marburg

Abstrakte Klassen in Java

- haben keine eigenen Objekte
  - Was sollte auch `new Figur()` liefern:
  - einen Kreis, ein Dreieck oder ein Quadrat ?
- können abstrakte und konkrete Methoden enthalten

```
abstract boolean flaeche();
void faerbe(String farbe){ this.farbe = farbe; }
```

- Sobald eine Methode abstrakt ist, muss die ganze Klasse abstrakt erklärt werden
- Der Compiler achtet darauf, dass jede abstrakte Methode in jeder Unterklasse implementiert wird.
- Wie in allen anderen Klassen auch:
  - Felder von Unterklassen können Oberklassen referenzieren
  - dadurch erhält man *induktive* Datentypen

©H. Peter Gumm, Philipps-Universität Marburg

Induktiv definierte Daten

- Die natürlichen Zahlen sind induktiv definiert
  - 0 ist eine natürliche Zahl
  - Wenn N eine natürliche Zahl ist, dann auch N+1

0, |, ||, |||, ||||, |||||, ||||||, ...

- Binärzahlen kann man induktiv definieren
  - Jede der Ziffern 0 und 1 ist eine Binärzahl
  - Ist B eine Binärzahl, dann auch
    - B0 und B1.

0, 1,
00, 01, 10, 11,
000, 001, 010, 011,

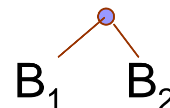
- Strings kann man induktiv definieren
  - Der leere String " " ist ein String
  - Ist S ein String und z ein Zeichen, dann ist S z ein String

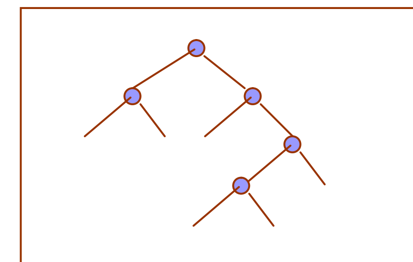
" ",
"a", "b", ..., "aa", "ab", ... , "ba", "bb", ... "aaa", "aab", ...

- Listen
  - die leere Liste [ ] ist eine Liste
  - ist e ein Element und L = [x_1, …, x_n] eine Liste, dann ist auch cons(e,L) := [e, x_1, …, x_n] eine Liste.

[ ], [2,3,5], [1,6, 19, 24, 0, 42]

- Binärbäume (ohne Blätter)
  - Der leere Baum ist ein Binärbaum
  - Sind B_1 und B_2 Binärbäume, dann auch

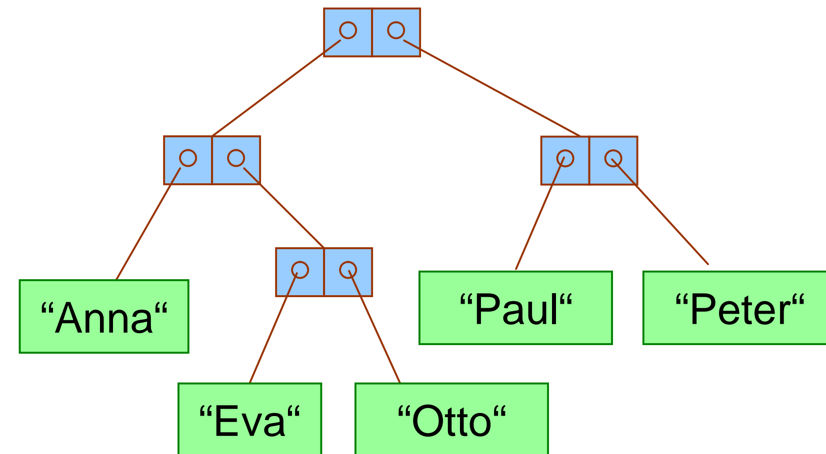




©H. Peter Gumm, Philipps-Universität Marburg

Bäume mit Blättern

- Jeder Zweig soll in einem Blatt enden
- Die Information speichern wir in den Blättern
- Jeder Knoten hat zwei Unterbäume
- In jedem Blatt speichern wir eine textuelle Information



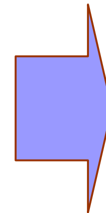

```
class Knoten{
    ??? links;
    ??? rechts;
}
```

```
class Blatt{
    String info;
}
```


©H. Peter Gumm, Philipps-Universität Marburg

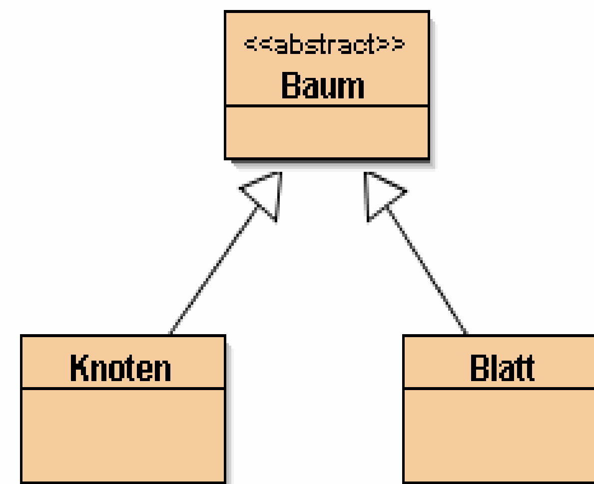
Ein Baum ist ein Blatt oder ein Knoten

- Ein Baum ist
  - ein Blatt (mit einem Inhalt), oder
  - ein Knoten mit zwei Unter**bäum**en
- Fasse Blatt und Knoten zu abstrakter Klasse Baum zusammen.



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

- Blatt und Knoten werden Unterklassen von Baum
- Viele Methoden müssen für alle Bäume funktionieren
  - istBlatt()
  - istKnoten
  - depth()
  - draw()

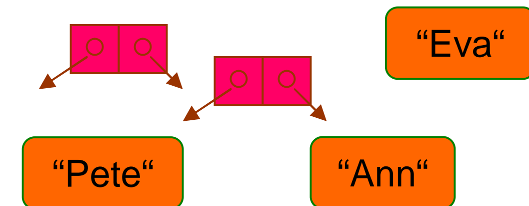


©H. Peter Gumm, Philipps-Universität Marburg

Abstrakte Klasse Baum

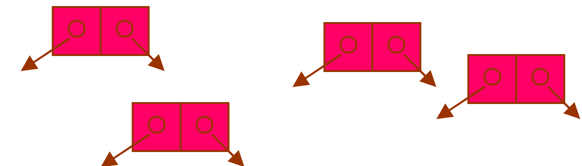
- Klassen werden wechselseitig rekursiv

```
abstract class Baum{
   …
}
```



Jeder Knoten
ist ein Baum

```
class Knoten extends Baum{
    Baum links;
    Baum rechts;
      …
}
```



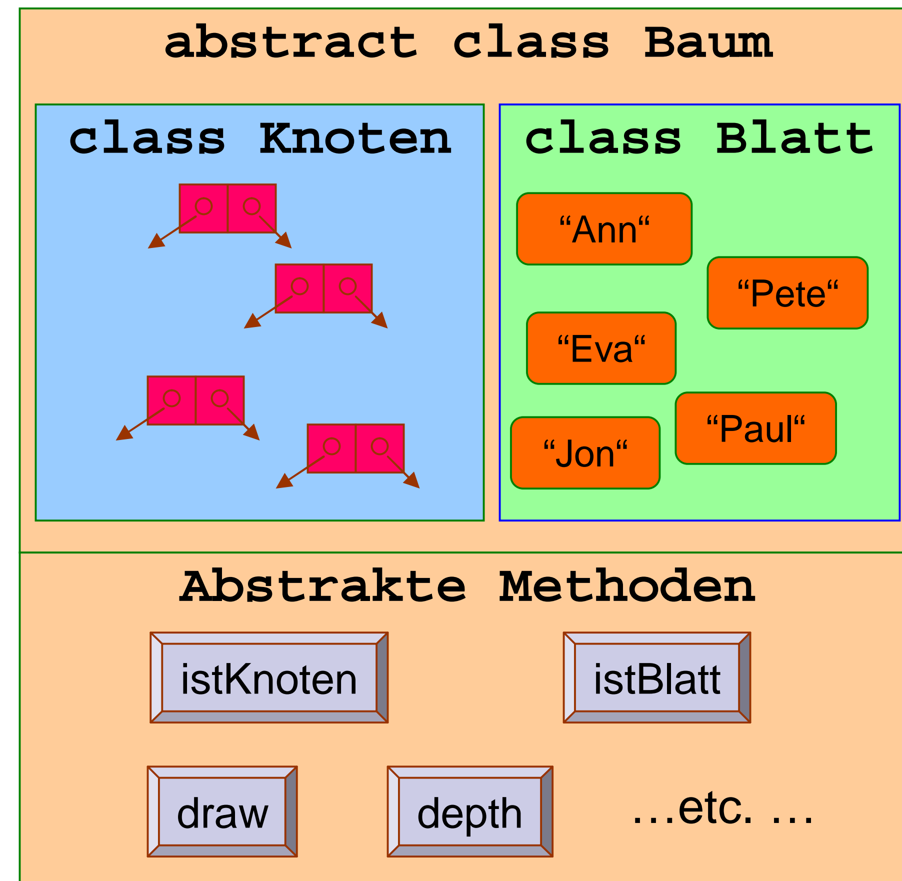
Jedes Blatt
ist ein Baum

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

©H. Peter Gumm, Philipps-Universität Marburg

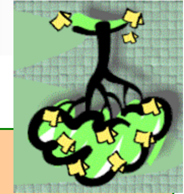
Abstrakte Klasse Baum

- Disjunkte Vereinigung von Unterklassen
  - ein Baum ist
    - entweder ein Blatt
    - oder ein Knoten
- Gemeinsame Methoden
  - In der Oberklasse abstract erklärt
  - Wichtig zu wissen:
    - bin ich ein Knoten ?
    - bin ich ein Blatt ?
  - `boolean istBlatt()`



©H. Peter Gumm, Philipps-Universität Marburg

Implementierung



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

```
abstract class Baum{

   abstract boolean istBlatt();

   abstract int depth();
        …
}
```

```
class Knoten extends Baum {
   Baum links, rechts;

   boolean istBlatt(){
        return false;}

   int depth(){
     return
        1+max(links.depth(),
              rechts.depth());
   }
}//
```

```
class Blatt extends Baum{
     String info;

     boolean istBlatt(){
       return true;}

     int depth(){ return 0; }
        …
}
```


©H. Peter Gumm, Philipps-Universität Marburg

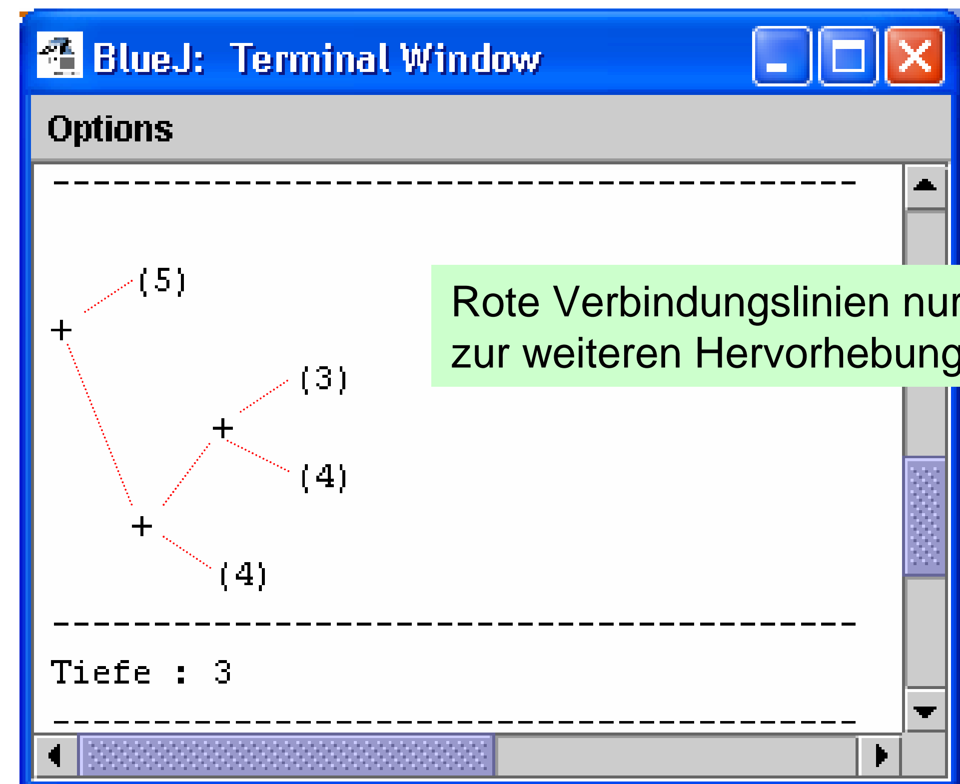
Operationen auf rekursiven Daten

- Funktionen auf induktiv definierten Daten sind rekursiv am einfachsten
  - Beispiel : `draw()`
    - Hilfsfunktion `draw(int n)`
    - Malt einen Baum ab Spalte n
  - In Blatt:

```
println(info);
```

  - In Knoten:

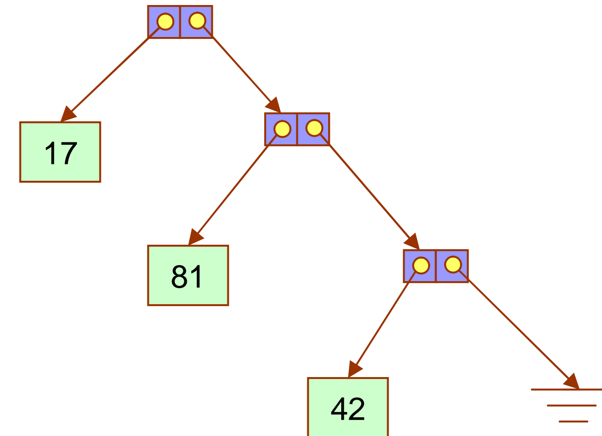
```
rechts.draw(n+5);
indent(n,"+");
links.draw(n+5);
```

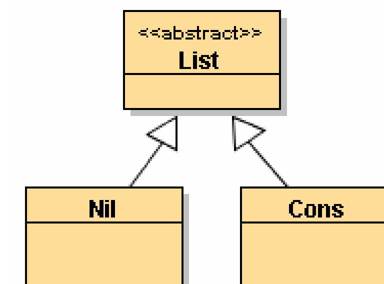


©H. Peter Gumm, Philipps-Universität Marburg

Listen rekursiv

- Eine Liste ist
  - entweder die leere Liste
  - oder sie hat
    - ein erstes Element
    - und eine Rest-Liste
- Eine Liste ist ein eindimensionaler Baum
  - statt Blatt :
    die leere Liste
  - statt linker und rechter Teilbaum:
    - inhalt und
    - Restliste
- Implementierung als abstrakte Klasse:
  - Nil – die Klasse die nur ein Objekt hat:
    - die leere Liste
  - Cons – die Klasse die alle nichtleeren Listen enthält

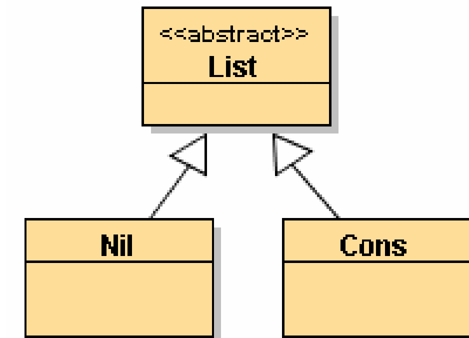




©H. Peter Gumm, Philipps-Universität Marburg

Implementierung …

```java
public abstract class Liste {

   abstract boolean istLeer();
   abstract int length();

} // Ende der abstrakten Klasse Liste
```




```java
public class Nil extends Liste{
// Objektfelder

  /** Keine Objektfelder */

//   Objekt-Methoden

 boolean istLeer(){ return true; }
 int length(){ return 0; }

}// Ende der Klasse Nil
```

```java
public class Cons extends Liste{
//    Objekt-Felder
      String inhalt; // diesmal Stzrings
      Liste rest;

//    Objekt-Methoden
      boolean istLeer(){ return false; }
      int length(){ return 1+rest.length(); }

// Konstruktor
      public Cons(String e, Liste l){
          inhalt = e;
          rest = l;
      }
}// Ende der Klasse Cons
```


©H. Peter Gumm, Philipps-Universität Marburg

… und Test

- im *codepad* erzeuge Listenobjekt [„Otto“,“Eva“,“Udo“] …
  - new Cons(„Otto“, new Cons(„Eva“, new Cons(„Udo“, new Nil())))
- … speichere es unter *test1*
  - in die *workbench* ziehen
- … inspiziere es
  - *Inspizieren* aus Kontextmenü von *test1*
  - Referenzen per Mausklick folgen

©H. Peter Gumm, Philipps-Universität Marburg

Typisch rekursiv

- append
  - soll zwei Listen zu einer Neuen zusammenfügen
  - zweite Liste ist Parameter
    - Aufruf z.B.: liste1.append(liste2)
  - Ergebnis: neue Liste
    - keine Veränderung der Ausgangslisten

In Klasse Liste

```
public abstract class Liste {

    abstract boolean istLeer();
    abstract int length();

    abstract Liste append(Liste l);
```

In Klasse Nil

```
    Liste append(Liste l){ return l; }
```

In Klasse Cons

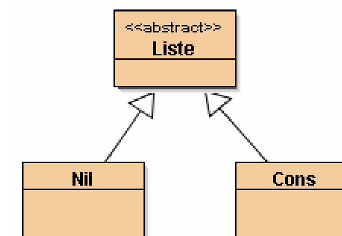
```
    Liste append(Liste l){
        if(l.istLeer()) return this;
        else return new
             Cons(this.inhalt,rest.append(l));
    }
```

Ein kleines Experiment

cons1: Cons

cons2: Cons

```
new Cons("Otto",new Cons("Eva",new Cons("Udo",new Nil())))
<object reference>   (Cons)
cons1.append(cons1)
<object reference>   (Cons)
cons2.length()
6   (int)
```



Indexkarte für Listen

- Beliebige Listenoperationen können sich auf diesen Methoden abstützen:

Liste

Konstruktoren:
Nil()
Cons(String,Liste)

Methoden:
boolean istLeer()
falls *istLeer() ≠ false*:
String head()
Liste tail(Liste)

- Wir verstecken alle anderen Felder und Methoden hinter dem Schlüsselwort:
private

```
public class Cons extends Liste{

 // Objektfelder - privat
    private String inhalt;  // diesmal Strings
    private Liste rest;

 // Konstruktor
    public Cons(String e, Liste l){
        inhalt=e;
        rest=l;
    }

// Objektmethoden
    //Selektoren
    public String head(){ return inhalt;}
    public Liste  tail(){ return rest; }

    // Prädikat
    public boolean istLeer(){ return false;}

    // Sonstige Objektmethoden
    public int length(){ return 1+rest.length();}
```

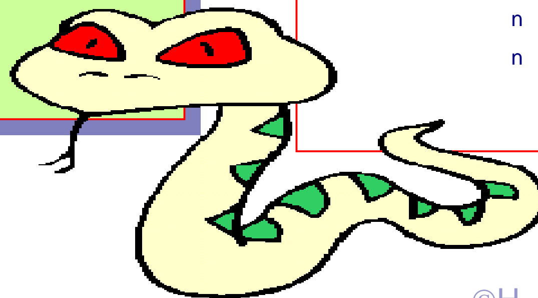
Notwendige Operationen

- Konstruktoren,
  - bauen Elemente des Datentyp
  - Meist gibt es mehrere Konstruktoren
- Prädikate
  - Testen, ob ein Datenobjekt mit einem bestimmten Konstruktor aufgebaut wurde
  - Hier: ob eine Liste *leer* oder *nichtleer* ist
- Selektoren
  - Liefern die Bestandteile, aus der das Datenobjekt aufgebaut wurde
  - Selektoren gehen davon aus, dass das Datenobjekt mit einem bestimmten Konstruktor gebaut wurde

Im Falle der Listen:

- Listenkonstruktoren

  Eine Liste ist leer, oder nicht leer,
  dh. erzeugt durch
  - `Nil()` oder durch
  - `Cons(String, Liste)`
- Listenprädikat
  - `istLeer()`
    testet, ob eine Liste leer ist
- Selektoren
  - Falls die Liste leer ist, hat sie keinen Bestandteil, braucht also keinen Selektor
  - Ansonsten besteht sie aus den Bestandteilen.
    - `head()` - Kopf, erstes Element
    - `tail()` - Liste der restlichen Elemente



©H. Peter Gumm, Philipps-Universität Marburg

Im Falle der Bäume

Baum

Konstruktoren:
- Blatt(String)
- Knoten(Baum,Baum)

Methoden
- istBlatt()
- istKnoten()

falls *istBlatt()* :
- String getInhalt()

falls *istKnoten()* :
- Baum getLeft()
- Baum getRight()

Im Falle der Bäume:

- Baumkonstruktoren

  Eine Baum ist ein Blatt, oder ein Knoten,
  dh. erzeugt durch
  - `Blatt(String inh)` oder durch
  - `Knoten(Baum b1, Baum b2)`

- Baumprädikat
  - `istBlatt()` **bzw** `istKnoten()`
    testet, durch welchen der
    Konstruktoren der Baum erzeugt wurde

- Selektoren
  - Falls `istBlatt():`
    - `getInhalt()`
  - Falls `istKnoten()`
    - `left()` - linker Teilbaum
    - `right()` - rechter Teilbaum

©H. Peter Gumm, Philipps-Universität Marburg

Mutatoren, Operationen

- Grundsätzlich zwei Möglichkeiten für Methoden zur Manipulation von Daten:
- Mutatoren
  - verändern das Objekt
  - sind Kennzeichen des *imperativen Programmierens*
  - `void` Methoden sind immer Mutatoren
  - Mutatoren sind manchmal effizienter
- Operationen
  - lassen Objekt unverändert
  - Resultat ist neues Objekt
  - sind Kennzeichen des *deklarativen Programmierens*
  - einfacher zu verstehen und zu handhaben
  - weniger fehleranfällig
  - können zu unbenötigten Duplizitäten führen

- `append` ist eine *Operation*
- Wir könnten stattdessen einen Mutator `attach()` definieren:

In Klasse `Cons`

```
public Liste attach(Liste l){
        Cons last = this;
        while(!last.tail.isEmpty())
            last=(Cons)(last.tail);
        last.tail=l;
        return this;
  }
```

- Vorteil:
  - schneller
- Nachteil:
  - destruktiv
  - `liste1.attach(liste1)` führt zu einer Katastrophe !!!

©H. Peter Gumm, Philipps-Universität Marburg