```
class Person implements Comparable<Person> {
    String nachname, vorname;
    public Person(String v, String n) {
        vorname = v; nachname = n;
    }
    public boolean equals(Object o) {
        if(o == null ||!(o instanceof Person)) {
            return false;
        }
        Person other = (Person)o;
        return nachname.equals(other.nachname) &&
            vorname.equals(other.vorname);
    }
    public int hashCode() {
        return nachname.hashCode() + vorname.hashCode();
    }
```

Wir können nun *HashSet⟨Person⟩* verwenden. Hierzu haben wir *equals* und *hashCode* implementiert.

Für *TreeSet⟨Person⟩* muß *Person* auch noch das Interface *Comparable⟨Person⟩* implementieren.

```
public int compareTo(Person other) {
  int nachnameOrder = nachname.compareTo(other.nachname);
  if(nachnameOrder != 0) {
    return nachnameOrder;
  }
  else {
    return vorname.compareTo(other.vorname);
  }
}
```

Gegeben sei eine Menge von Zahlen $\{3, 5, 10\}$.

Welche Zahlen können wir bilden, indem wir die Zahlen einer Untermenge addieren?

Antwort: $\{0, 3, 5, 8, 10, 13, 15, 18\}$.

```
static Set<Integer> subsetsums(int size[]) {
  Set<Integer> sums = new TreeSet<Integer>();
  sums.add(0);
  for(int k : size) {
    Set<Integer> newsums = new TreeSet<Integer>();
    for(int l : sums) {
      newsums.add(k + l);
    }
    sums.addAll(newsums);
  }
  return sums;
}
```

# Abbildungen und Wörterbücher

Das Interface $Map\langle K, E\rangle$ modelliert Abbildungen $K \to E$.

So können wir ein Wörterbuch implementieren:

```
Map<String, String> deuita = new HashMap<String, String>();
deuita.put("schneiden", "tagliare");
deuita.put("Butter", "burro");
deuita.put("Igel", "riccio");
String x = deuita.get("Butter");
```

Mit *put* legen wir Werte fest und mit *get* schlagen wir sie nach.

Typische Implementierungen sind *HashMap* und *TreeMap*.

Bei den *Subsetsums* berechneten wir die Summen, aber nicht durch welche Untermenge sie gebildet werden. Hier berechnen wir eine *Map*, welche uns zu jeder Summe *eine* Zahl ihrer Menge gibt.

```
static Map<Integer, Integer> subsetmap(int size[ ]) {
  Map<Integer, Integer> sums = new TreeMap<Integer, Integer>();
  sums.put(0, 0);
  for(int k : size) {
    Map<Integer, Integer> newsums;
    newsums = new TreeMap<Integer, Integer>();
    for(int l : sums.keySet()) {
      newsums.put(k + l, k);
    }
    newsums.putAll(sums);
    sums = newsums;
  }
  return sums;
}
```

# List

Es gibt in der Java-Standardbibliothek das Interface $List\langle E\rangle$ (nicht zu verwechseln mit unserer Implementierung).

Folgende Operationen gibt es u.a.:

- **void** *add*(*E x*), füge $x$ am Ende ein.
- **void** *add*(**int** *n*, *E x*), füge $x$ an Position $n$ ein.
- **void** *set*(**int** *n*, *E x*), ersetzt die $n$te Position durch $x$.
- *E get*(**int** *n*), was ist an Position $n$?
- **int** *size*(), wieviele Elemente enthält es?

Es wird sowohl ein „wachsendes“ Array als auch eine Liste modelliert.

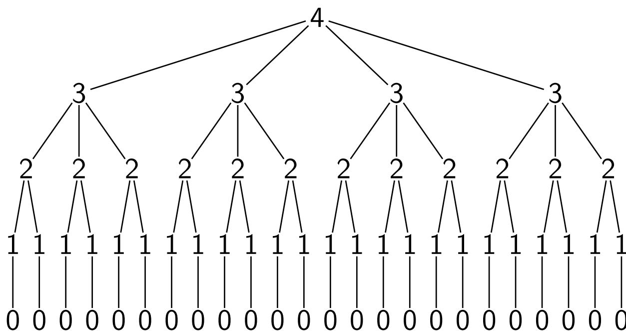
Typische Implementierungen: $ArrayList\langle E\rangle$ und $LinkedList\langle E\rangle$.

Oft ist es bequemer eine *List* als ein normales Array zu verwenden.

# Visitor

Erinnern wir uns an die Klasse *Tree*.

Sie implementiert Bäume, deren Knoten Zahlen enthalten.




Öffentlichen Methoden *getRoot*(), *getChildren*(), *addChild*(*Tree t*) und Konstruktor **new** *Tree*(**int** *root*).

# Visitor

Wie können wir das Gesamtgewicht eines Baumes berechnen?

Wir machten dies früher durch Schreiben einer geeigneten Methode.

Nachteil: „Enge Kopplung“.

So müssen wir oft die Klasse selbst verändern, was nicht gut ist.

Alternative: Nur die Grundoperationen *getRoot*() und *getChildren*() verwenden.

Weitere Möglichkeit: Manchmal hilft es immens einen Visitor oder Iterator zu verwenden.

# Visitor

Wir erweitern die Klasse *Tree* so, daß sie einen *TreeVisitor* „akzeptiert“.

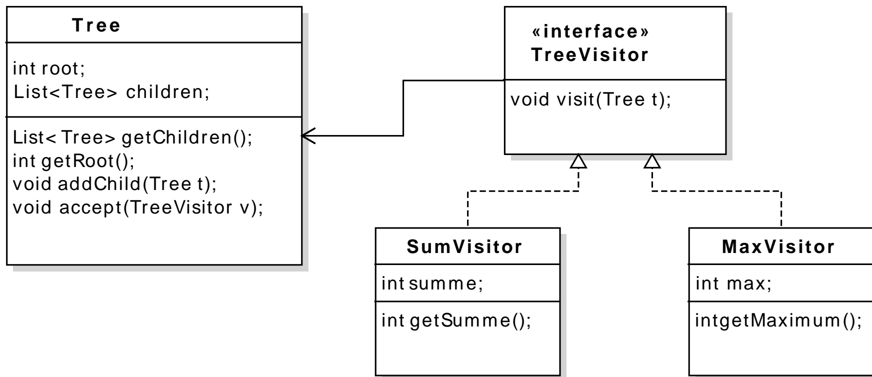
```
interface TreeVisitor {
   public void visit(Tree t);
}
```

Die Klasse *Tree* ermöglicht es einem *TreeVisitor* durch einen einfachen Aufruf alle Knoten des Baumes „zu besuchen“.

```
Tree tree = new Tree(24);
...
TreeVisitor visitor = new MyTreeVisitor();
tree.accept(visitor);
```

Hierin ist *MyTreeVisitor* eine konkrete Klasse, die *TreeVisitor* implementiert.

# Klassendiagramm



# Visitor

So implementieren wir die Methode *accept*(*Tree t*) in der Klasse *Tree*:

```
public void accept(TreeVisitor v) {
  v.visit(this);
  for(Tree child : children) {
    child.accept(v);
  }
}
```

Die Gesamtsumme aller Knoten können wir jetzt so berechnen:

```
public static void main(String args[ ]) {
    Tree t1 = new Tree(1);
    Tree t2 = new Tree(2);
    Tree t3 = new Tree(3);
    Tree t4 = new Tree(4);
    Tree t5 = new Tree(5);
    Tree t6 = new Tree(6);
    t1.addChild(t2);
    t1.addChild(t3);
    t3.addChild(t4);
    t3.addChild(t5);
    t3.addChild(t6);
    SumVisitor visitor = new SumVisitor();
    t1.accept(visitor);
    System.out.println(visitor.summe());
}
```

Hierfür müssen wir nur den geeigneten Visitor implementieren:

```
class SumVisitor implements TreeVisitor {
  private int sum = 0;
  public void visit(Tree t) {
    sum = sum + t.getRoot();
  }
  public int summe() {
    return sum;
  }
}
```

Dies ist eine Art neue Funktionalität einer Klasse hinzuzufügen.

Es ist jetzt leicht, weitere Visitors zu schreiben.

Zum Beispiel, um das Maximum zu berechnen:

```
class MaxVisitor implements TreeVisitor {
  private int max = −1;
  public void visit(Tree t) {
    int value = t.getRoot();
    if(value > max) {
      max = value;
    }
  }
  public int maximum() {
    return max;
  }
}
```