Programmiermethoden

Sven Eric Panitz
Hochschule RheinMain

Version 719

Generiert von LectureNotes Teaching System
12. April 2020

Dieses Skript ist begleitend zum ersten Teil des Moduls Programmiermethoden und Techniken des Studiengangs AI. Es hat im SS 2017 umfassende Änderungen bekommen, um die mit Java 8 eingeführten Spliterator und Stream-Apis zu berücksichtigen.

Inhaltsverzeichnis

1 Iterierbare Objekte

Im ersten Semester haben wir viele klassische Konzepte der imperativen Programmierung kennen gelernt. Eine zentrale Art von Anweisungen stellen die Schleifenkonstrukte dar, die zu kontrolliertem wiederholten Ausführen eines Code-Blocks verwendet werden. In diesen Kapitel wird es darum gehen Schleifen zu ersetzen, durch Objekte, die die Schleifenlogik repräsentieren.

Gehen wir von den Schleifen aus. Eine häufig zu lösende Aufgabe in der Programmierung ist es, durch bestimmte Elemente zu iterieren. Für die Iteration stehen die Schleifenkonstrukte zur Verfügung. Eine Iteration (lateinisch iteratio, die Wiederholung), soll nacheinander einen bestimmten Code-Block für verschiedene Elemente durchführen. In klassischer Weise wird zum Iterieren eine Indexvariable in einer Schleife durchgezählt. Mit dem Index wird dann in einem Schleifenrumpf jeweils etwas gerechnet.

Eine klassische Schleife zum Durchlaufen der geraden Zahlen kleiner 10:

Iteration1.java

```
package name.panitz.pmt.iteration;
public class Iteration1{
  public static void main(String[] args){
    for (int i = 2; i<10; i = i+2){
      System.out.println(i);
    }
  }
}
```

Das obige Beispiel ist in keiner Weise ein objektorientiertes Beispiel. Es ist eine ganz einfache Schleife, wie aus der imperativen Programmierung gewohnt. In einer objektorientierten Umsetzung wird stets versucht, ein Objekt zu definieren, das seine Funktionalität kapselt. In diesem Fall ein Objekt, das weiß, wie zu iterieren ist. Hierzu kann man versuchen, alle im Kopf der for-Schleife gebündelten Informationen in einem Objekt zu kapseln.

In der for-Schleife gibt es zunächst eine Zählvariable i, dann einen Test, ob die Schleife noch weiter durchlaufen werden soll, und eine Anweisung, wie sich die Schleifenvariable verändert, wenn es zum nächsten Durchlauf geht. Packt man diese drei Informationen in ein Objekt, so erhalten wir die folgende Klasse:

GeradeZahlenKleiner10Iterierer.java

```
package name.panitz.pmt.iteration;
public class GeradeZahlenKleiner10Iterierer{
  int i; //die Schleifenvariable
  GeradeZahlenKleiner10Iterierer(int i){
    this.i = i; //Initialisierung der Schleifenvariable
  }
  boolean schleifenTest(){
    return i < 10; //Test ueber die Schleifenvariable
  }
  void schleifeWeiterschalten(){
    i = i + 2; //fuer naechsten Schleifendurchlauf
  }
```

Im Schleifenrumpf wird der aktuelle Wert der Iteration benutzt. In der obigen for-Schleife ist das die Zahl i. Auch hierfür können wir noch eine Methode vorsehen.

GeradeZahlenKleiner10Iterierer.java

```
int schleifenWert(){
  return i;
}
```

Mit einem Objekt dieser Klasse können wir jetzt eine Schleife durchlaufen, ebenso wie in der ersten Schleife oben:

GeradeZahlenKleiner10Iterierer.java

```
  public static void main(String[] args){
    for (GeradeZahlenKleiner10Iterierer it
          = new GeradeZahlenKleiner10Iterierer(0)
        ; it.schleifenTest()
        ; it.schleifeWeiterschalten()){
      System.out.println(it.schleifenWert());
    }
  }
}
```

1.1 Eine Schnittstelle zum Iterieren

Noch ist überhaupt kein Vorteil darin erkennbar, dass wir die Komponenten, die zur Programmierung einer Schleife benötigt werden (Initialisierung des Iterierungsobjektes, Test, Weiterschaltung, Abfragen des Schleifenelementes) in einem Objekt gekapselt haben. Der Vorteil der Objektorientierung liegt in zusätzlichen Abstraktionen. Eine der

stärksten Abstraktionen in Java sind Schnittstellen. Naheliegend ist es, die Methoden der obigen Klasse in einer Schnittstelle vorzugeben:

IntIterierer.java

```
package name.panitz.pmt.iteration;

import java.util.function.Consumer;

public interface IntIterierer{
  boolean schleifenTest();
  void schleifeWeiterschalten();
  Integer schleifenWert();
```

Zwei neue Konzepte bringt Java 1.8 mit, die es ermöglichen, unsere Schnittstelle um wesentliche Funktionalität zu erweitern und ihre eine noch zentralere Rolle zukommen zu lassen.

- Mit Java 1.8 können Schnittstellen, wie bisher nur von abstrakten Klassen her bekannt, auch Methoden bereits konkret implementieren, wobei sie die abstrakten Methoden bereits benutzen können. Dieses sind die sogenannten default-Methoden.
- Mit Java 1.8 wurden sogenannte Lambda-Ausdrücke, die auch als Closures bezeichnet werden, der Sprache Java hinzugefügt. Sie ermöglichen eine einfache Übergabe von einzelnen Methoden als Parameter.

Mit diesen neuen Sprachkonzepten wird eine bisher in Java nur schwer realisierbare Programmiermethodik möglich. Die sogenannte Programmierung höherer Ordnung. Dabei werden Methoden oder auch Codeblöcke als Parameter anderen Methoden übergeben. Dieses ermöglicht eine wesentlich stärkere Abstraktion in der Parametrisierung des Codes. Es kann jetzt über eine ganze Befehlsfolge auf einfache Weise abstrahiert werden.

Es lässt sich so für die Schleifenschnittstelle eine Methode implementieren, die die Schleife ausführt. Wir nennen diese Methode vorerst `run`. Sie bekommt als Parameter ein Objekt des Typs `Consumer`. Dieses ist eine funktionale Schnittstelle. Objekte dieser Schnittstelle haben die Methode `accept`.

Die Methode `run` realsiert die Schleife. Hierzu verwendet sie die noch abstrakten Methoden `schleifenTest`, `schleifeWeiterschalten` und `schleifenWert`. Die Initialisierung des Schleifenobjekts findet weiterhin durch den Konstruktor statt.

Im Rumpf der Schleife wird die Methode `accept` des Parameters als Anwendungslogik ausgeführt.

IntIterierer.java

```
  default void run(Consumer<Integer> action){
    for( ; schleifenTest(); schleifeWeiterschalten()){
      action.accept(schleifenWert());
    }
  }
}
```

Die ursprüngliche Klasse zum Iterieren über einen Zahlenbereich lässt sich nun als Implementierung dieser Schnittstelle schreiben. Um die Klasse dabei noch ein wenig allgemeiner zu schreiben, übergeben wir jetzt auch die obere Schranke der Iteration im Konstruktor.

GeradeZahlenIterierer.java

```
package name.panitz.pmt.iteration;
public class GeradeZahlenIterierer implements IntIterierer{
  int from;
  int to;
  GeradeZahlenIterierer(int from, int to){
    this.from = from;
    this.to = to;
  }
  public boolean schleifenTest(){
    return from < to;
  }
  public void schleifeWeiterschalten(){
    from = from + 2;
  }
  public Integer schleifenWert(){
    return from;
  }
```

Das so erhaltene Schleifenobjekt kann zum Iterieren durch Aufruf der Methode `run` verwendet werden. Man braucht die eigentlich Schleife nicht mehr selbst zu implementieren. Das Initialisieren der Schleife findet durch Aufruf des Konstruktors der konkreten Schleifenklasse statt, das Ausführen der Schleife durch Aufruf der Methode `run` auf dem Objekt. Der Schleifenrumpf wird als Lambda-Ausdruck der Methode `run` übergeben.

GeradeZahlenIterierer.java

```
  public static void main(String[] args){
    new GeradeZahlenIterierer(0,10).run(x->System.out.println(x));
  }
}
```

1.2 Eine generische Version der Iterationsschnittstelle

Eine weitere Abstraktion, die von heutigen Programmiersprachen angeboten wird, liegt in der Möglichkeit, Typen variabel zu halten über generische Klassen und Schnittstellen. Statt eine Iterations-Schnittstelle für ganze Zahlen zu definieren, lässt sich allgemein eine solche für beliebig aber feste Typen definieren:

GenerischerIterierer.java

```
package name.panitz.pmt.iteration;

import java.util.function.Consumer;

public interface GenerischerIterierer<A> {
  boolean schleifenTest();
  void schleifeWeiterschalten();
  A schleifenWert();

  default void run(Consumer<? super A> action){
    for( ; schleifenTest(); schleifeWeiterschalten()){
      action.accept(schleifenWert());
    }
  }
}
```

Statt des festen Typs `Integer` der ursprünglichen Schnittstelle wurde ein variabler Typ `A` eingeführt. Beim Implementieren dieser generischen Schnittstelle gilt es jetzt zu entscheiden, welcher konkrete Typ für diese Typvariable eingesetzt werden soll, also was für einen Typ die Elemente haben, über die iteriert werden soll. In unserem Beispielfall sind dieses ganze Zahlen. Da für Typvariablen in generischen Typen keine primitiven Typen eingesetzt werden dürfen, benutzen wir hier die Klasse `Integer` und vertrauen an mehreren Stellen darauf, dass intern Daten vom Typ `int` und `Integer` gegenseitig konvertiert werden. Dieses wird als automatisches *Boxing/Unboxing* bezeichnet.

Die neue Implementierung der Iterationsklasse sieht entsprechend wie folgt aus:

GeradeZahlenIterierer2.java

```
package name.panitz.pmt.iteration;
public class GeradeZahlenIterierer2
         implements GenerischerIterierer<Integer>{
  int from;
  int to;
  GeradeZahlenIterierer2(int from, int to){
    this.from = from;
    this.to = to;
  }
  public boolean schleifenTest(){
    return from < to;
  }
  public void schleifeWeiterschalten(){
    from = from + 2;
  }
  public Integer schleifenWert(){
    return from;
  }
```

An der Benutzung ändert sich in diesem Fall wenig gegenüber der nicht generischen Version.

GeradeZahlenIterierer2.java

```
  public static void main(String[] args){
    new GeradeZahlenIterierer2(0,10).run(x->System.out.println(x));
  }
}
```

Gewonnen haben wir nun die Möglichkeit, auf gleiche Weise Iteratorobjekte zu implementieren, deren Elemente andere Typen haben.

1.3 Die Standard-Schnittstelle `Iterator`

Die im vorangegangenen Abschnitt entwickelte Schnittstelle `GenerischerIterierer` hat ein Pendant in Javas Standard-API. Im Paket `java.util` findet sich dort ebenfalls eine generische Schnittstelle, die beschreibt, dass ein Objekt zum Iterieren benutzt werden kann, die Schnittstelle `Iterator`. Diese ist allerdings ein wenig anders aufgebaut, als unsere Schnittstelle. Sie definiert nur zwei abstrakte Methoden:

- `boolean hasNext()`: diese Methode entspricht ziemlich genau unserer Methode `schleifenTest`. Sie ergibt `true`, wenn es noch weitere Elemente zum Iterieren gibt.

- `A next()`: diese Methode vereint die zwei Methoden `schleifenWert()` und `schleifeWeiterschalten` unserer Schnittstelle. Es wird dabei das aktuelle Element zurück gegeben und gleichzeitig intern der Schritt weiter zum nächsten Element vorgenommen. Dieser Schritt kann nicht mehr rückgängig gemacht werden.

Um unsere Überlegungen zu einer Iterationsschnittstelle mit der Standardschnittstelle `Iterator` zu verbinden, bietet sich an, eine Unterschnittstelle zu definieren, die unsere drei Methoden als abstrakte, noch zu implementierende Methoden vorgibt, die Methoden der Standardschnittstelle mit Hilfe der abstrakten Methoden aber bereits default-Methoden implementiert.

Wir definieren zunächst die drei bereits bekannten abstrakten Methoden:

PmtIterator.java

```
package name.panitz.pmt.iteration;
import java.util.Iterator;

public interface PmtIterator<A> extends Iterator<A> {

  boolean schleifenTest();
  void schleifeWeiterschalten();
  A schleifenWert();
```

Nun können wir mit diesen die Methoden der Standardschnittstelle `Iterator` umsetzen. Beginnen wir mit der Methode `next()`:

PmtIterator.java

```
public default A next(){
  A result = schleifenWert();
  schleifeWeiterschalten();
  return result;
}
```

Man sieht genau die Bedeutung der Methode `next()` der aktuelle Schleifenwert wird zurück gegeben. Zusätzlich wird der Iterator weiter geschaltet, so dass ein nächster Aufruf das darauffolgende Iterationselement zurück gibt.

Die Methode `schleifenTest()` entspricht exakt der Methode `hasNext()` der Standardschnittstelle. Deshalb können wir in der Implementierung unsere Methode direkt aufrufen.

PmtIterator.java

```
  public default boolean hasNext(){
    return schleifenTest();
  }
}
```

Um wieder an unseren konkreten Iterator über einen bestimmten Zahlenbereich zu kommen, können wir jetzt eine implementierende Klasse dieser Schnittstelle schreiben. Diese implementiert dann automatisch auch die Standardschnittstelle `Iterator` durch die geerbten default-Methoden.

Um noch ein wenig flexibler zu werden, sei noch ein drittes Feld der Klasse zugefügt, dass die Schrittweite beim Weiterschalten des Iterators speichert.

IntegerRangeIterator.java

```
package name.panitz.pmt.iteration;
import java.util.Iterator;

public class IntegerRangeIterator implements PmtIterator<Integer>{
  int from;
  int to;
  int step;

  IntegerRangeIterator(int from, int to, int step){
    this.from = from;
    this.to = to;
    this.step = step;
  }
  public boolean schleifenTest(){
    return from <= to;
  }
  public void schleifeWeiterschalten(){
    from = from + step;
  }
  public Integer schleifenWert(){
    return from;
  }
```

Damit können wir jetzt in der standardmäßig in Java empfohlenen Art und Weise mit dem Iterationsobjekt über die Elemente mit einer for-Schleife iterieren. Der Iterator wird initialisiert und seine Methode `hasNext()` zum Schleifentest benutzt. Das Weiterschalten wird nicht im Kopf der for-Schleife vorgenommen, sondern als erster Befehl im Rumpf, indem dort die Methode `next()` als erste Anweisung aufgerufen wird.

IntegerRangeIterator.java

```
  public static void main(String[] args){
    for (Iterator<Integer> it = new IntegerRangeIterator(0,10,2)
        ; it.hasNext()
        ;
        ) {
      int i = it.next();
      System.out.println(i);
    }
  }
}
```

Aber die Standardschnittstelle `Iterator` hat in ähnlicher Weise wie unsere erste Schleifenschnittstelle eine Methode, um den Iterator laufen zu lassen. Bei uns hieß diese Methode schlicht `run`. In der Standardschnittstelle `Iterator` findet sich folgende default Methode:

```
default void forEachRemaining(Consumer<? super E> action)
```

Somit lässt sich auch ein Iterator durch einen einzigen Methodenaufruf komplett durchlaufen:

IntegerRangeForEach.java

```
package name.panitz.pmt.iteration;
public class IntegerRangeForEach{
  public static void main(String[] args){
    new IntegerRangeIterator(0,10,2)
     .forEachRemaining(i -> System.out.println(i));
  }
}
```

1.4 Die Schnittstelle `Iterable`

Bisher haben wir definiert, was unter Iterator-Objekten zu verstehen ist. In Javas Standard-API gibt es eine weitere Schnittstelle, die sich mit dem Konzept der Iteration beschäftigt. Es handelt sich dabei um die Schnittstelle `Iterable`. Diese liegt nicht im Paket `java.util` sondern im Standardpaket `java.lang`. Sie muss also nicht explizit importiert werden, wenn man sie benutzen will. Auch die Schnittstelle `Iterable` ist generisch. Sie soll ausdrücken, dass ein Objekt iterierbar ist, in dem Sinne, dass es ein Iterator-Objekt gibt, mit dessen Hilfe über die Elemente des Objekts iteriert werden kann.

Deshalb kommt die Schnittstelle `Iterable` mit einer einzigen abstrakten Methode aus. Die Methode heißt `iterator()` und ist dazu gedacht, das Iteratorobjekt für die Klasse zu erfragen. Typische Beispiele sind hierbei natürlich die klassischen Sammlungsklassen, wie Listen und Mengen. Diese implementieren alle die Schnittstelle `Iterable`.

Bleiben wir zunächst bei unserem durchgängigen Beispiel. Statt jetzt direkt die Iteratorklasse für einen Zahlenbereich zu definieren, können wir zunächst eine Klasse definieren, die nur einen Zahlenbereich beschreibt. Wir lassen sie aber die Schnittstelle `Iterable` implementieren. Erst der Aufruf der Methode `iterator()` erzeugt dann ein entsprechendes Iterator-Objekt.

IntegerRange.java

```
package name.panitz.pmt.iteration;
import java.util.Iterator;

public class IntegerRange implements Iterable<Integer>{
  int from;
  int to;
  int step;

  public IntegerRange(int from, int to, int step){
    this.from = from;
    this.to = to;
    this.step = step;
  }
```

Um diese Klasse noch etwas flexibler benutzen zu können, sehen wir zwei weitere Konstruktoren vor. Diese setzen die Schrittweite und die obere Schranke auf Standardwerte:

IntegerRange.java

```
public IntegerRange(int from, int to){
  this(from, to, 1);
}
public IntegerRange(int from){
  this(from, Integer.MAX_VALUE, 1);
}
```

Die Klasse zum Iterieren über einen Zahlenbereich haben wir bereits entwickelt. Diese kann nun für die Methode `iterator()` benutzt werden.

IntegerRange.java

```
public java.util.Iterator<Integer> iterator(){
  return new IntegerRangeIterator(from,to,step);
}
```

Java hat eine syntaktische Besonderheit für Objekte, die die Schnittstelle `Iterable` implementieren, eingeführt. Eine besondere Art der Schleife, die als for-each Schleife bezeichnet wird. Diese hat im Schleifenrumpf zwei Komponenten, die durch einen Doppelpunkt getrennt werden. Nach dem Doppelpunkt steht das iterierbare Objekt, vor dem

Doppelpunkt wird die Variable definiert, an der in jedem Schleifendurchlauf das aktuelle Element gebunden ist.

Für unser Beispiel erhalten wir dann die folgende kompakte Schleife:

IntegerRange.java

```
public static void main(String[] args){
  IntegerRange is = new IntegerRange(0,10,2);
  for (int i: is){
    System.out.println(i);
  }
```

Gelesen wird dieses Konstrukt als: *Für alle Zahlen i aus dem iterierbaren Objekt is führe den Schleifenrumpf aus.*

So wie unsere eigene erste Iterierschnittstelle eine Methode `run` hatte, enthält die Schnittstelle `Iterable` auch eine Methode, um einmal durch alle Elemente zu iterieren. Sie hat den Namen `forEach`. Mit ihr lässt sich also die Schleife komplett vermeiden und durch einen einzigen kurzen Methodenaufruf ersetzen:

IntegerRange.java

```
is.forEach(i->System.out.println(i));
```

Im Vergleich hierzu, sei hier noch einmal die entsprechende Schleife ohne Verwendung der for-each Schleife oder die Methode `forEach` geschrieben. In diesem Fall wird explizit nach dem Iterator-Objekt gefragt.

IntegerRange.java

```
    for (Iterator<Integer> it = is.iterator(); it.hasNext(); ) {
      int i = it.next();
      System.out.println(i);
    }
  }
}
```

1.4.1 Iterator als Innere Klasse

Da in der Regel die Iteratorklasse fest an der Klasse gebunden ist, über deren Elemente iteriert werden soll, bietet sich an, die Iteratorklasse zu verstecken. In Java kann hierfür eine innere Klasse benutzt werden. Mit einem Sichtbarkeitsattribut `private` ist dann die Iteratorklasse komplett versteckt:

Wir sehen also zunächst die Klasse mit den notwendigen Feldern für die Beschreibung des Iterationsbereichs vor:

IntegerRange.java

```
package name.panitz.pmt.util;
import java.util.Iterator;

public class IntegerRange implements Iterable<Integer>{
  int from;
  int to;
  int step;

  public IntegerRange(int from, int to, int step){
    this.from = from;
    this.to = to;
    this.step = step;
  }

  public IntegerRange(int from, int to){
    this(from, to, 1);
  }
  public IntegerRange(int from){
    this(from, Integer.MAX_VALUE, 1);
  }
```

Für diese Klasse kann ein Iterator-Objekt erzeugt werden:

IntegerRange.java

```
public Iterator<Integer> iterator(){
  return new IntegerRangeIterator(from,to,step);
}
```

Das Iterator-Objekt ist dabei von einer inneren Klasse, die beschreibt, wie durch die Elemente iteriert wird:

IntegerRange.java

```
  private static class IntegerRangeIterator implements Iterator<Integer>{
    int from;
    int to;
    int step;

    IntegerRangeIterator(int from, int to, int step){
      this.from = from;
      this.to = to;
      this.step = step;
    }
    public boolean hasNext(){
      return from < to;
    }
    public Integer next(){
      int result  = from;
      from = from + step;
      return result;
    }
  }
}
```

1.4.2 Verknüpfung zweier iterierbarer Objekte

In diesem Abschnitt werden wir aus zwei iterierbaren Objekten eines machen. Das neue Objekte soll erst durch die Elemente des einen iterierebaren Objekts und anschließend durch die Elemente des zweiten iterieren. Es ist also ein Aneinanderhängen der beiden Iterables. Fast wie bei Listen, die aneinander gehängt werden.

Hierzu sei die Klasse `Append` entwickelt. Objekte dieser Klasse sollen aus zwei iterierbaren Objekten ein neues erzeugen. Somit ist sie als generische Klasse, die `Iterable` implementiert, entworfen:

Append.java

```
package name.panitz.pmt.util;
import java.util.Iterator;

public class Append<E> implements Iterable<E> {
```

Die Klasse `Append` soll zwei iterierbare Objekte zu einem neuen verbinden. Hierzu braucht sie Felder, um die beiden zu verkknüpfenden Objekte zu speichern:

Append.java

```
Iterable<E> xs;
Iterable<E> ys;
```

In gewohnter Weise ist ein Konstruktor vorzusehen, der diese Felder initialisiert.

Append.java

```
public Append(Iterable<E> xs, Iterable<E> ys) {
  this.xs = xs;
  this.ys = ys;
}
```

Es muss schließlich eine Methode geben, die den neuen Iterator zurück gibt, der durch beide Objekte iteriert.

Append.java

```
@Override
public Iterator<E> iterator() {
  return new MyIt();
}
```

Die hier instantiierte Klasse `MyIt` wird als innere Klasse der Klasse `Append` realisiert:

Append.java

```
public class MyIt implements Iterator<E> {
```

Der neue Iterator muss die beiden ursprünglichen Iteratoren kennen. Daher werden zwei Felder für diese vorgesehen. Initialisiert werden die Felder mit den Iteratoren, die die beiden zu verknüpfenden iterierbaren Objekte der äußeren Klasse `Append` zurück geben:

Append.java

```
Iterator<E> xsIt = xs.iterator();
Iterator<E> ysIt = ys.iterator();
```

Es bleiben die Methoden der Schnittstelle `Iterator`, die implementiert werden müssen. Es stellt sich die Methode `hasNext()` als besonders einfach heraus. Der neue Iterator hat noch weitere Elemente, wenn einer der beiden ursprünglichen Iteratoren noch ein nächstes Element hat.

Append.java

```
@Override
public boolean hasNext() {
  return xsIt.hasNext() || ysIt.hasNext();
}
```

Wo holt der neue Iterator sein nächstes Element her? Wenn es noch weitere Elemente im ersten Iterator gibt, dann aus diesem, ansonsten aus dem zweiten Iterator.

Append.java

```
    @Override
    public E next() {
      if (xsIt.hasNext()) return xsIt.next();
      return ysIt.next();
    }
  }
}
```

In einer kleinen Testklasse soll die Klasse `Append` an zwei Beispielen ausprobiert werden.

AppendTest.java

```
package name.panitz.pmt.util;
```

Einer der beiden Beispielaufrufe soll mit Standardlisten gemacht werden, so dass diese importiert werden.

AppendTest.java

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import name.panitz.pmt.iteration.IntegerRange;
```

Im ersten Beispielaufruf sollen zwei iterierte Zahlenbereiche nacheinander iteriert werden:

AppendTest.java

```
public class AppendTest{
  public static void main(String[] args) {
    for (int i : new Append<>
                              ( new IntegerRange(1,15)
                              , new IntegerRange(47,52)) ) {
      System.out.println(i);
    }
```

Im zweiten Beispielaufruf werden als iterierbare Objekte zunächst Listen erzeugt:

AppendTest.java

```
List<String> xs = new LinkedList<>();
xs.add("Freunde");
xs.add("Roemer");
xs.add("Landsleute");

List<String> ys = new ArrayList<>();
ys.add("leiht");
ys.add("mir");
ys.add("Euer");
ys.add("Ohr");
```

Diese beiden Listen lassen sich nun auch zu einem neuen iterierbaren Objekt mit `Append` verbinden:

AppendTest.java

```
    new Append<>(xs,ys)
        .forEach( s -> System.out.println(s.toUpperCase()));

  }
}
```

1.5 Funktionen als Parameter

In diesen Abschnitt werden wir die Schnittstelle `Iterable` um einige mächtige Funktionalität erweitern. Bisher hatte die Schnittstelle `Iterable` nur die einzige Methode `iterator()`, mit der ein Iterator-Objekt erzeugt wurde, das zum einmaligen Iterieren durch die Elemente eines Objekts benutzt werden konnte.

Zwei neue Konzepte bringt Java 1.8 mit, die es ermöglichen, die Schnittstelle `Iterable` um wesentliche Funktionalität zu erweitern und ihre eine noch zentralere Rolle zukommen zu lassen.

- Mit Java 1.8 können Schnittstellen, wie bisher nur von abstrakten Klassen her bekannt, auch Methoden bereits konkret implementieren, wobei sie die abstrakten Methoden bereits benutzen können.
- Mit Java 1.8 wurden sogenannte Lambda-Ausdrücke, die auch als Closures bezeichnet werden, sowie auch Methodenobjekte der Sprache Java hinzugefügt.

Mit diesen neuen Sprachkonzepten wird eine bisher in Java nur schwer realisierbare Programmiermethodik möglich. Die sogenannte Programmierung höherer Ordnung. Dabei werden Methoden oder auch Codeblöcke als Parameter anderen Methoden übergeben. Dieses ermöglicht eine wesentlich stärkere Abstraktion in der Parameterisierung des Codes. Es kann jetzt über eine ganze Befehlsfolge auf einfache Weise abstrahiert werden.

Die Schnittstelle `Iterable` hat weiterhin nur einen abstrakte Methode, die Methode `iterator()`, die wir bereits kennen. Zusätzlich sind eine ganze Menge weiterer Methoden implementiert. Bei diesen handelt es sich zum großen Teil um Methoden höherer Ordnung.

1.5.1 Schleifen als Methoden

Die erste neue Methode höherer Ordnung der Schnittstelle `Iterable`, die wir betrachten, ist die Methode `forEach`. Sie hat ein Argument. Dieses ist ein Parameter vom neuen Typ Block. Es handelt sich dabei um einen belieben Code-Block, der abhängig ist von einer Variablen. Hierfür gibt es eine neue Syntax. Es wird die Variable (eine in diesem Kontext neu deklarierte Variable) in runde Klammern geschrieben. Anschließend folgt ein aus einem Minuszeichen und dem Größerzeichen gebildeter Pfeil. Danach kommt in geschweifte Klammern ein beliebiger Code-Block.

Damit ist es möglich, die for-Schleife komplett als Methodenaufruf darzustellen:

ForEachExample.java

```
package name.panitz.pmt.iteration;
public class ForEachExample{
  public static void main(String[] args){
    IntegerRange is = new IntegerRange(0,10,1);

    is.forEach( (x) -> {System.out.println(x);});
  }
}
```

Die Zeile `is.forEach( (x) -> {System.out.println(x);})` ist wie folgt zu lesen:

Für jedes Element x im iterierbaren Objekt *is* führe den Code-Block aus, der aus der einzigen Anweisung `println` besteht.

1.5.2 Standard- (default) Methoden in Schnittstellen

Mit Java 1.8 ist es jetzt möglich, ähnlich wie in abstrakten Klassen bestimmte Methoden bereits auszuimplementieren und dabei bereits auf die noch abstrakten Methoden zuzugreifen. Ein Beispiel hierfür haben wir bereits gesehen. In der Schnittstelle `Iterable` gibt es nun die Methode `forEach`. Diese ist bereits in der Schnittstelle konkret implementiert. Beim Implementieren der Schnittstelle brauchten wir diese nicht selbst zu implementieren. Wir haben sie quasi geschenkt bekommen.

Mit Java 1.8 wird auch in der Schnittstelle `Iterator` eine Standardmethode eingeführt. Die bisher abstrakte aber in der Dokumentation als optional beschriebene Methode `remove` hat von nun an eine Standardimplementierung und braucht beim Implementieren der Schnittstelle nicht mehr selbst implementiert werden.

Wie man sieht, könnte man mit dieser Methode fast auf die Schleifen in Java verzichten. Tatsächlich gibt Es Programmiersprachen, die diesen Weg gegangen sind und keine speziellen Schleifenkonstrukte kennen.

1.5.3 `Iterable` als verzögerte Listen

Man kann mit den default-Methoden jetzt soweit gehen, dass man gleich die Schnittstelle `Iterable` soweit aufmotzt, dass sie bereits eine einfach verkettete Liste darstellt. Hierzu braucht sie im Prinzip fünf Methoden:

- eine Testmethode `empty()`, die testet, ob die Liste leer ist.
- eine Methode `head()`, die das erste Element zurück gibt.
- eine Methode `tail()`, die die Liste ohne das erste Element zurück gibt.
- eine Methode `nil()` zum Erzeugen einer leeren Liste.
- eine Methode `cons(x,xs)`, die eine neue Liste erzeugt, bestehend aus einem neuem ersten Element und einer Restliste.

Wir können tatsächlich eine eigene Schnittstelle `Iterable` schreiben, die diese fünf Methoden hat. Hierzu schreiben wir eine Unterschnittstelle von `Iterable` mit den entsprechenden dafault-Methoden:

Iterable.java

```
package name.panitz.util;
import java.util.Iterator;
import java.util.function.*;
public interface Iterable<A> extends java.lang.Iterable<A>{
  public Iterator<A> iterator();
```

Die Testmethode, ob es sich um eine leere Liste handelt, lässt sich einfach implementieren. Man lässt sich einen Iterator geben und fragt diesen, ob er ein nächstes Element hat.

Iterable.java

```
public default boolean empty(){
  return !iterator().hasNext();
}
```

Die Methode `head` ist einfach zu implementieren. Es muss das `Iterable`-Objekt nach seinem Iterator gefragt werden und von diesem das erste Element zurück gegeben werden.

Iterable.java

```
public default A head(){
   return iterator().next();
}
```

Auch die Methode `tail` ist einfach zu implementieren. Es wird zunächst der Iterator erfragt. Dieser wird durch einmaliges Aufrufen von `next()` auf das zweite Element geschaltet. Nun ist es der Iterator für das neue `Iterable`.

Iterable.java

```
public  default Iterable<A> tail(){
    return ()->{
         Iterator<A> result = iterator();
         result.next();
         return result;
    };
}
```

Aufmerksamen Beobachtern mag aufgefallen sein, dass sich mit den Methoden `empty` und `tail` bereits viele typische Methoden für Listenstrukturen umsetzen lassen. So zum Beispiel die Berechnung der Länge:

Iterable.java

```
public  default int length(){
  return empty() ? 0 : 1+tail().length();
}
```

`cons` und `nil` für `Iterables`

Wir brauchen die Methode `nil()` die uns eine `Iterable`-Objekt erzeugt, dessen Iterator über kein einziges Element iteriert. Es ist also ein leeres `Iterable`-Objekt. Deshalb gibt die Methode `hasNext` immer `false` zurück.:

Iterable.java

```
public static <A> Iterable<A> nil(){
  return ()->{
      return new Iterator<A>(){
        public boolean hasNext(){return false;}
        public A next(){
          throw new
          ↪  java.util.NoSuchElementException("next for empty iterator");
        }
      };
  };
}
```

Als Objektmethode schreiben wir als nächstes die Methode `cons`, diese erhält ein Element, welches bei der resultierenden Iteration als erstes Element zurück gegeben werden soll. Es wird also an den Iterationsbereich vorne noch ein Element angehängt.

Iterable.java

```
public default Iterable<A> cons(final A hd){
  final Iterable<A> tl = this;
  return () -> {
    final java.util.Iterator<A> it = tl.iterator();
    final A first = hd;
    return new Iterator<A>(){
      boolean firstCall = true;
      public boolean hasNext(){return firstCall||it.hasNext();}
      public A next(){
        if (firstCall){
          firstCall = false;
          return first;
        }else {
          return it.next();
        }
      }
    };
  };
}
```

Als eine statische Methode wird noch eine Variante dieser Methode implementiert. Diese Variante erhält das erste Element für die Iteration, sowie ein iterierbares Objekt für die übrigen Elemente. Dieses zweite iterierbare Objekt wird allerdings nicht direkt übergeben, sondern über ein `Supplier`-Objekt. Dieses hat genau eine Methode namens `get`, mit der das Objekt erfragt werden kann, das der Supplier liefert.

Iterable.java

```
public static <A> Iterable<A> cons(A hd,final
↪  java.util.function.Supplier<Iterable<A>> tl){
  return ()->{
    final A first = hd;
    return new Iterator<A>(){
      java.util.Iterator<A> it = null;
      boolean firstCall = true;

      public boolean hasNext(){
        if (firstCall) return true;
        if (it==null) it=tl.get().iterator();
        return it.hasNext();
      }

      public A next(){
        if (firstCall){
          firstCall = false;
          return first;
        }else {
          if (it==null) it=tl.get().iterator();
          return it.next();
        }
      }
    };
  };
}
```

Filter für die Iteration

Mit den oben geschriebenen Methoden `head`, `tail` und `cons` lassen sich jetzt auf einfache Weise weitere sinnvolle Methoden für iterierbare Objekte schreiben. Zum Beispiel ein einfacher Filter. Es wird hierzu ein Prädikat übergeben, dass für ein einzelnen Element zu wahr oder falsch auswertet. Es soll ein neues iterierbares Objekt erzeugt werden, das nur noch als den Objekten des ursprünglichen existiert, für die das Prädikat zu wahr auswertet.

Es gibt drei Fälle:

> für ein leeres iterierbares Objekt ist auch das Ergebnis ein solche leeres Objekt. dann ist zu testen, ob das erste Element das Prädikat erfüllt. Wenn ja, dann wird dieses als erstes Element des Ergebnisses mit `cons` auf den rekursiven Aufruf auf den Reste zugefügt. ansonsten wird die Methode nur rekursiv auf die übrigen Elemente des Iterationsbereiches angewendet.

Iterable.java

```
public default Iterable<A> filter(Predicate<A> p){
  if (empty()) return this;
  if (p.test(head())) return cons(head(),()->tail().filter(p));
  return tail().filter(p);
}
```

Elemente Abbilden

Eine einfache und vielfach in Sprachen, die Lambda-Ausdrücke unterstützen, verwendete Funktion, generiert aus einem Iterationsbereich mit Elementen eines bestimmten Typs einen neuen Iterationsbereich ,der entsteht, indem eine Funktion auf jedes Element angewendet wird. Es ist also wieder eine Form, der Anwendung für alle Elemente. Hier wird für jedes Element in ein neues Element mit Hilfe einer übergebenen Funktion generiert.

Iterable.java

```
public default <B> Iterable<B> map(Function<A,B> f){
  if (empty()) return nil();
  return cons(f.apply(head()),()->tail().map(f));
}
```

Faltungen

Eine mächtige Funktion für Iterationsbereiche sind sogenannte Faltungen (in vielen Bibliotheken heißen die entsprechenden Funtionen *fold*). Bei Faltungen geht es darum, die Elemente alle mit einer Operation zu verknüpfen. Hierzu wird eine Operation benötigt, die zwei Elemente verknüpfen kann. Zusätzlich wird ein Startwert erwartet, der insbesondere als Ergebnis benutzt wird, wenn keine Elemente in der Iteration sind. Für eine Funktion $\otimes$ als Operation in Infix-Schreibweise sei die Faltung wie folgt definiert:

$$reduce([x_1, x_2, \ldots, x_n], start, \otimes) = start \otimes x_1 \otimes x_2 \otimes \cdots \otimes x_n$$

Die Implementierung dieser Funktion lässt sich wunderbar per Iteration machen. Dazu werden die Elemente in einer Schleife nacheinander mit dem Startwert verknüpft.

Iterable.java

```
public default <B> B reduce(B start, BiFunction<B,A,B> op){
  for (A x:this) start = op.apply(start,x);
  return start;
}
```

So kurz und einfach diese Funktion wirkt, umso mächtiger ist sie, was auf den ersten Blick nicht so ersichtlich ist. Ein paar Beispiele für weitere Funktionen, die sich mit reduce umsetzen lassen. Im Prinzip ist es die Verallgemeinerung einer Iteration, bei der nacheinander die Elemente der Iteration auf ein Ergebnis akkumuliert werden. Dabei ist der Parameter start jeweils die Initialisierung der Ergebnisvariablen.

So lässt sich zum Beispiel eine Funktion schreiben, die testet, ob eines der Elemente im Iterationsbereich ein bestimmtes Prädikat erfüllt. Diese Funktion entspricht dem logischen Existenzquantor. Der Startwert ist in diesem Fall false. Es wird für jedes Element getestet, ob es das Prädikat erfüllt und schließlich die Teilergebnisse mit Oder verknüpft.

Iterable.java

```
public default boolean exists(Predicate<A> p){
  return  reduce(false,(x,y)-> x || p.test(y));
}
```

Analog funktioniert die Funktion, die testet, ob ein Prädikat für alle Elemente des Iterationsbereichs zutrifft. Diese Funktion entspricht dem logischen Allquantor.

Iterable.java

```
public default boolean forAll(Predicate<A> p){
  return  reduce(true,(x,y)-> x && p.test(y));
}
```

So lässt sich wiederum mit der Funktion exists schnell testen, ob ein bestimmtes Element in einem Iterationsbereich enthalten ist.

Iterable.java

```
public default boolean contains(A el){
  return  exists(x -> x.equals(el));
}
```

Es lassen sich auch wie bei einer Methode toString die Stringdarstellungen aller Elemente in einem String zusammenfügen:

Iterable.java

```
public default String stringRepresentation(){
  return  reduce("",(x,y) ->  x+" "+y);
}
```

Eine entsprechende Funktion, die ein Objekt der Klasse StringBuffer nutzt, um die Elemente zu einem Gesamtstring zu akkumulieren, lässt sich auch mit Hilfe von reduce formulieren:

Iterable.java

```
public default String show(){
  return  reduce(new StringBuffer(),
    (buf,y) -> {
      buf.append(" ");
      buf.append(y);
      return buf;
    }).toString();
}
```

Auch die Längenberechnung lässt sich mittels reduce formulieren. Als Operation wird immer die Konstante 1 hinzuaddiert:

Iterable.java

```
public default int length2(){
  return  reduce(0, (result,x) -> result+1);
}
```

Es lässt sich sogar die Funktion filter aus diesem Abschnitt nun komplett mit Hilfe von reduce darstellen:

Iterable.java

```
public default Iterable<A> filter2(Predicate<A> p){
  return  () -> (reduce(new java.util.ArrayList<A>(),
    (result,y) -> {
      if (p.test(y)) result.add(y);
      return result;
    }).iterator());
}
```

Gleiches gilt auch für die Funktion map, die sich mit Hilfe von reduce formulieren lässt.

Iterable.java

```
public default <B> Iterable<B> map2(Function<A,B> f){
  return  () -> (reduce(new java.util.ArrayList<B>(),
    (result,y) -> {
      result.add(f.apply(y));
      return result;
    }).iterator());
}
```

Ebenso lässt sich für einen größer Vergleich mittels eines Comparator-Objekts mit reduce das Maximum der Elemente eines Iterationsbereichs ermitteln.

Iterable.java

```
  public default A max(java.util.Comparator<A> comp){
    return  reduce(null,
        (result,y) -> {
          if (null==result) return y;
          if (comp.compare(result,y) > 0) return result;
          return y;
        }
      );
  }
}
```

Sieb des Eratosthenes

Mit der obigen Filter-Funktion lässt sich eine iterierbares Objekt schreiben, dass über alle Primzahlen iteriert. Hierzu kann das bekannte Verfahren des Siebs des Eratosthenes angewendet werden. Gestartet wird mit der Folge von Zahlen beginnend von 2.

PrimesExample.java

```
package name.panitz.pmt.util;
import name.panitz.util.Iterable;
public class PrimesExample{
  public static Iterable<Integer> sieve(final Iterable<Integer> it){
     final int first = it.head();
     return Iterable.cons(first,()->sieve(it.filter( (x)-> x%first != 0
     ↪  )));
  }

  public static void main(String[] args){
    Iterable<Integer> is = ()->new IntegerRange(2).iterator();
    Iterable<Integer> primes = sieve(is);
    primes.forEach(x->System.out.println(x));
  }
}
```

1.6 Parallel Iterieren mit `Streams` und `Spliterator`

Mit Java 8 wurde eine neue Art von Iteratoren eingeführt, die als Grundlage der der Ströme (*Streams*) dienen. Die Grundfunktionalität ist wieder dieselbe, wie schon für Iteratoren: es werden nach und nach Elemente geliefert. Die zusätzliche Funktionalität ist, dass es möglich sein soll, die Iteration wenn möglich zu verteilen auf mehrere nebenläufige Iterationen.

Dieses neue API findet sich im Paket `java.util.streams`[1]. Hier übernimmt die Rolle der Schnittstelle `Iterator` die neue Schnittstelle `Spliterator` und die Rolle der Schnittstelle `Itrable` die neue Schnittstelle `Stream`.

Anders als bei der Schnittstelle `java.util.Iterator` gibt es in `Spliterable` nicht zwei Methoden, die jeweils prüfen, ob es ein neues Element gibt (`hasNext`), bzw. dieses Element liefern (`next`), sondern nur eine Methode für beide Aufgaben. Diese Methode hat wie `hasNext` eine Wahrheitswert als Rückgabe, der angibt, ob es noch ein weiteres Element gegeben hat. Gleichzeitig führt diese Methode eine Aktion auf, falls es ein weiteres Element gegeben hat.

Die Ströme in Java 8 benutzen exzessiv Methoden höherer Ordnung, d.h. Methoden, die Parameter einer funktionalen Schnittstelle haben. Damit können diesen Methoden per Lambda-Ausdrücke die Argumente übergeben werden.

1.6.1 Die Schnittstelle `java.util.Spliterator`

Die zentrale neue Schnittstelle, die die Rolle von `Iterator` übernimmt, heißt `Spliterator`. In diesen Abschnitt soll diese Schnittstelle mit dem bekannten Beispiel eines Integerbereichs zum Iterieren veranschaulicht werden. Wir beginnen wie auch bei den `Iterator`-Beispielen mit einer Klasse `Range`, die einen Startwert, einen Endpunkt und einen Iterationsschritt enthält und damit wieder die klassischen drei Elemente einer for-Schleife hat. Statt der Schnittstelle `Iterator` soll jetzt die Schnittstelle `Spliterator` implementiert werden:

Range.java

```
package name.panitz.util.streams;

import java.util.Spliterator;
import java.util.function.Consumer;


public class Range implements Spliterator<Integer> {
  int i;
  int to;
  int step;

  public Range(int from, int to, int step) {
    this.i = from;
    this.to = to;
    this.step = step;
  }
```

In der Schnittstelle `Spliterator` existiert nur eine Methode, die alle Teile einer klassischen Schleife übernimmt, die Method `tryAdvance` mit folgender Signatur:

[1] Achtung, es hat nichts mit den *Streams* aus dem Paket `hava.io` zu tun.

`boolean tryAdvance(Consumer<? super Integer> action),`

Die Methode hat einen Rückgabewert, der wie `hasNext` anzeigt, ob es noch ein weiteres Element für die Iteration gegeben hat. Wenn die Methode `true` als Ergebnis liefert, dann sind drei Dinge erfolgt:

> Das nächste Element der Iteration wurde erfragt. Mit diesem Element wurde die als Parameter übergebene Aktion ausgeführt. Dieses entspricht in einer klassischen Schleife dem Schleifenrumpf. Der Iterator wurde intern weiter geschaltet, um beim nächsten Aufruf ein Element weiter zu liefern.

Der Parameter dieser Methode entspricht also der Aktion, die in einer Schleife im Rumpf für das aktuelle Iterationsobjekt durchgeführt wird. Betrachten wir die ursprüngliche Schleife eines Iterators der Form:

.

```
for (Iterator<A> it = someIterator; it.hasNext();){
  A x = it.next();
  doSomethingSmartWithElement(x);
}
```

Diese würde mit einem Spliterator wie folgt funktionieren:

.

```
while (someSplitertor.tryAdvance(x->doSomethingSmartWithElement(x))){}
```

Mit diesem Hintergrundwissen ergibt sich für den Spliterator eines Zahlenbereichs, folgende Implementierung:

Range.java

```
@Override
public boolean tryAdvance(Consumer<? super Integer> action) {
  if (i<=to){
    action.accept(i);
    i += step;
    return true;
  }
  return false;
}
```

In diesem Fall sind also alle Komponenten einer Schleife in eine Methode gepackt. Die if-Bedingung prüft, ob es noch einen Schleifendurchgang gibt. Ansonsten wird direkt `false` zurück gegeben. Dann wird der Schleifenrump durch die Methode `accept` des übergebenen `Consumer`-Objekts auf die aktuelle Schleifenvariable `i` aufgerufen. Schließlich wird diese um den vorgegebenen Schritt weiter geschaltet. Da es noch ein weiteres Element gab, das verarbeitet wurde, gibt die Methode aus der if-Bedingung schließlich `true` zurück.

Die Schnittstelle `Spliterator` hat aber noch drei weitere abstrakte Methoden. Diese beschäftigen sich alle damit, ob der Spliterator zur nebenläufigen Abarbeitung geteilt werden kann. Die wichtigste Methode ist dabei die Methode `trySplit` mit folgender Signatur:

```
public Spliterator<T> trySplit()
```

Gibt die Methode `null` zurück, so zeigt es an, dass der Spliterator nicht aufgeteilt werden kann und sequentiell abzuarbeiten ist. Wird hingegen ein neues Spliteratorobjekt zurück gegeben, so gibt es dann zwei Spliteratoren: den ursprünglichen und den durch `trySplit` erzeugten. Beide zusammen sollen über alle Elemente des ursprünglichen Spliterators iterieren. Es ändert sich also dann auch der Iterationsbereich des Originalspliterators.

Am deutlichsten wird dieses sicher durch unser Standardbeispiel. Zunächst starten wir mit einer if-Bedigung, die prüft, ob sich ein Aufsplitten des Spliterators auf zwei Objekte noch lohnt. Wenn keine 4 Elemente mehr zu iterieren sind, geben wir einfach `null` zurück.

Range.java

```
@Override
public Spliterator<Integer> trySplit() {
  if (to-i<4*step){
    return null;
  }
```

Spannender ist es natürlich, tatsächlich zu splitten und einen neuen Spliterator zu erzeugen. Hierzu berechnen wir die Mitte des Iterationsbereichs. Setzen den Startwert `i` des aktuellen Spliterators auf diesen Mittelwert und geben einen neuen Spliterator zurück, der vom Startwert `i` bis zu diesen Mittelwert iteriert.

Range.java

```
  int steps = (to-i)/step;
  int middle = (i+steps/2*step);
  int iOld = i;
  i=middle;
  return new Range(iOld, middle-1, step);
}
```

Eine weitere abstrakte Methode der Schnittstelle `Spliterator` soll dazu dienen, abzuschätzen, über wie viele Elemente iteriert wird. Diese Information ist hilfreich zur Planung, ob es sich lohnt die Iteration zu parallelisieren. Es gibt Spliteratoren, die nicht genau wissen, wie viel Elemente sie haben, oder es kann sein, dass die Information zu berechnen zu aufwändig ist. Dann soll die Methode die größtmögliche Zahl zurück geben. Ebenso, wenn der Spliterator über unendlich viele Elemente läuft. Dann ist der Wert `Long.MAX_VALUE` zurück zu geben.

In unserem Beispiel können wir die Gfröße leicht genau berechnen:

Range.java

```
@Override
public long estimateSize() {
  return (to-i)/step;
}
```

Die letzte abstrakte Methode heißt `characteristics`. In einer Zahl codiert werden hier einige Charakteristiken genannt. Die Menge der Charakteristiken beinhaltet. `ORDERED`, `DISTINCT`, `SORTED`, `SIZED`, `NONNULL`, `IMMUTABLE`, `CONCURRENT`, `SUBSIZED`.

Viele dieser Charakteristiken erklären sich vom Namen her. Für die genaue Bedeutung verweisen wir hier auf die Java API Dokumentation.

Die Werte der einzelnen Charakteristiken können bitweise verodert werden. In unserem Beispiel können wir vier Charakteristiken setzen.

Range.java

```
@Override
public int characteristics() {
  return ORDERED | SIZED | IMMUTABLE | SUBSIZED;
}
```

Wir haben einen ersten Spliterator implementiert. Mit der einfachen while-Schleife können wir einmal testweise über alle Elemente iterieren:

Range.java

```
  public static void main(String[] args) {
    while (new Range(1, 100, 2).tryAdvance(x-> System.out.println(x))){};
  }
}
```

Spliteratoren haben noch weitere default-Methoden, die bei Bedarf überschrieben werden können. Auch hier verweisen wir nur auf die Dokumentation.

1.6.2 Ströme

Spliteratoren sind die Grundlage für die Ströme (streams) in Java 8. Ströme kapseln noch einmal Spliteratoren und bieten eine Vielzahl von Operationen, um mit Ihnen zu arbeiten. Ströme bieten einen Iterationsbereich. Sie sind zum einmaligen Gebrauch gemacht. Ein Strom hat drei Phasen seiner Lebenszeit:

> Zunächst muss ein Strom erzeugt werden. Oft wird ein Strom aus einer Datenhaltungsklasse, meist Sammlungsklassen, heraus erzeugt. Es gibt aber auch andere Möglichkeiten Ströme zu erzeugen. Es können viele Transformationen auf einem

Strom durchgeführt werden. Er kann gefiltert werden, die Elemente können auf andere Elemente abgebildet werden, aber auch komplexere Operationen wie eine Sortierung kann durchgeführt werden. Ein Strom muss erst zur Verarbeitung angestoßen werden. Hierzu dienen die terminierenden Methoden auf Strömen. Sie stoßen die eigentliche Iteration an, um ein Endergebnis zu erzielen.

Betrachten wir alle drei Phasen im Einzelnen:

Erzeugen von Strömen

Ströme für Spliteratoren Eine Möglichkeit ist es, einen Strom für ein bereits existierendes Spliterator-Objekt zu erzeugen. Hierzu gibt es in der Hilfsklasse `StreamSupport` eine statische Methode `stream`, die einen Spliterator als Parameter erhält. So können wir für unsere Spliterator-Objekte der Klasse `Range` einen Strom erzeugen.

RangeStream.java

```
package name.panitz.util.streams;

import java.util.stream.StreamSupport;
import java.util.stream.Stream;

public class RangeStream{
  public static void main(String[] args){
    Stream<Integer> is = StreamSupport.stream(new Range(1, 100,
    ↪  2),false);
    is.forEach(x-> System.out.println(x));
  }
}
```

In diesem Beispiel rufen wir testweise bereits die terminierende Methode `forEach` des Stroms auf.

Wie man in diesem Beispiel sieht, hat die Methode `stream` einen zweiten Parameter. dieser zeigt an, ob für den Strom eine nebenläufige Abarbeitung erlaubt ist. Starten Sie einmal die zweite Version dieses Beispiels, in dem diesem Parameter `true` übergeben wird. kommen die Ausgaben weiterhin in aufsteigender Reihenfolge.

ParallelRangeStream.java

```
package name.panitz.util.streams;

import java.util.stream.StreamSupport;

public class ParallelRangeStream{
  public static void main(String[] args){
    StreamSupport
      .stream(new Range(1, 100, 2),true)
      .forEach(x-> System.out.println(x));
  }
}
```

Stromelemente Aufzählen Die vielleicht einfachste Form, einen Strom zu erzeugen, besteht darin, die Stromelemente einfach aufzuzählen. Hierzu gibt es in der Klasse `Stream` de statische Methode `of`. Diese hat eine variable Parameterzahl, es können also beliebig viele Parameter eines in diesem Fall generisch gehaltenen Typ übergeben werden.

StreamOf.java

```
package name.panitz.util.streams;

import java.util.stream.Stream;

public class StreamOf{
  public static void main(String[] args){
    Stream.of("Hallo", "Freund", "Hallo", "Ilja")
      .forEach(x->System.out.println(x.toUpperCase()));
  }
}
```

Ströme aus Generatorfunktion Es gibt zwei Möglichkeiten, wie man unendliche Ströme mit Hilfe einer Generatormethode erzeugen kann:

- Eine Möglichkeit unendliche Ströme zu erzeugen, besteht darin, das erste Element des Stroms zu geben und eine Funktion, die berechnet, wie das nächste Element aus dem aktuellen berechnet wird. Für einen Startwert s und eine Funktion f wird damit folgender Strom erzeugt:

  $$s, f(s), f(f(s)), f(f(f(s))), f(f(f(f(s)))), \ldots$$

  Die entsprechende Methode, auf diese Weise einen Strom zu erzeugen befindet sich in der Klasse `Stream` und heißt: `iterate`.

So lässt sich zum Beispiel auf einfache Weise die Folge der natürlichen Zahlen als Strom erzeugen. Hierzu ist der Startwert die Zahl 1 und die Funktion addiert jeweils 1 auf den aktuellen Wert.

StreamGeneration1.java

```
package name.panitz.util.streams;

import java.util.stream.Stream;

public class StreamGeneration1{
  public static void main(String[] args){
    Stream
      .iterate(1l, x -> x+1l)
      .forEach(x-> System.out.println(x));
  }
}
```

Mit dieser Methode kann auch ein simpler Strom erzeugt werden, der immer wieder ein Element unendlich oft wiederholt. Hierzu ist die zu übergebene Funktion die Identität `(x) -> x`.

StreamGeneration2.java

```
package name.panitz.util.streams;

import java.util.stream.Stream;

public class StreamGeneration2{
  public static void main(String[] args){
    Stream
      .iterate("hallo", x -> x)
      .forEach(x-> System.out.println(x));
  }
}
```

- Eine zwei Möglichkeit, mit Hilfe eines Generators einen Strom zu erzeugen, ist die Methode `generate`

  ```
  static <T> Stream<T> generate(Supplier<T> s)
  ```

  Das Supplier-Objekt liefert über die Methode `get` auch auf wiederholten Aufruf ein Objekt. So kann auch diese Methode dazu verwendet werden, eine unendliche Folge von aufsteigenden Zahlen zu liefern.

StreamGeneration3.java

```
package name.panitz.util.streams;

import java.util.function.Supplier;
import java.util.stream.Stream;

public class StreamGeneration3{
  public static void main(String[] args){
    Stream
      .generate( new Supplier<Long>(){
        long v = 1;
        public Long get(){return v++;}
      })
      .forEach(x-> System.out.println(x));
  }
}
```

Ströme aus Standardsammlungsklassen Die in der Praxis am häufigsten vorkommende Methode, um einen Strom zu erzeugen, dürfte über eine Standard-Sammlungsklasse sein. Alle Sammlungsklassen haben die Methode `stream`, die für die Elemente der Sammlung einen Strom erzeugen.

CollectionStream.java

```
package name.panitz.util.streams;

import java.util.Arrays;
import java.util.Collection;
import java.util.HashSet;


public class CollectionStream{
  public static void main(String[] args){
    Arrays
      .asList("Freunde", "Römer", "Landsleute")
      .stream()
      .forEach(x -> System.out.println(x.toUpperCase()));
```

Die Sammlungsklassen haben eine zweite Methode, um einen Strom zu liefern. Diese gibt einen Strom, der nebenläufig abgearbeitet werden darf.

CollectionStream.java

```
Arrays
  .asList("Freunde", "Römer", "Landsleute")
  .parallelStream()
  .forEach(x -> System.out.println(x.toUpperCase()));
```

CollectionStream.java

```
    Collection<Integer> xs = new HashSet<>();
    xs.add(2);
    xs.add(42);
    xs.add(-678);
    xs.add(1789);
    xs.add(2);
    xs.add(1789);
    xs.add(0);

    xs.parallelStream().forEach(x -> System.out.println(x));
  }
}
```

Spezielle Ströme

Methoden auf Strömen

Terminierende Methoden Terminierende Methoden verbrauchen den Strom. Der Strom ist damit einmal durch iteriert und verbraucht. Der Strom wird beendet und kann nicht mehr weiter verwendet werden.

count - Anzahl der Elemente zählen Die mit Sicherheit einfachste terminierende Methode für Ströme, iteriert den Strom einmal durch, und zählt die Elemente auf. Die entschrechende Methode heißst count:
`long count()`

Es ist zu beachten, dass diese Methode den Strom verbraucht. Einmal durchiteriert, um die Elemente zu zählen, bedeutet, dass der Strom verbraucht wurde. Deshalb werden diese Methoden als *terminierende* Methoden bezeichnet.

forEach Eine übliche Form einen Strom zu verbrauchen, ist die Methode `forEach`, die wir in all den Beispielen bereits aufgerufen haben. Nach Aufruf der Methode `forEach` ist jedes Element des Stroms bereits iteriert worden. Die Methode `forEach` entspricht

den Aufruf einer Schleife die über die Elemente der Iteration läuft und einen Code-Block für jedes Element im Rumpf der Schleife aufruft. Die genaue Signatur lautet:

```
void forEach(Consumer<? super T> action)
```

Der Parameter ist ein Objekt der funktionalen Schnittstelle `Consumer`, dass die Elemente von Typ `T` konsumieren kann, sprich als Parameter der Methode `accept` erhalten kann. Das ist für alle Konsumer vom Elementtyp `T` oder einem Obertyp von `T` der Fall. Deshalb ist der Parameter als `Consumer<? super T>` gesetzt. Der Parametertyp der Methode `accept` kann ein beliebiger Obertyp von `T` sein. Daher `? super T`.

Faltungen Eine mächtige Funktion für Iterationsbereiche sind sogenannte Faltungen (in vielen Bibliotheken heißen die entsprechenden Funtionen *fold*). In Javas Strom-Api heißt die entsprechende Methode `reduce`.

Bei Faltungen geht es darum, die Elemente alle mit einer Operation zu verknüpfen. Hierzu wird eine Operation benötigt, die zwei Elemente verknüpfen kann. Zusätzlich wird ein Startwert erwartet, der insbesondere als Ergebnis benutzt wird, wenn keine Elemente in der Iteration sind. Für eine Funktion $\otimes$ als Operation in Infix-Schreibweise sei die Faltung wie folgt definiert:

$$fold([x_1, x_2, \ldots, x_n], start, \otimes) = start \otimes x_1 \otimes x_2 \otimes \cdots \otimes x_n$$

Die Implementierung dieser Funktion lässt sich wunderbar per Iteration machen. Dazu werden die Elemente in einer Schleife nacheinander mit dem Startwert verknüpft. Faltungen ersetzen dabei Schleifen, die nach dem folgenden Schema gebildet sind:

```
B result = startValue;

for (A x: xs){
  result = someOperation(result, x);
}
return result;
```

Es wird also ein Startwert und eine Operation benötigt. So kurz und einfach diese Funktion wirkt, umso mächtiger ist sie, was auf den ersten Blick nicht so ersichtlich ist. Es istdie Verallgemeinerung einer Iteration, bei der nacheinander die Elemente der Iteration auf ein Ergebnis akkumuliert werden. Dabei ist der Parameter start jeweils die Initialisierung der Ergebnisvariablen.

Bevor wir uns die Methode `reduce` des Standard-APIs anschauen, sei das Rad einmal neu erfunden und wir definieren eine eigene Methode `fold` und ein paar Anwendungen dieser.

FoldExample.java

```
package name.panitz.util;
import java.util.function.*;
import java.util.*;
class FoldExample{

  static <A,B> B fold(Spliterator<A> it,B start, BiFunction<B,A,B> op){

    class Box{B v = start;};
    Box result = new Box();

    while (it.tryAdvance(x->result.v = op.apply(result.v,x))){}

    return result.v;
  }
```

So lässt sich zum Beispiel eine Funktion schreiben, die testet, ob eines der Elemente im Iterationsbereich ein bestimmtes Prädikat erfüllt. Diese Funktion entspricht dem logischen Existenzquantor. Der Startwert ist in diesem Fall false. Es wird für jedes Element getestet, ob es das Prädikat erfüllt und schließlich die Teilergebnisse mit Oder verknüpft.

FoldExample.java

```
public static <A> boolean exists(Spliterator<A> it,Predicate<A> p){
  return  fold(it,false,(x,y)-> x || p.test(y));
}
```

Analog funktioniert die Funktion, die testet, ob ein Prädikat für alle Elemente des Iterationsbereichs zutrifft. Diese Funktion entspricht dem logischen Allquantor.

FoldExample.java

```
public static <A> boolean forAll(Spliterator<A> it,Predicate<A> p){
  return  fold(it,true,(x,y)-> x && p.test(y));
}
```

So lässt sich wiederum mit der Funktion exists schnell testen, ob ein bestimmtes Element in einem Iterationsbereich enthalten ist.

FoldExample.java

```
public static <A> boolean contains(Spliterator<A> it,A el){
  return  exists(it, x -> x.equals(el));
}
```

Es lassen sich auch wie bei einer Methode toString die Stringdarstellungen aller Elemente in einem String zusammenfügen:

FoldExample.java

```
public static <A> String show(Spliterator<A> it){
  return  fold(it,new StringBuffer(),
    (buf,y) -> {
      buf.append(" ");
      buf.append(y);
      return buf;
    }).toString();
}
```

Auch die Längenberechnung lässt sich mittels reduce formulieren. Als Operation wird immer die Konstante 1 hinzuaddiert:

FoldExample.java

```
public static <A> int length(Spliterator<A> it){
  return  fold(it,0, (result,x) -> result+1);
}
```

Ebenso lässt sich für einen größer Vergleich mittels eines Comparator-Objekts mit reduce das Maximum der Elemente eines Iterationsbereichs ermitteln.

FoldExample.java

```
  public static <A> A max(Spliterator<A> it,java.util.Comparator<A>
  ↪  comp){
    return  fold(it,null,
        (result,y) -> {
          if (null==result) return y;
          if (comp.compare(result,y) > 0) return result;
          return y;
        }
      );
  }
}
```

Faltungen im Standard Stream API Ein Blick in die Schnittstelle `Stream` zeigt, dass sich hier mehrere überladene Versionen einer Methode `reduce` befinden. Der Grund dafür, dass verschiedene überladene Versionen angeboten werden, liegt darin, dass auch die Methode `reduce` nach Möglichkeit die Iteration über die Elemente arallelisiert.

- Die einfachste Form der Faltung hat folgende Signatur:
  `T reduce(T identity, BinaryOperator<T> accumulator)`
  Anders als in unserer Methode `fold` gibt es nicht zwei unterschiedliche Typen für die Elemente des Iterationsbereichs und das Ergebnis der Faltung, sondern beide

sind von demselben Typ `T`. Somit ist die Funktion eine Funktion, die zwei Elemente des Typs `T` zu einem neuen Element des Typs `T` verknüpfen.
Der Parameter für den Startwert heißt ganz bewusst `identity`. Es soll ein neutrales Element bezüglich der Operation sein, mit der die Elemente verknüpft werden.

- Eine Version verzichtet auf einen Parameter für den Startwert
  `Optional<T> reduce(BinaryOperator<T> accumulator)`
  Da es keinen Startwert gibt, mit dem das Ergebnis initial initialisiert ist, ist nicht klar, was das Ergebnis sein soll, wenn der Strom gar kein Element zum Iterieren enthält. Daher ist hier der Ergebnistyp nicht vom Typ `T` sondern vom Typ `Optional<T>`. Dieser Typ enthält entweder ein Element vom Typ `T` oder zeigt an, dass es kein Element gibt.
- Die komplette Version benötigt drei Parameter:
  `<U> U reduce\\`(U identity
  , BiFunction<U,? super T,U> accumulator
  , BinaryOperator<U> combiner)+
  Diese Version entspricht am nächsten unserer Methode `fold`. Sie hat allerdings noch einen dritten Parameter. Dieser wird benötigt, damit die Abarbeitung der Methode auch parallelisiert werden kann.

Transformierende Methoden

filter

distinct - Doppelte Eliminieren

Abbilden der Elemente auf andere Elemente

Teilstrom aus ersten Elementen

Beispiele

Sieb des Eratosthenes Im Modul Algorithmen und Datenstrukturen haben Sie den Algorithmus *Sieb des Eratosthenes* kennen gelernt. Dieser lässt sich auf einfache Weise mit Strömen realisieren, so dass ein unendlicher Strom aller Primzahlen erzeugt wird:

Eratosthenes.java

```java
package name.panitz.streams.example;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class Eratosthenes {
  public static void main(String[] args) {
    //wir starten mit dem Strom aller Zahlen ab 2
    Stream<Long> xs = Stream.iterate(2l, x -> x + 1);

    //dieser Strom wird gesiebt. Alle Zahlen mit Teiler
    //werden herausgefiltert.
    sieb(xs).forEach(x -> System.out.println(x));
  }

  static Stream<Long> sieb(Stream<Long> xs) {
    //Diese interne Liste hält die bisher gefundenen Primzahlen
    List<Long> primes = new ArrayList<>();

    //Der Eingabestrom wird gefiltert:
    return xs.filter(x -> {
      //nur Zahlen, die durch keine  bisherige Primzahl teilbar ist.
      boolean result = primes.parallelStream()

          //Teste für alle bisherigen primes, ob sie kein Teiler sind.
          .map(y -> x % y != 0)

          //alle dürfen keinen Teiler haben.
          //Logisches UND über alle Wahrheitswerte
          .reduce(true, (a, b) -> a && b, (a, b) -> a && b);

      //eine neue Primzahl für spätere Tests
      if (result)primes.add(x);

      return result;
    });
  }
}
```

Filter auf BigInteger Das folgende Beispiel gibt eine Lösung für eine Aufgabe des Kollegen Reith im Modul *Diskrete Strukturen*. Gesucht ist die kleines Zahl n für die gilt:

$$\gcd((n+1)^5+5, n^5+5) > 1$$

Die entsprechende Suche im Hauruckverfahren lässt sich knapp mit Strömen implementieren:

BigInt8.java

```
package name.panitz.streams.example;

import java.math.BigInteger;
import java.util.stream.Stream;
class BigInt8{
  static final BigInteger FIVE = new BigInteger("5");

  public static void main(String[] args){
    //mache einen unendlichen Strom der Zahlen x_1 x_2 x_3 x_4 x_5 ...
    // mit x_1 = 1 und x_{n+1} = x_n + 1
    Stream.iterate(BigInteger.ONE, n -> n.add(BigInteger.ONE))
      .parallel()
    // filtere aus diesem alle Zahlen n heraus, für die gilt
    // gcd((n+1)^5+5,n^5+5) > 1
      .filter(n -> n.add(BigInteger.ONE).pow(5).add(FIVE)
                    .gcd(n.pow(5).add(FIVE))
                    .compareTo(BigInteger.ONE) > 0)
      .limit(10)
    //drucke jeden Wert des gefilterten Stroms auf der Konsole in eigener
    ↪  Zeile
      .forEach(n -> System.out.println(n));
  }
}
```

2 Hierarchische Strukturen

Von zentraler Bedeutung in der Softwareentwicklung ist es, Daten auf eine hierarchische Art und Weise zu strukturieren. Solche hierarchisch strukturierten Daten sind als Bäume bekannt. Sie sind in der Softwareentwicklung von so zentraler Bedeutung, dass sich verschiedene Standards durchgesetzt haben, solche Daten in textueller Form zu serialisieren. Zum einen in Form von XML-Dokumenten, zum anderen aber auch z.B. durch das JASON-Format.

In diesem Kapitel werden wir uns mit hierarchischen Daten zunächst in allgemeiner Form und anschließend in Form von XML-Dokumenten beschäftigen.

2.1 Einfach verkettete Listen

Eine der häufigsten Datenstrukturen in der Programmierung sind Sammlungstypen. In fast jedem nichttrivialen Programm wird es Punkte geben, an denen eine Sammlung mehrerer Daten gleichen Typs anzulegen sind. Eine der einfachsten Strukturen, um Sammlungen anzulegen, sind Listen. Da Sammlungstypen oft gebraucht werden, stellt Java entsprechende Klassen als Standardklassen zur Verfügung. Bevor wir uns aber diesen bereits vorhandenen Klassen zuwenden, wollen wir in diesem Kapitel Listen selbst spezifizieren und programmieren.

2.1.1 Formale Definition

Wir werden Listen als abstrakten Datentyp formal spezifizieren. Ein abstrakter Datentyp (ADT) wird spezifiziert über eine endliche Menge von Methoden. Hierzu wird spezifiziert, auf welche Weise Daten eines ADT konstruiert werden können. Dazu werden entsprechende Konstruktormethoden spezifiziert. Dann wird eine Menge von Funktionen definiert, die wieder Teile aus den konstruierten Daten selektieren können. Schließlich werden noch Testmethoden spezifiziert, die angeben, mit welchem Konstruktor ein Datenobjekt erzeugt wurde. Der Zusammenhang zwischen Konstruktoren und Selektoren sowie zwischen den Konstruktoren und den Testmethoden wird in Form von Gleichungen spezifiziert.

Der Trick, der angewendet wird, um abstrakte Datentypen wie Listen zu spezifizieren, ist die Rekursion. Das Hinzufügen eines weiteren Elements zu einer Liste wird dabei als das Konstruieren einer neuen Liste aus der Ursprungsliste und einem weiteren Element

betrachtet. Mit dieser Betrachtungsweise haben Listen eine rekursive Struktur: eine Liste besteht aus dem zuletzt vorne angehängten neuen Element, dem sogenannten Kopf der Liste, und aus der alten Teilliste, an die dieses Element angehängt wurde, dem sogenannten Schwanz der Liste. Wie bei jeder rekursiven Struktur bedarf es eines Anfangs der Definition. Im Falle von Listen wird dieses durch die Konstruktion einer leeren Liste spezifiziert.[1]

Listenkonstruktoren

Abstrakte Datentypen wie Listen lassen sich durch ihre Konstruktoren spezifizieren. Die Konstruktoren geben an, wie Daten des entsprechenden Typs konstruiert werden können. In dem Fall von Listen bedarf es nach den obigen Überlegungen zweier Konstruktoren:

- einem Konstruktor für neue Listen, die noch leer sind.
- einem Konstruktor, der aus einem Element und einer bereits bestehenden Liste eine neue Liste konstruiert, indem an die Ursprungsliste das Element angehängt wird.

Wir benutzen in der Spezifikation eine mathematische Notation der Typen von Konstruktoren.[2] Dem Namen des Konstruktors folgt dabei mit einem Doppelpunkt abgetrennt der Typ. Der Ergebnistyp wird von den Parametertypen mit einem Pfeil getrennt.

Somit lassen sich die Typen der zwei Konstruktoren für Listen wie folgt spezifizieren:

- Empty: () → **List**
- Cons: (**Object**,**List**) →**List**

Listenselektoren

Die Selektoren können wieder auf die einzelnen Bestandteile der Konstruktion zurückgreifen. Der Konstruktor Cons hat zwei Parameter. Für Cons-Listen werden zwei Selektoren spezifiziert, die jeweils einen dieser beiden Parameter wieder aus der Liste selektieren. Die Namen dieser beiden Selektoren sind traditioneller Weise *head* und *tail*.

- head: **(List)** →**Object**
- tail: **(List)** →**List**

Der funktionale Zusammenhang von Selektoren und Konstruktoren läßt sich durch folgende Gleichungen spezifizieren:

$$head(Cons(x, xs)) = x$$

[1] Man vergleiche es mit der Definition der natürlichen Zahlen: die 0 entspricht der leeren Liste, der Schritt von n nach n+1 dem Hinzufügen eines neuen Elements zu einer Liste.

[2] Entgegen der Notation in Java, in der der Rückgabetyp kurioser Weise vor den Namen der Methode geschrieben wird.

$$tail(Cons(x, xs)) = xs$$

Testmethoden für Listen

Um für Listen Algorithmen umzusetzen, ist es notwendig, unterscheiden zu können, welche Art der beiden Listen vorliegt: die leere Liste oder eine Cons-Liste. Hierzu bedarf es noch einer Testmethode, die mit einem bool'schen Wert als Ergebnis angibt, ob es sich bei der Eingabeliste um die leere Liste handelte oder nicht. Wir wollen diese Testmethode *isEmpty* nennen. Sie hat folgenden Typ:

- isEmpty: **List**→ **boolean**

Das funktionale Verhalten der Testmethode läßt sich durch folgende zwei Gleichungen spezifizieren:

$$isEmpty(Empty()) = true$$
$$isEmpty(Cons(x, xs)) = false$$

Somit ist alles spezifiziert, was eine Listenstruktur ausmacht. Listen können konstruiert werden, die Bestandteile einer Liste wieder einzeln selektiert und Listen können nach der Art ihrer Konstruktion unterschieden werden.

2.1.2 Implementierung

LiLi.java

```
package name.panitz.util;
public class LiLi<A> implements Iterable<A>{
  private final A hd;
  private final LiLi<A> tl;

  public LiLi(A hd, LiLi<A> tl){
    this.hd = hd;
    this.tl = tl;
  }

  public LiLi(){
    this.hd = null;
    this.tl = null;
  }

  public boolean isEmpty(){
    return tl==null && hd==null;
  }

  public A getHead(){
    return hd;
  }
  public LiLi<A> getTail(){
    return tl;
  }
```

Ein Iterator für Listen

Für die so definierten Listen können wir jetzt sofort das gelernte aus dem letzten Kapitel anwenden, indem wir einen Iterator für Listen schreiben. Listen sind natürlich die Paradeanwendung für iterierbare Objekte.

LiLi.java

```
public java.util.Iterator<A> iterator(){
  return new It(this);
}

private class It implements java.util.Iterator<A>{
  LiLi<A> theList;
  It(LiLi<A> li){theList = li;}
  public boolean hasNext(){
    return !theList.isEmpty();
  }
  public A next(){
    A result = theList.getHead();
    theList = theList.getTail();
    return result;
  }
}
```

2.1.3 Mathematische Definition

Betrachten wir jetzt wieder die Listen von der mathematischen Seite. Die obige Javaklasse definiert implizit eine induktive Struktur, indem gesagt wird, dass man eine leere Liste konstruieren kann und aus einer Liste und einem weiteren Element eine neue Liste konstruieren kann. Etwas formaler und nicht programmiersprachlich ließen sich Listen wie folgt definieren:

Sei eine Menge E der Elemente gegeben. Die Menge der Listen über E im Zeichen $List_E$ sei dann die kleinste Menge, für die gilt:

- Die leere Liste ist eine Liste: $\text{Nil}() \in \text{List}_E$.
- Sei $e \in E$ und sei $es \in List_E$.
  Dann ist auch $\text{Cons}(e, es) \in \text{List}_E$.

Induktive Strukturen

Daten sollen nach Möglichkeit dynamisch wachsen können, d.h. potentiell beliebig groß werden können. Um einen Datentyp zu definieren, dessen Elemente potentiell beliebig groß werden können, bedient man sich einer strukturell induktiven Definition. Mit den einfach verketteten Listen haben wir eine induktive Struktur definiert. Dieses Verfahren, Mengen mit zu definieren, ist aus der Mathematik bekannt. Dort ist die Menge der natürlichen Zahlen ein prominentes Beispiel einer induktiv definierten Menge. Die Zahl 0 ist Element der Menge der natürlichen Zahlen. Bei unseren Listen entspricht dieses der leeren Liste. Zusätzlich gilt, wenn eine Zahl n in der Menge der natürlichen Zahlen ist,

dass ist auch eine Zahl n+1 in der Menge der natürlichen Zahlen enthalten. Entsprechend können wir für jede Liste *xs* eine Liste, die ein Element mehr enthält, erzeugen.

Um in der Mathematik eine Eigenschaft für alle Elemente der Menge der natürlichen Zahlen zu beweisen, gibt es ein spezielles Beweisverfahren, dass sich an der strukturellen Definition orientiert: die vollständige Induktion. Diese besteht aus drei Schritte:

- **Induktions-Anfang**: zeige, dass die Eigenschaft für die Zahl 0 gilt.
- **Induktions-Vorraussetzung**: nimm an, dass die Eigenschaft bereits für alle Zahlen kleiner gleich einer bestimmten Zahl n korrekt bewiesen ist.
- **Induktions-Schluss**: zeige, dass die Annahme dann auch für die Zahl n+1 gilt.

Als Beispiel können wir die Gaußsche Summenformel beweisen, die behauptet:

$$\sum_{i=0}^{n} i = \frac{n * (n+1)}{2}$$

- **Induktions-Anfang**:

  $$0 = (0 * 1)/2$$

  ist korrekt.

- **Induktions-Vorraussetzung**:

  $$\sum_{i=0}^{n} i = \frac{n * (n+1)}{2}$$

  ist beweisen für alle Zahlen kleiner oder gleich eines bestimmten n.

- **Induktions-Schluss**:

$$\begin{aligned}
\sum_{i=0}^{n+1} i &= (n+1) + \sum_{i=0}^{n} i \\
&= (n+1) + \frac{n * (n+1)}{2} \\
&= \frac{2 * (n+1)}{2} + \frac{n * (n+1)}{2} \\
&= \frac{2 * (n+1) + n * (n+1)}{2} \\
&= \frac{(n+2) * (n+1)}{2} \\
&= \frac{(n+1) * (n+2)}{2}
\end{aligned}$$

Rekursive Algorithmen

Ebenso wie der Mathematiker bei einem Beweis per vollständiger Induktion vorgeht, können wir Algorithmen für die induktiv definierte Struktur unserer Listen definieren. Auch hier gibt es die drei Schritte:

- Induktionsanfang: das Ergebnis der zu entwickelnden Funktion wird für die leere Liste definiert.
- Induktionsvorraussetzung. wir nehmen an, dass die Funktion bereits für kürzere Listen funktioniert. Insbesondere für die um ein Element kürzere Liste, nämlich für die Liste, die wir mit `getTail()` erhalten.
- Induktionsschritt: aus dem Ergebnis für die Induktionsvorraussetzung wird das Ergebis für den Gesamtliste definiert.

Für viele einfache Algorithmen ist dieser Weg recht einfach.

Längenberechnung für Listen Ein einfacher Algorithmus für unsere induktive Listendefinition stellt die Berechnung der Länge der Liste dar, mit Hilfe der wir die einzelnen Elemente zählen.

LiLi.java

```
public int length(){
```

Zunächst als der Induktionsanfang. Das Ergebnis für die leere List. Dieses ist einfach. Eine leere Liste hat die Länge 0.

LiLi.java

```
if (isEmpty()) return 0;
```

Anschließend können wir über die Induktionsvoraussetzung annehmen, dass die Funktion für die um ein Element kleinere Liste bereits definiert ist und das korrekte Ergebnis berechnet:

LiLi.java

```
final int tailResult = getTail().length();
```

Schließlich ist zu überlegen, wie von dem Ergebnis der Voraussetzung das Gesamtergebnis zu definieren ist. Im Falle der Länge ist dieses besonders einfach, es muss 1 hinzugezählt werden:

LiLi.java

```
final int result = 1 + tailResult;
```

Damit lässt sich das Ergebnis zurück geben:

LiLi.java

```
  return result;
}
```

Test auf ein Element Nach dem gleichen Schema lässt sich auch die Funktion schreiben, die testet, ob in der Liste ein bestimmtes Element enthalten ist:

LiLi.java

```
public boolean contains(A a){
```

Zunächst als der Induktionsanfang. Das Ergebnis für die leere List.

LiLi.java

```
if (isEmpty()) return false;
```

Anschließend können wir über die wieder Induktionsvoraussetzung annehmen, dass die Funktion für die um ein Element kleinere Liste bereits definiert ist und das korrekte Ergebnis berechnet:

LiLi.java

```
final boolean tailResult = getTail().contains(a);
```

Schließlich ist zu überlegen, wie von dem Ergebnis der Voraussetzung das Gesamtergebnis zu definieren ist. In diesen Falle betrachten wir erst das erste Element und benutzen dann das Ergebnis der Induktionsvoraussetzung.

LiLi.java

```
final boolean result =
  getHead().equals(a) || tailResult;
```

Damit haben wir das Ergebnis:

LiLi.java

```
  return result;
}
```

Inhaltlich liegt eine komplett andere Aufgabenstellung als bei der Längenberechnung vor. Strukturell benutzen wir exakt dasselbe Vorgehen, das der vollständigen Induktion

der Mathematik entspricht.

Aneinanderhängen von Listen Nach dem gleichen Schema lässt sich auch die Funktion schreiben, die eine neue Liste erzeugt durch das Anhängen einer zweiten Liste.:

LiLi.java

```
public LiLi<A> append(LiLi<A> that){
```

Zunächst als der Induktionsanfang. Das Ergebnis für die leere List.

LiLi.java

```
if (isEmpty()) return that;
```

Anschließend können wir über die wieder Induktionsvoraussetzung annehmen, dass die Funktion für die um ein Element kleinere Liste bereits definiert ist und das korrekte Ergebnis berechnet:

LiLi.java

```
final LiLi<A> tailResult = getTail().append(that);
```

Schließlich ist zu überlegen, wie von dem Ergebnis der Voraussetzung das Gesamtergebnis zu definieren ist. In diesem Fall ist noch das erste Element an die Ergebnisliste vorne anzuhängen.:

LiLi.java

```
final LiLi<A> result  = new LiLi<A>(getHead(),tailResult);
```

Damit haben wir das Ergebnis:

LiLi.java

```
  return result;
}
```

Inhaltlich liegt wieder eine komplett andere Aufgabenstellung als bei der Längenberechnung oder dem Test auf ein Element vor. Strukturell benutzen wir aber wieder exakt dasselbe Vorgehen, das der vollständigen Induktion der Mathematik entspricht.

LiLi.java

```
}
```

2.2 Bäume als verallgemeinerte Listen

In diesem Abschnitt soll jetzt das Prinzip der Listen verallgemeinert werden zu Bäumen, Listen sind nämlich nur ein Speziellfall einer Baumstruktur.

Bäume sind ein gängiges Konzept um hierarchische Strukturen zu modellieren. Sie sind bekannt aus jeder Art von Baumdiagramme, wie Stammbäumen oder Firmenhierarchien. In der Informatik sind Bäume allgegenwärtig. Fast alle komplexen Daten stellen auf die eine oder andere Art einen Baum dar. Beispiele für Bäume sind mannigfach:

- **Dateisystem**: Ein gängiges Dateisystem, wie es aus Unix, MacOS und Windows bekannt ist, stellt eine Baumstruktur dar. Es gibt einen ausgezeichneten Wurzelordner, von dem aus zu jeder Datei einen Pfad existiert.
- **Klassenhierarchie**: Die Klassen in Java stellen mit ihrer Ableitungsrelation eine Baumstruktur dar. Die Klasse `Object` ist die Wurzel dieses Baumes, von der aus alle anderen Klassen über einen Pfad entlang der Ableitungsrelation erreicht werden können.
- **XML**: Die logische Struktur eines XML-Dokuments ist ein Baum. Die Kinder eines Elements sind jeweils die Elemente, die durch das Element eigeschlossen sind.
- **Parserergebnisse**: Ein Parser, der gemäß einer Grammatik prüft, ob ein bestimmter Satz zu einer Sprache gehört, erzeugt im Erfolgsfall eine Baumstruktur. Im nächsten Kapitel werden wir dieses im Detail kennenlernen.
- **Listen**: Auch Listen sind Bäume, allerdings eine besondere Art, in denen jeder Knoten nur maximal ein Kind hat.
- **Berechnungsbäume**: Zur statischen Analyse von Programmen, stellt man Bäume auf, in denen die Alternativen eines bedingten Ausdrucks Verzweigungen im Baum darstellen.
- **Tableaukalkül**: Der Tableaukalkül ist ein Verfahren zum Beweis logischer Formeln. Die dabei verwendeten Tableaux sind Bäume.
- **Spielbäume**: Alle möglichen Spielverläufe eines Spiels können als Baum dargestellt werden. Die Kanten entsprechen dabei einem Spielzug.
- **Prozesse**: Auch die Prozesse eines Betriebssystems stellen eine Baumstruktur dar. Die Kinder eines Prozesses sind genau die Prozesse, die von ihm gestartet wurden.

Wie man sieht, lohnt es sich, sich intensiv mit Bäumen vertraut zu machen, und man kann davon ausgehen, was immer in der Zukunft neues in der Informatik entwickelt werden wird, Bäume werden darin in irgendeiner Weise eine Rolle spielen.

Ein Baum besteht aus einer Menge von Knoten die durch gerichtete Kanten verbunden sind. Die Kanten sind eine Relation auf den Knoten des Baumes. Die Kanten verbinden jeweils einen Elternknoten mit einem Kinderknoten. Ein Baum hat einen eindeutigen Wurzelknoten. Dieses ist der einzige Knoten, der keinen Elternknoten hat, d.h. es

gibt keine Kante, die zu diesen Knoten führt. Knoten, die keinen Kinderknoten haben, d.h. von denen keine Kante ausgeht, heißen Blätter.

Die Kinder eines Knotens sind geordnet, d.h. sie stehen in einer definierten Reihenfolge. Eine Folge von Kanten, in der der Endknoten einer Vorgängerkante der Ausgangsknoten der nächsten Kanten ist, heißt Pfad.

In einem Baum darf es keine Zyklus geben, das heißt, es darf keinen Pfad geben, auf dem ein Knoten zweimal liegt.

Knoten können in Bäumen markiert sein, z.B. einen Namen haben. Mitunter können auch Kanten eine Markierung tragen.

Bäume, in denen Jeder Knoten maximal zwei Kinderknoten hat, nennt man Binärbäume. Bäume, in denen jeder Knoten maximal einen Kinderknoten hat, heißen Listen.

2.2.1 Formale Definition

Auch Bäume werden wir induktiv definieren:

Sei eine Menge E der Elemente gegeben. Die Menge der Bäume über E im Zeichen $Tree_E$ sei dann die kleinste Menge, für die gilt:

- Der leere Baum ist ein Baum: $Empty \in Tree_E$.
- Sei $e \in E$ und seien $t_1, \ldots, t_n \in Tree_E$.
  Dann ist auch $Branch(e, t_1, \ldots, t_n) \in Tree_E$.

Der einzige Unterschied zu unserer formalem Definition der Listen ist, dass im zweiten Punkt ein Baum aus n bereits existierenden Bäumen und einen neuem Element gebaut werden und nicht wie bei Listen aus einer existierenden Liste und einem Element. Wenn in der Definition der Bäume das n immer nur 1 ist, dann ist die Definition identisch mit der Definition von Listen.

2.2.2 Implementierung einer Baumklasse

Die Definition eines Baumes war sehr ähnlich der Definition der Listen. Entsprechend ist auch die Implementierung sehr ähnlich zu unserer Listenimplementierung. Wir schreiben eine Klasse, die generisch über die Elemente, die an den Baumknoten stehen, ist:

Tree.java

```
package name.panitz.util;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;


public class Tree<E> implements Iterable<E>{
```

Ebenso wie bei Listen, benötigen wir ein Feld, für das an dem Baumknoten stehende Element. Dieses wurde bei Listen meist als `head` bezeichnet.

Tree.java

```
public E element = null;
```

Eine Listenzelle hatte einen Verweis auf die Restliste. Bei Bäumen gibt es nicht nur einen Unterbaum, sondern mehrere Unterbäume, die wiederum in einer Liste gespeichert werden. Hierzu nutzen wir die Standardliste. Also anstatt eines Feldes `LiLi<A> tail` wie bei Listen, benötigen wir für Bäume ein Feld, das eine Liste von Teilbäumen enthält.

Tree.java

```
public List<Tree<E>> childNodes = new ArrayList<>();
```

In einem boolschen Flag sei vermerkt, ob es sich um einen leeren Baum, der noch gar keinen Knoten enthält handelt oder nicht.

Tree.java

```
public boolean isEmptyTree;
```

Wir verketten unsere Bäume auch rückwärts, von den Kindern zu ihren Elternknoten.

Tree.java

```
public Tree<E> parent = null;
```

Laut Spezifikation gibt es zwei Konstruktoren. Den ersten zum Erzeugen eines leeren Baumes:

Tree.java

```
public Tree(){
  isEmptyTree = true;
}
```

Der zweite erzeugt eine neue Baumwurzel mit einer bestimmten Markierung.

Tree.java

```
public Tree(E e,Tree<E>... ts){
  element = e;
  isEmptyTree = false;
  for (Tree<E> t:ts){
    if (!t.isEmptyTree) {
      childNodes.add(t);
      t.parent = this;
    }
  }
}
```

Hier wird das Konstrukt einer variablen Parameteranzahl verwendet. Die drei Punkte an dem letzten Parameter zeigen an, dass hier 0 bis n Parameter eines Typs erwartet werden.

Intern wird für diese Parameter ein Array erzeugt, über den wir in diesem Fall mit einer for-each Schleife iterieren.

Für weitere kleine Tests seien ein paar Beispielbäume bereit gestellt.

Tree.java

```
static Tree<String> a1 = new Tree<>("a1");
static Tree<String> a2 = new Tree<>("a2");
static Tree<String> a3 = new Tree<>("a3");
static Tree<String> b1 = new Tree<>("b1", a1, a2, a3);
static Tree<String> a4 = new Tree<>("a4");
static Tree<String> a5 = new Tree<>("a5");
static Tree<String> b2 = new Tree<>("b2", a4, a5);
static Tree<String> a6 = new Tree<>("a6");
static Tree<String> a7 = new Tree<>("a7");
static Tree<String> a8 = new Tree<>("a8");
static Tree<String> b3 = new Tree<>("b3", a6, a7, a8);
static Tree<String> c1 = new Tree<>("c1", b1, b2, b3);
```

Als ein weniger abstraktes Beispiel seien die Nachfahren des englische Königs George VI als ein Stammbaum angegeben:

Tree.java

```
static Tree<String> windsor =
  new Tree<>
    ("George"
    ,new Tree<>
      ("Elizabeth"
      ,new Tree<>
        ("Charles"
        ,new Tree<>("William",new Tree<>("George"))
        ,new Tree<>("Henry")
        )
      ,new Tree<>
        ("Andrew"
        ,new Tree<>("Beatrice")
        ,new Tree<>("Eugenie")
        )
      ,new Tree<>
        ("Edward"
        ,new Tree<>("Louise")
        ,new Tree<>("James")
        )
      ,new Tree<>
        ("Anne"
        ,new Tree<>("Peter")
        ,new Tree<>("Zara")
        )
      )
    ,new Tree<>
      ("Magaret"
      ,new Tree<>
        ("David"
        ,new Tree<>("Carles")
        ,new Tree<>("Maragrita")
        )

      ,new Tree<>("Sarah")
      )
    );
```

Stringdarstellung

Eine der ersten Methoden, die üblicher Weise für Klassen geschrieben wird, ist die Darstellung des Objektes als ein Text in Form eines Stringobjektes. Hierzu wird gängiger Weise die Methode `toString` aus der Klasse Objekt überschrieben. Für die Baumklassen wäre folgende Umsetzung denkbar, die wir allerdings auf Deutsch als `alsString` bezeichnet haben.

Tree.java

```
public String alsString(){
  if (isEmptyTree) return "Empty()";
  String result = "Branch("+element+")[";
  for (Tree<E> child:childNodes){
    result = result+child.alsString();
  }
  return result+"]";
}
```

Diese Umsetzung ist rekursiv, indem die Methode für jeden Kindknoten rekursiv aufgerufen wird. Die Teilergebnisse für die Kindbäume werden dabei über die Stringkonkatenation mit dem eingebauten Operator + aneinander gehängt.

Dieses ist nicht ganz unproblematisch. Stringobjekte sind in Java unveränderbar (unmutable). Ein einmal erzeugtes Stringobjekt wird nie verändert. Der plus-Operator erzeugt aus zwei Stringobjekten ein neues Stringobjekt. Dieses geschieht dadurch, dass ein neues Objekt erzeugt wird und alle Zeichen der beiden Ausgangsobjekte in dieses neue Objekt kopiert werden. Wenn für die Berechnung eines Stringergebnisses viele Teilsstring mit dem plus-Operator aneinander gehängt werden, dann werden sehr viele Objekte erzeugt und insbesondere sehr viele Zeichen immer wieder im Speicher von einem Speicherbereich auf einen anderen kopiert.. Dieses werden wir im zweiten Teil der Vorlesung noch direkter sehen, wenn in der Programmiersprache C, die Operationen des plus-Operators explizit selbst programmiert werden müssen.

Es empfiehlt sich also, bei der Konstruktion großer Texte, nicht gleich alle Teilstrings mit dem plus-Operator aneinander zu hängen und so permament Teilstrings zu kopieren.

Strings puffern mit `StringBuffer` Das Java Standard-API stellt eine Klasse zur Verfügung, die es ermöglicht eine Folge von Strings zu sammeln, die dann im Endeffekt zu einem großen String konkateniert werden: die Klasse `StringBuffer`. Diese hat intern eine Liste der einzelnen aneinander zu hängenden Strings. Erst wenn alle Strings eingesammelt wurden, dann werden alle Strings zu einem großen String zusammen kopiert. Somit werden die Zeichen der einzelnen Strings nicht mehrfach im Speicher kopiert und das Anlegen temporärer Stringobjekte vermieden.

Die Methode `toString()` kann mit Hilfe der Klasse `StringBuffer` effizienter umgesetzt werden. Es wird ein `StringBuffer`-Objekt angelegt. Mit der Methode `append` werden die einzelnen Strings dann diesem Objekt angefügt. Ein terminaler Aufruf der Methode `toString` auf dem `StringBuffer`-Objekt führt dann dazu, dass der Ergebnisstring erzeugt wird:

Zunächst nehmen wir aber den einfachen Fall des leeren Baumes aus:

Tree.java

```
@Override public String toString(){
  if (isEmptyTree) return "Empty()";
```

Andernfalls wird ein `StringBuffer`-Objekt mit dem Anfang des Ergebnisstrings erzeugt:

Tree.java

```
StringBuffer result = new StringBuffer("Branch(");
```

Diesem wird zunächst das Element in Stringdarstellung zugefügt:

Tree.java

```
result.append(element.toString());
result.append(")[");
```

Schließlich werden noch die Kinder durchiteriert, um deren String-Darstellung dem `StringBuffer`-Objekt zuzufügen:

Tree.java

```
for (Tree<E> child:childNodes){
  result.append(child.toString());
}
result.append("]");
```

Nun sind alle Teilstrings aufgesammelt und das Ergebnisobjekt kann erzeugt werden:

Tree.java

```
  return result.toString();
}
```

String schreiben mit einem `Writer` Wer sich die letzte Lösung der Methode `toString()` etwas genauer angeschaut hat, der wird bemerkt haben, dass trotzdem viele temporäre Teilstrings erzeugt werden. Bei jedem rekursiven Aufruf, wird wieder ein neues `StringBuffer`-Objekt erzeugt, welches bei der Rückgabe einen neuen String erzeugt. Es werden also auch in dieser Umsetzung immer wieder dieselben Teilstrings im Speicher kopiert. Dieses Problem lässt sich nur umgehen, wenn man das `StringBuffer`-Objekt als Parameter an die Methode übergibt.

Eine andere Möglichkeit, die noch flexibler ist, ermöglicht es Strings nicht explizit zu erzeugen, sondern in eine Datensenke zu schreiben. Hierzu dient die aus dem letzten

Semester bekannte Klasse `java.io.Writer`. Von dieser gibt es unterschiedliche Implementierungen, die jeweils die Zeichen in unterschiedliche Datensenken schreiben. Häufig wird diese natürlich eine Datei sein, es kann aber auch über ein Netzwerk an einen Server geschrieben werden, oder aber auch wieder nur einfach in einen String.

Um in ein `Writer`-Objekt zu schreiben, muss dieses wie am Ende des letzten Abschnitts erörtert, als Parameter übergeben werde. Zusätzlich ist zu beachten, dass die IO-Operationen zu Ausnahmen führen können:

Tree.java

```
public void writeAsString(java.io.Writer out) throws java.io.IOException{
```

Im Falle des leeren Baumes, wird dessen Darstellung geschrieben und die Methode verlassen:

Tree.java

```
if (isEmptyTree) {
  out.write("Empty()");
  return;
}
```

Wenn es sich um einen nichtleeren Baum handelt, können jetzt die einzelnen Teile nacheinander in den `Writer` geschrieben werden. Ähnlich wie in der vorherigen Lösung die einzelnen Teile dem `StringBuffer` angehängt wurden.

Tree.java

```
out.write("Branch(");
out.write(element.toString());
out.write(")[");
```

Der Unterschied zu der vorherigen Lösung ist, dass jetzt die rekursiven Aufrufe den `Writer` als Parameter übergeben.

Tree.java

```
  for (Tree<E> child:childNodes){
    child.writeAsString(out);
  }
  out.write("]");
}
```

Um diese Implementierung jetzt auf gleiche Weise nutzen zu können wie die Standardmethode `toString`, kann ein `StringWriter`-Objekt erzeugt werden, in den der String geschrieben wird.

Tree.java

```
public String asString(){
  try{
    java.io.StringWriter out = new java.io.StringWriter();
    writeAsString(out);
    return out.toString();
  }catch (java.io.IOException e){
   return "";
  }
}
```

Mit dieser Implementierung ist unnötiges Kopieren von Teilstrings komplett vermieden worden. zusätzlich ist sie flexibler, weil die Methode `writeAsString` nicht nur zum Erzeugen eines Strings genutzt werden kann, sondern auch dazu, die Stringdarstellung in unterschiedliche Datensenken zu schreiben.

Elemente in einer Liste gesammelt

Ist man nicht mehr an der Baumstruktur interessiert, so kann man sich die Elemente eines Baums in einer Liste speichern.

Tree.java

```
public java.util.List<E> asList(){
  return asList(new java.util.LinkedList<>());
}
public List<E> asList(List<E> result){
  if (!isEmptyTree) result.add(element);

  for (Tree<E> child:childNodes) child.asList(result);

  return result;
}
```

Ein einfacher Iterator über die Listendarstellung

Mit der Listensammlung aller Elemente eines Baumes lässt sich schließlich leicht ein Baum über alle seine Elemente iterierbar machen. Es muss nur die Liste erzeugt werden und über diese iteriert werden.

Tree.java

```
@Override public java.util.Iterator<E> iterator(){
  return asList().iterator();
}
```

Rekursive Methoden mit reduce auf Kindknoten

Mit den Möglichkeiten von Java 8 lassen sich fast alle Algorithmen, die in den Baum absteigen, um dort Informationen zu sammeln, auch ohne explizite Schleifen sondern durch Methode höherer Ordnung auf die Kinderliste ausdrücken. Ob das immer besser ist, sei allerdings dahingestellt.

Betrachten wir zunächst die Größe eines Baumes. Diese lässt sich als ein einziger Ausdruck formulieren. Zunächst wird durch den Bedingungsoperator der Spezialfall eines leeren Baumes behandelt. Ansonsten ist es 1 zur Summe der Größen aller Kinderbäume:

Tree.java

```
public int size(){
  return  isEmptyTree ? 0 :
   1+childNodes
    .parallelStream()
    .reduce(0,(result,child) -> result+child.size(), Math::addExact);
}
```

Die Summe der Kinderbäume ist mit der Methode `reduce` auf Stream gelöst. Diese entspricht der Methode `reduce`, wie wir sie selbst in dem eigenen Interface `Iterable` umgesetzt haben. Sie bekommt allerdings zwei Funktionsobjekte. Die erste Funktion beschreibt, wie ein einzelnen Element sein Ergebnis zum Gesamtergebnis hinzuzählt. In unserem Fall, wird von jedem Kind die Größe zum Gesamtergebnis addiert. Die zweite Funktion beschreibt, wie Teilergebnisse zu verknüpfen sind. In unserem Fall durch die Addition. Diese zwei Funktionen werden gebraucht, wenn der Stream, auf dem gearbeitet wird, parallelisiert wird. Dann wird z.B. vielleicht parallel das Teilergebnis der ersten Hälfte der Kinder und der zweiten Hälfte der Kinder berechnet. Diese Teilergebnisse sind zu addieren.

Da wir unsere Bäume unser Interface `Iterable` haben implementieren lassen, können wir aber sehr einfach die dort als default definierte Methode `reduce` benutzen. Hier erhalten wir allerdings keine Parallelisierung und erzeugen zusätzlich über die Methode `iterator()` der Klasse `Tree` erst eine Liste aller Elemente.

Tree.java

```
public int size2(){
  return reduce(0, (result,elem) -> result + 1);
}
```

Sehr sehr ähnlich sieht nun die Lösung für eine gänzlich andere Fragestellung aus. Was ist die maximale Tiefe des Baumes, also wie viele Generationen hat der Baum. Statt die Teilergebnisse aufzuaddieren, nehmen wir jetzt immer das Maximum der Teilergebnisse. Auch diese Aufgabe ist wunderbar parallelisierbar:

Tree.java

```
  public int generations(){
    return isEmptyTree ? 0 :
     1 +
      childNodes
      .parallelStream()
      ↪  .reduce(0,(result,child)->Math.max(result,child.generations()),Math::max);
  }
}
```

2.3 XML als serialisierte Baumstruktur

XML ist eine Sprache, die es erlaubt Dokumente mit einer logischen Struktur zu beschreiben. Die Grundidee dahinter ist, die logische Struktur eines Dokuments von seiner Visualisierung zu trennen. Ein Dokument mit einer bestimmten logischen Struktur kann für verschiedene Medien unterschiedlich visualisiert werden, z.B. als HTML-Dokument für die Darstellung in einem Webbrowser, als pdf- oder postscript-Datei für den Druck des Dokuments und das für unterschiedliche Druckformate. Eventuell sollen nicht alle Teile eines Dokuments visualisiert werden. XML ist zunächst eine Sprache, die logisch strukturierte Dokumente zu schreiben, erlaubt.

Dokumente bestehen hierbei aus den eigentlichen Dokumenttext und zusätzlich aus Markierungen dieses Textes. Die Markierungen sind in spitzen Klammern eingeschlossen.

Der eigentliche Text des Dokuments sei:

.

```
The Beatles White Album
```

Die einzelnen Bestandteile dieses Textes können markiert werden:

```
<cd>
  <artist>The Beatles</artist>
  <title>White Album</title>
</cd>
```

Die XML-Sprache wird durch ein Industriekonsortium definiert, dem W3C. Dieses ist ein Zusammenschluß vieler Firmen, die ein gemeinsames Interesse eines allgemeinen Standards für eine Markierungssprache haben. Die eigentlichen Standards des W3C heißen nicht Standard, sondern Empfehlung *(recommendation)*, weil es sich bei dem W3C nicht um eine staatliche oder überstaatliche Standartisiertungsbehörde handelt. Die aktuelle Empfehlung für XML liegt seit August 2006 als Empfehlung in der Version 1.1 vor .

XML enstand Ende der 90er Jahre und ist abgeleitet von einer umfangreicheren Dokumentenbeschreibungssprache: SGML. Der SGML-Standard ist wesentlich komplizierter und krankt daran, daß es extrem schwer ist, Software für die Verarbeitung von SGML-Dokumenten zu entwickeln. Daher fasste SGML nur Fuß in Bereichen, wo gut strukturierte, leicht wartbare Dokumente von fundamentaler Bedeutung waren, so dass die Investition in teure Werkzeuge zur Erzeugung und Pflege von SGML-Dokumenten sich rentierte. Dies waren z.B. Dokumentationen im Luftfahrtbereich.[3]

Die Idee bei der Entwicklung von XML war: eine Sprache mit den Vorteilen von SGML zu Entwickeln, die klein, übersichtlich und leicht zu handhaben ist.

Wie wir im Laufe des Kapitels sehen werden, ist die logische Struktur eines XML-Dokuments auch wieder eine hierarchische Baumstruktur, so dass wir alle unsere Überlegungen zu Bäumen direkt auf XML-Dokumente anwenden können.

2.4 Das syntaktische XML-Format

Die grundlegendste Empfehlung des W3C legt fest, wann ein Dokument ein gültiges XML-Dokument ist, die Syntax eines XML-Dokuments. Die nächsten Abschnitte stellen die wichtigsten Bestandteile eines XML-Dokuments vor.

Jedes Dokument beginnt mit einer Anfangszeile, in dem das Dokument angibt, daß es ein XML-Dokument nach einer bestimmten Version der XML Empfehlung ist:

```
<?xml version="1.1"?>
```

Dieses ist die erste Zeile eines XML-Dokuments. Vor dieser Zeile darf kein Leerzeichen stehen. Die derzeitig aktuellste und einzige Version der XML-Empfehlung ist die Version 1.1. vom 16. August 2006. Nach Aussage eines Mitglieds des W3C ist es sehr unwahrscheinlich, dass es jemals eine Version 2.0 von XML geben wird. Zuviele weitere

[3] Man sagt, ein Pilot brauche den Copiloten, damit dieser die Handbücher für das Flugzeug trägt.

Techniken und Empfehlungen basieren auf XML, so daß die Definition von dem, was ein XML-Dokument ist kaum mehr in größeren Rahmen zu ändern ist.

2.4.1 Elementknoten

Der Hauptbestandteil eines XML-Dokuments sind die Elemente. Dieses sind mit der Spitzenklammernotation um Teile des Dokuments gemachte Markierungen. Ein Element hat einen *Tagnamen*, der ein beliebiges Wort ohne Leerzeichen sein kann. Für einen Tagnamen *name* beginnt ein Element mit <name> und endet mit </name>. Zwischen dieser Start- und Endemarkierung eines Elements kann Text oder auch weitere Elemente stehen.

Es wird für XML-Dokument verlangt, daß es genau ein einziges oberstes Element hat. Somit ist ein einfaches XML-Dokument ein solches Dokument, in dem der gesammte Text mit einem einzigen Element markiert ist:

.

```
<?xml version="1.0"?>
<myText>Dieses ist der Text des Dokuments. Er ist
mit genau einem Element markiert.
</myText>
```

Im einführenden Beispiel haben wir schon ein XML-Dokument gesehen, das mehrere Elemente hat. Dort umschließt das Element `<cd>` zwei weitere Elemente, die Elemente `<artist>` und `<title>`. Die Teile, die ein Element umschließt, werden der Inhalt des Elements genannt.

Ein Element kann auch keinen, sprich den leeren Inhalt haben. Dann folgt der öffnenden Markierung direkt die schließende Markierung.

Folgendes Dokument enthält ein Element ohne Inhalt:

.

```
<?xml version="1.0"?>
<skript>
  <page>erste Seite</page>
  <page></page>
  <page>dritte Seite</page>
</skript>
```

2.4.2 Leere Elemente

Für ein Element mit Tagnamen *name*, das keinen Inhalt hat, gibt es die abkürzenden Schreibweise: `<name/>`

Das vorherige Dokument läßt sich somit auch wie folgt schreiben:

```
<?xml version="1.0"?>
<skript>
  <page>erste Seite</page>
  <page/>
  <page>dritte Seite</page>
</skript>
```

2.4.3 Gemischter Inhalt

Die bisherigen Beispiele haben nur Elemente gehabt, deren Inhalt entweder Elemente oder Text waren, aber nicht beides. Es ist aber auch möglich Elemente mit Text und Elementen als Inhalt zu schreiben. Man spricht dann vom gemischten Inhalt *(mixed content)*. Ein Dokument, in dem das oberste Element einen gemischten Inhalt hat:

```
<?xml version="1.0"?>
<myText>Der <landsmann>Italiener</landsmann>
<eigename>Ferdinand Carulli</eigename> war  als Gitarrist
ebenso wie der <landsmann>Spanier</landsmann>
<eigename>Fernando Sor</eigename> in <ort>Paris</ort>
ansäßig.</myText>
```

Tatsächlich ist gemischter Inhalt nicht bei jedermann beliebt. Es gibt zwei große Anwendungshintergründe für XML. Zum einen die Dokumentenbeschreibung. In diesem Kontext ist gemischter ganz natürlich, wie man am obigen Beispiel erkennen kann. Die zweite Fraktion von Anwendern, die XML nutzt, kommt eher aus dem Bereich allgemeiner Datenverarbeitung. Hier stehen eher Daten, wie in Datenbanken im Vordergrund und nicht Textdokumente. Es wird da zum Beispiel XML genutzt um an einen Webservice Daten zu übermitteln, die dort verarbeitet werden sollen. In diesem Kontext stört gemischter Inhalt mitunter oder hat zumindest wenig Sinn.

Die Problematik wird sofort ersichtlich, wenn man sich überlegt, dass Leerzeichen auch Text sind. Unter dieser Prämisse haben mit Sicherheit alle XML-Dokumente gemischten Inhalt, weil zur optischen besseren Lesbarkeit auch in einem Datendokument, das eigentlich keinen gemischten Inhalt hat, zwischen den einzelnen Elementknoten Leerzeichen in Form von Einrückungen und neuen Zeilen stehen.

2.4.4 XML Dokumente sind Bäume

Die wohl wichtigste Beschränkung für XML-Dokumente ist, dass sie eine hierarchische Struktur darstellen müssen. Zwei Elemente dürfen sich nicht überlappen. Ein Element darf erst wieder geschlossen werden, wenn alle nach ihm geöffneten Elemente wieder

geschlossen wurden. Das folgende ist kein gültiges XML-Dokument. Das Element `<bf>` wird geschlossen bevor das später geöffnete Element `<em>` geschlossen worde.

```
<?xml version="1.0"?>
<illegalDocument>
  <bf>fette Schrift <em>kursiv und fett</bf>
  nur noch kursiv</em>.
</illegalDocument>
```

Das Dokument wäre wie folgt als gültiges XML zu schreiben:

```
<?xml version="1.0"?>
<validDocument>
  <bf>fette Schrift <em>kursiv und fett</em></bf>
  <em>nur noch kursiv</em>.
</validDocument>
```

Dieses Dokument hat eine hierarchische Struktur.

Die hierarchische Struktur eines XML-Dokuments lässt sich in der Regel gut erkennen, wenn man das Dokument in einem Web-Browser öffnet. Fast alle Web-Browser stellen dann die logische Baumstruktur so dar, dass die Kindknoten eines Elementknoten aufgeklappt und zugeklappt werden können.

2.4.5 Chracter Entities

Sobald in einem XML-Dokument eine der spitzen Klammern `<` oder `>` auftaucht, wird dieses als Teil eines Elementtags interpretiert. Sollen diese Zeichen hingegen als Text und nicht als Teil der Markierung benutzt werden, sind also Bestandteil des Dokumenttextes, so muss man einen Fluchtmechanismus für diese Zeichen benutzen. Diese Fluchtmechanismen nennt man *character entities*. Eine Character Entity beginnt in XML mit dem Zeichen `&` und endet mit einem Semikolon `;`. Dazwischen steht der Name des Buchstabens. XML kennt die folgenden Character Entities:

EntityZeichenBeschreibung`&lt;` `<` (less than)`&gt;` `>` (greater than)`&amp;` `&` (ampersant)`&quot;` `"` (quotation

Manchmal gibt es große Textabschnitte in denen Zeichen vorkommen, die eigentlich durch character entities zu umschreiben wären, weil sie in XML eine reservierte Bedeu-

tung haben. XML bietet die Möglichkeit solche kompletten Abschnitte als eine sogenannte *CData Section* zu schreiben. Eine *CData section* beginnt mit der Zeichenfolge <![CDATA[und endet mit der Zeichenfolge:]]>. Dazwischen können beliebige Zeichenstehen, die eins zu eins als Text des Dokumentes interpretiert werden. Die im vorherigen Beispiel mit Character Entities beschriebenen Formeln lassen sich innerhalb einer CDATA-Section wie folgt schreiben.

```
<?xml version="1.0"?>
<formeln><![CDATA[
  x+1>x
  x*x<x*x*x für x > 1
]]></formeln>
```

Strukturell sind die CDATA-Sektionen auch nur Textknoten im Dokument. Sie unterscheiden sich von anderen Textknoten nur in der Darstellung in der serialisierten Textform.

2.4.7 Kommentare

XML stellt auch eine Möglichkeit zur Verfügung, bestimmte Texte als Kommentar einem Dokument zuzufügen. Diese Kommentare werden mit `<!--` begonnen und mit `-->` beendet. Kommentartexte sind nicht Bestandteil des eigentlichen Dokumenttextes. Im folgenden Dokument ist ein Kommentar eingefügt:

```
<?xml version="1.0"?>
<drehbuch>
<titel>Ben Hur</titel>
<akt>
  <szene>Ben Hur am Vorabend des Wagenrennens.
    <!--Diese Szene muß noch ausgearbeitet werden.-->
  </szene>
</akt>
</drehbuch>
```

Kommentarknoten werden nicht als eigentliche Knoten der Baumstruktur des Dokuments betrachtet.

2.4.8 Processing Instructions

In einem XML-Dokument können Anweisung stehen, die angeben, was mit einem Dokument von einem externen Programm zu tun ist. Solche Anweisungen können z.B. angeben, mit welchen Mitteln das Dokument visualisiert werden soll. Wir werden hierzu im nächsten Kapitel ein Beispiel sehen. Syntaktisch beginnt eine *processing instruction*

mit <? und endet mit ?>. Dazwischen stehen wie in der Attributschreibweise Werte für den Typ der Anweisung und eine Referenz auf eine externe Quelle. Ausschnitt aus dem XML-Dokument diesen Skripts, in dem auf ein Stylesheet verwiesen wird, daß das Skript in eine HTML-Darstellung umwandelt:

```
<?xml version="1.0"?>
<?xml-stylesheet
   type="text/xsl"
   href="../transformskript.xsl"?>

<skript>
<titelseite>
<titel>Grundlagen der Datenverarbeitung<white/>II</titel>
<semester>WS 02/03</semester>
</titelseite>
</skript>
```

Processing Instruction-knoten werden nicht als eigentliche Knoten der Baumstruktur des Dokuments betrachtet.

2.4.9 Attribute

Die Elemente eines XML-Dokuments können als zusätzliche Information auch noch Attribute haben. Attribute haben einen Namen und einen Wert. Syntaktisch ist ein Attribut dargestellt durch den Attributnamen gefolgt von einem Gleichheitszeichen gefolgt von dem in Anführungszeichen eingeschlossenen Attributwert. Attribute stehen im Starttag eines Elements.

Attribute werden nicht als Bestandteils des eigentlichen Textes eines Dokuments betrachtet. Dokument mit einem Attribut für ein Element.

```
<?xml version="1.0"?>
<text>Mehr Information zu XML findet man auf den Seiten
des <link address="www.w3c.org">W3C</link>.</text>
```

Mit Hilfe der Attribute ist es so möglich eine Abbildung an jeden Elementknoten zu definieren. Wie wir im letzten Semester gelernt haben, ist eine Abbildung eine Liste von Paaren. Die Paare sind dabei als Schlüssel/Wert-paare zu verstehen. Ein XML-Attribut stellt ein solches Schlüssel/Wert-Paar dar. Vorr dem Gleichheitszeichen steht der Schlüssel und eingeschlossen in Anführungsstrichen findet sich der Wert, auf dem der Schlüssel abgebildet wird.

2.4.10 Zeichencodierungen

Im Zusammenhang mit IO-Operationen wurde im letzten Semester bereits eingeführt, dass Textdokumente in unterschiedlichen Encodings physikalisch auf einem Datenträger gespeichert werden können.

Auch XML ist ein Dokumentenformat, das nicht auf eine Kultur mit einer bestimmten Schrift beschränkt ist, sondern in der Lage ist, alle im Unicode erfassten Zeichen darzustellen, seien es Zeichen der lateinischen, kyrillischen, arabischen, chinesischen oder sonst einer Schrift bis hin zur keltischen Keilschrift. Jedes Zeichen eines XML-Dokuments kann potentiell eines dieser mehreren zigtausend Zeichen einer der vielen Schriften sein. In der Regel benutzt ein XML-Dokument insbesondere im amerikanischen und europäischen Bereich nur wenige kaum 100 unterschiedliche Zeichen. Auch ein arabisches Dokument wird mit weniger als 100 verschiedenen Zeichen auskommen.

Wenn ein Dokument im Computer auf der Festplatte gespeichert wird, so werden auf der Festplatte keine Zeichen einer Schrift, sondern Zahlen abgespeichert. Diese Zahlen sind traditionell Zahlen die 8 Bit im Speicher belegen, ein sogenannter Byte (auch Oktett). Ein Byte ist in der Lage 256 unterschiedliche Zahlen darzustellen. Damit würde ein Byte ausreichen, alle Buchstaben eines normalen westlichen Dokuments in lateinischer Schrift (oder eines arabischen Dokuments darzustellen). Für ein Chinesisches Dokument reicht es nicht aus, die Zeichen durch ein Byte allein auszudrücken, denn es gibt mehr als 10000 verschiedene chinesische Zeichen. Es ist notwendig, zwei Byte im Speicher zu benutzen, um die vielen chinesischen Zeichen als Zahlen darzustellen.

Die *Codierung* eines Dokuments gibt nun an, wie die Zahlen, die der Computer auf der Festplatte gespeichert hat, als Zeichen interpretiert werden sollen. Eine Codierung für arabische Texte wird den Zahlen von 0 bis 255 bestimmte arabische Buchstaben zuordnen, eine Codierung für deutsche Dokumente wird den Zahlen 0 bis 255 lateinische Buchstaben inklusive deutscher Umlaute und dem ß zuordnen. Für ein chinesisches Dokument wird eine *Codierung* benötigt, die den 65536 mit 2 Byte darstellbaren Zahlen jeweils chinesische Zeichen zuordnet. Man sieht, dass es *Codierungen* geben muss, die für ein Zeichen ein Byte im Speicher belegen, und solche, die zwei Byte im Speicher belegen. Es gibt darüber hinaus auch eine Reihe Mischformen, manche Zeichen werden durch ein Byte andere durch 2 oder sogar durch 3 Byte dargestellt. Im Kopf eines XML-Dokuments kann angegeben werden, in welcher Codierung das Dokument abgespeichert ist.

Dieses Skript ist in einer Codierung gespeichert, die für westeuropäische Dokumente gut geeignet ist, da es für die verschiedenen Sonderzeichen der westeuropäischen Schriften einen Zahlenwert im 8-Bit-Bereich zugeordnet hat. Die Codierung mit dem Namen: `iso-8859-1`. Diese wird im Kopf des Dokuments angegeben:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<skript><kapitel>blablabla</kapitel></skript>
```

Wird keine Codierung im Kopf eines Dokuments angegeben, so wird als Standardcodierung die sogenannte `utf-8` Codierung benutzt. In ihr belegen lateinische Zeichen einen Byte und Zeichen anderer Schriften (oder auch das Euro Symbol) zwei bis drei Bytes. Eine Codierung, in der alle Zeichen mindestens mit zwei Bytes dargestellt werden ist: `utf-16`, die Standardabbildung von Zeichen, wie sie im *Unicode* definiert ist.

2.4.11 DTD

In der XML-Empfehlung integriert ist die Definition der document type description language, kurz DTD.

Die DTD ermöglicht es zu formulieren, welche Tags in einem Dokument vorkommen sollen. DTD ist keine eigens für XML erfundene Sprache, sondern aus SGML geerbt.

DTD-Dokumente sind keine XML-Dokumente, sondern haben eine eigene Syntax. Allerdings kann ein XML Dokument mit einer DTD beginnen. DTD-Abschnitte zu Beginn des Dokuments sind ein legaler und normaler Bestandteil eines XML Dokuments.

Wir stellen im einzelnen diese Syntax vor:

- <!DOCTYPE *root-element* [*doctype-declaration...*]>
  Legt den Namen des top-level Elements fest und enthält die gesammte Definition des erlaubten Inhalts.
- <!ELEMENT *element-name content-model*>
  assoziiert einen *content model* mit allen Elementen, die diesen Namen haben.

  content models können wie folgt gebildet werden.:
  - `EMPTY`: Das Element hat keinen Inhalt.
  - `ANY`: Das Element hat einen beliebigen Inhalt
  - `#PCDATA`: Zeichenkette, also der eigentliche Text.
  - durch einen regulären Ausdruck, der aus den folgenden Komponenten gebildet werden kann.
    - Auswahl von Alternativen (oder): `(...|...|...)`
    - Sequenz von Ausdrücken: `(...,...,...)`
    - Option: `...?`
    - Ein bis mehrfach Wiederholung: `...*`
    - Null bis mehrfache Wiederholung: `...+`
- <!ATTLIST *element-name attr-name attr-type attr-default ...*>
  Liste, die definiert, was für Attribute ein Element hat:

  Attribute werden definiert durch:
  - `CDATA`: beliebiger Text ist erlaubt

- (*value*|...): Aufzählung erlaubter Werte

Attribut *default* sind:

- `#REQUIRED`: Das Attribut muß immer vorhanden sein.
- `#IMPLIED`: Das Attribut ist optional
- `"value"`: Standardwert, wenn das Attribut fehlt.
- `#FIXED "value"`: Attribut kennt nur diesen Wert.

Ein Beispiel für eine DTD, die ein Format für eine Rezeptsammlung definiert.

```
<!DOCTYPE collection SYSTEM "collection.dtd" [
<!ELEMENT collection (description,recipe*)>

<!ELEMENT description ANY>

<!ELEMENT recipe (title,ingredient*,preparation
                 ,comment?,nutrition)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT ingredient (ingredient*,preparation)?>
<!ATTLIST ingredient name CDATA #REQUIRED
                     amount CDATA #IMPLIED
                     unit CDATA #IMPLIED>

<!ELEMENT preparation (step)*>

<!ELEMENT step (#PCDATA)>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT nutrition EMPTY>
<!ATTLIST nutrition fat CDATA #REQUIRED
                    calories CDATA #REQUIRED
                    alcohol CDATA #IMPLIED>]>
```

Es ist auch möglich DTDs als externe Dateien zu haben und zu Beginn eines XML-Dokuments aus diese zu referenzieren:

```
<!DOCTYPE note SYSTEM "collection.dtd">
```

2.5 XML Verarbeitung

2.5.1 Das DOM Api

Das DOM-API ist in der Standard-Javaentwicklungsumgebung implementiert. Hierzu findet sich im API das Paket `org.w3c.dom`. Es fällt auf, dass dieses Paket nur Schnittstellen und keine Klassen enthält. Der Grund hierfür ist, dass es sich bei DOM um ein implementierungsunabhängiges API handelt. So soll der Programmierer auch gar keine Klassen kennenlernen, die das API implementieren.

Die wichtigsten Schnittstellen

Die zentrale Schnittstelle in *dom* ist `Node`. Sie hat als Unterschnittstellen alle Knotentypen, die es in XML gibt. Folgende Graphik gibt über diese Knotentypen einen Überblick.

```
.

interface org.w3c.dom.Node
 |
 |--interface org.w3c.dom.Attr
 |--interface org.w3c.dom.CharacterData
 |   |
 |   |--interface org.w3c.dom.Comment
 |   |--interface org.w3c.dom.Text
 |       |
 |       |--interface org.w3c.dom.CDATASection
 |
 |--interface org.w3c.dom.Document
 |--interface org.w3c.dom.DocumentFragment
 |--interface org.w3c.dom.DocumentType
 |--interface org.w3c.dom.Element
 |--interface org.w3c.dom.Entity
 |--interface org.w3c.dom.EntityReference
 |--interface org.w3c.dom.Notation
 |--interface org.w3c.dom.ProcessingInstruction
```

Wie man sieht, findet sich die Struktur, die wir in unserem eigenen Versuch, ein XML-Baum-API zu entwickeln, entworfen haben, wieder. Die allgemeine Schnittstelle `Node` und deren beiden Unterschnittstellen `Element` und `Text`.

Eine der entscheidenen Methoden der Schnittstelle `Node` selektiert die Liste der Kinder eines Knotens:
`public NodeList getChildNodes()`

Knoten, die keine Kinder haben können (Textknoten, Attribute etc.) geben bei dieser Methode die leere Liste zurück. Attribute zählen auch wie in unserer Modellierung nicht zu den Kindern eines Knotens. Um an die Attribute zu gelangen, gibt es eine eigene

Methode:

`NamedNodeMap getAttributes()`

Wie man sieht, benutzt Javas *dom* Umsetzung keine von Javas Listenklassen zur Umsetzung einer Knotenliste, sondern nur genau die in *DOM* spezifizierte Schnittstelle `NodeList`. Eine `NodeList` hat genau zwei Methoden:

```
.
int getLength()
Node item(int index)
```

Mit diesen beiden Methoden lässt sich über die Elemente einer Liste über den Indexzugriff iterieren. Die Schnittstelle `NodeList` implementiert nicht die Schnittstelle `Iterable`. Dieses ist insofern schade, da somit nicht die neue *for-each*-Schleife aus Java für die Knotenliste des *DOM* benutzt werden kann.

Die Schnittstelle `Node` enthält ebenso die anderen Methoden, die wir uns vorher in der eigenen Umsetzung überlegt haben:

```
.
String getNodeName();
String getNodeValue();
short getNodeType();
```

Die Methode `getNodeType()` gibt keine Konstante einer Aufzählung zurück, sondern eine Zahl des Typs `short`, die den Knotentyp kodiert. In der Schnittstelle `Node` sind Konstanten für die einzelnen Knotentypen definiert. Die anderen beiden Methoden verhalten sich so, wie wir es in unserer Minimalumsetzung auch implementiert haben.

Die Schnittstelle `Node` des DOM-APIs ist wesentlich umfangreicher als die kleine Schnittstelle `Node`, die wir uns zum Einstieg überlegt haben. Insbesondere gibt es eine Methode, mit der von einem Knoten auch wieder hoch zu seinem Elternknoten navigiert werden kann.

```
.
Node getParentNode()
```

Sofern ein Knoten einen Elternknoten besitzt, kann dieser mit dieser Methode erfragt werden. Ansonsten ist das Ergebnis `null`.

Parser zur Erzeugung von DOM Objekten

Wir benötigen einen Parser, der uns die Baumstruktur eines XML-Dokuments erzeugt. In der Javabibliothek ist ein solcher Parser integriert, allerdings nur über seine Schnittstellenbeschreibung. Im Paket `javax.xml.parsers` gibt es nur Schnittstellen. Um einen

konkreten Parser zu erlangen, bedient man sich einer Fabrikmethode: In der Schnittstelle `DocumentBuilderFactory` gibt es eine statische Methode `newInstance` und über das `DocumentBuilderFactory`-Objekt, läßt sich mit der Methode `newDocumentBuilder` ein Parser erzeugen.

Wir können so eine statischen Methode zum Parsen eines XML-Dokuments schreiben:

ParseXML.java

```
package name.panitz.domtest;

import org.w3c.dom.Document;
import javax.xml.parsers.*;
import java.io.File;

public class ParseXML {
  public static Document parseXml(String xmlFileName){
    try{
     return
      DocumentBuilderFactory
       .newInstance()
       .newDocumentBuilder()
       .parse(new File(xmlFileName));
    }catch(Exception _){return null;}
  }

  public static void main(String [] args){
   System.out.println(parseXml(args[0]));
  }
}
```

Wir können jetzt z.B. den Quelltext dieses Skripts parsen.

```
sep@linux:~/fh/prog4/examples> java -classpath classes/
↪  name.panitz.domtest.ParseXML ../skript.xml
[#document: null]
```

Wie man sieht ist die Methode `toString` in der implementierenden Klasse der Schnittstelle `Document`, die unser Parser benutzt nicht sehr aufschlußreich.

2.5.2 XPath

Für Bäume stellen Pfade ein wichtiges Konzept dar. Entlang der Pfade navigiert man durch einen Baum. Anhand eines Pfades können Teilbäume selektiert werden. Deshalb hat das W3C eine Empfehlung heraus gegeben, mit deren Hilfe Pfade in XML-Dokumenten beschrieben werden können. Die Empfehlung für *XPath*.

Eines der wichtigsten Konzepten von XPath sind die sogenannten Achsen. Sie beschreiben bestimmte Arten sich in einem Baum zu bewegen. XPath kennt 12 Achsen. Eine Achse beschreibt ausgehend von einem Baumknoten eine Liste von Knoten, die diese Achse definiert. Die am häufigste verwendete Achse ist, die Kinderachse. Sie beschreibt die Liste aller direkten Kinder eines Knotens. Entsprechend gibt es die Elternachse, die den Elternknoten beschreibt. Diese Liste hat maximal einen Knoten. Eine sehr simple Achse beschreibt den Knoten selbst. Diese Achse hat immer eine einelementige Liste als Ergebnis. Achsen für Vorfahren und Nachkommen sind der transitive Abschluss der Eltern- bzw. Kinderachse. Zusätzlich gibt es noch Geschwisterachsen, eine für die Geschwister, die vor dem aktuellen Knoten stehen und einmal für die Geschwister, die nach dem aktuellen Knoten stehen. Eine eigene Achse steht für Attribute. Schließlich gibt es noch eine Achse für Namensraumdeklarationen.

Mit einem Aufzählungstypen, lässt sich in Java einfach ein Typ, der alle 12 Achsen aufzählt, implementieren.

Axes.java

```
package name.panitz.xml.xpath;
import name.panitz.pmt.iteration.IntegerRange;
import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.*;



public enum Axes
  {self
  ,child
  ,descendant
  ,descendant_or_self
  ,parent
  ,ancestor
  ,ancestor_or_self
  ,following_sibling
  ,following
  ,preceding_sibling
  ,preceding
  ,attribute;
```

Für jede dieser Achsen soll in der Folge eine Implementierung auf DOM gegeben werden. Eine Achsenimplementierung entspricht dabei einer Funktion, die für einen Knoten eine Liste von Knoten zurück gibt.

Da das DOM-API bereits eine eigene Listenimplementierung hat, wir aber bei den Achsen auf Java-Standardlisten zurückgreifen wollen, sei hier eine simple Konvertierungsmethode von `org.w3c.dom.NodeList` auf `java.util.List` gegeben.

Axes.java

```
public static List<Node> nodelistToList (NodeList nl){
    List<Node> result = new ArrayList<Node>();
    for (int i:new IntegerRange(0,nl.getLength()-1)){
      result.add(nl.item(i));
    }
    return result;
  }
```

Die Achse *self*

Die einfachste Achse liefert genau den Knoten selbst. Die entsprechende Methode liefert die einelementige Liste des Eingabeknotens.

Wir geben für jede Achse zwei Methoden, um deren Ergebnisliste zu erhalten. Die erste Methode erstellt die Ergebnisliste.

Axes.java

```
public static List<Node> self(Node n){
  return self(n,new ArrayList<Node>());
}
```

Die zweite bekommt die Ergebnisliste als Parameter übergeben und fügt in diese die Knoten ein.

Axes.java

```
public static List<Node> self(Node n,List<Node> result){
  result.add(n);
  return result;
}
```

Auf diese Weise kann verhindert werden, dass wenn die Vereinigung mehrerer Achsen berechnet werden soll, oder wenn die Berechnung einer Achse einen rekursive Methode darstellt, unnötig Zwischenlisten erzeugt werden.

Die Achse *child*

Die gebräuchlichste Achse beschreibt die Menge aller Kinderknoten. Wir lassen uns die Kinder des DOM Knotens geben und konvertieren die `NodeList` zu einer Javaliste.

Auch hier werden wieder zwei Versionen der Methode definiert. Einmal wird eine Liste für das ergebnis neu erzeugt, im zweiten Fall wird eine Liste, die die Ergebnisknoten sammelt als Parameter übergeben.

Axes.java

```
public static List<Node> child(Node n){
  return child(n, new ArrayList<Node>());
}
public static List<Node> child(Node n,List<Node> result){
  NodeList nl = n.getChildNodes();
  for (int i:new IntegerRange(0,nl.getLength()-1)){
    result.add(nl.item(i));
  }
  return result;
}
```

Die Achse *descendant*

Die Achse der Nachkommen eines Knotens beschreibt die Liste aller Kinder, Kindeskinder usw. also aller Knoten, die Unterhalb des Ausgangsknotens liegen. Diese sollen in die Ergebnisliste in *preorder* eingefügt werden.

Axes.java

```
public static List<Node> descendant(Node n){
  return descendant(n,new ArrayList<Node>());
}
public static List<Node> descendant(Node n, List<Node> result){
  for (Node child:child(n)){
    result.add(child);
    descendant(child,result);
  }
  return result;
}
```

Die Achse *descendant-or-self*

In der Nachkommenachse taucht der Knoten selbst nicht auf. Diese Achse fügt den Knoten selbst zusätzlich vorne ans Ergebnis an. Es ist also die Vereinigung der Achse *self* mit der Achse *descendant*.

Axes.java

```
public static List<Node> descendantOrSelf(Node n){
  return descendantOrSelf(n,new ArrayList<Node>());
}
public static List<Node> descendantOrSelf(Node n, List<Node> result){
  descendant(n,result);
  result.add(0,n);
  return result;
}
```

Die Achse *parent*

Die Kinderachse lief in den Baum Richtung Blätter eine Ebene nach unten, die Elternachse läuft diese eine Ebene Richtung Wurzel nach oben. Die Ergebnisliste hat maximal ein Element, eventuell 0 Elemente, wenn der Knoten selbst als Elternknoten `null` zurück gibt.

Axes.java

```
public static List<Node> parent(Node n){
  return parent(n, new ArrayList<Node>());
}
public static List<Node> parent(Node n,List<Node> result){
  Node pa = n.getParentNode();
  if (pa!=null && pa.getNodeType()!=Node.DOCUMENT_NODE)
    result.add(pa);
  return result;
}
```

Die Achse *ancestor*

Die Vorfahrenachse ist der transitive Abschluß über die Elternachse, sie bezeichnet also die Eltern und deren Vorfahren. Diese Achse enthält also auf jeden Fall den Wurzelknoten. Sie enthält tatsächlich den Pfad von der Wurzel zum Ausgangsknoten, allerdings ohne den Ausgangsknoten. Dieser ist erst in der Achse *ancestor-or-self* des nächsten Punktes enthalten.

Axes.java

```
public static List<Node> ancestor(Node n){
  return ancestor(n,new ArrayList<Node>());
}
public static List<Node> ancestor(Node n, List<Node> result){
  for (Node pa:parent(n)){
    result.add(pa);
    ancestor(pa,result);
  }
  return result;
}
```

Die Achse *ancestor-or-self*

Ebenso wie für die Achse aller Nachkommen ist auch für die Achse aller Vorfahren eine Version definiert, in der der Ausgangsknoten selbst noch enthalten ist.

Axes.java

```
public static List<Node> ancestorOrSelf(Node n){
  return ancestorOrSelf(n,new ArrayList<Node>());
}
public static List<Node> ancestorOrSelf(Node n, List<Node> result){
  ancestor(n,result);
  result.add(n);
  return result;
}
```

Die Achse *following-sibling*

Auch für die Geschwister eines Knotens sind Achsen definiert. Geschwister sind Knoten, die denselben Elternknoten haben. Die *following-sibling*-Achse definiert alle Geschwister, die in der Kinderliste meines Elternknotens nach mir kommen. Sozusagen alle jüngeren Geschwister.

Axes.java

```
public static List<Node> followingSibling(Node n){
  return followingSibling(n,new ArrayList<Node>());
}
public static List<Node> followingSibling(Node n, List<Node> result){
  Node sib = n.getNextSibling();
  while(sib!=null){
    result.add(sib);
    sib = sib.getNextSibling();
  }
  return result;
}
```

Die Achse *preceding-sibling*

Die *preceding-sibling*-Achse definiert alle Geschwister, die in der Kinderliste meines Elternknotens vor mir kommen. Sozusagen alle älteren Geschwister.

Axes.java

```
public static List<Node> precedingSibling(Node n){
  return precedingSibling(n,new ArrayList<Node>());
}
public static List<Node> precedingSibling(Node n, List<Node> result){
  Node sib = n.getPreviousSibling();
  while(sib!=null){
    result.add(sib);
    sib = sib.getPreviousSibling();
  }
  return result;
}
```

Die Achse *following*

Die Achse *following* bezieht sich auf die *following-sibling*-Achse. Sie bezeichnet alle *following-sibling*-Knoten und deren Nachkommen, sowie dann noch alle weiteren following-siblings der Vorfahren und wieder deren Nachkommen. Textuell bezeichnet diese Achse alle Knoten des XML-Baums, die beginnen, nachdem der Ausgangsknoten beendet ist. Daher auch der Name, alle Knoten die folgen. Mathematisch lässt sich die Achse wie folgt beschreiben:

$$\text{following}(n) = \{f | a \in \text{ancestor-or-self}(n), s \in \text{following-sibling}(a), f \in \text{descendant-or-self}(s)\}$$

Axes.java

```
public static List<Node> following(Node n){
  return following(n,new ArrayList<Node>());
}
public static List<Node> following(Node n, List<Node> result){
  for (Node a:ancestorOrSelf(n))
    for (Node s:followingSibling(a))
      for (Node f:descendantOrSelf(s))
        result.add(f);
  return result;
}
```

Es werden also von allen Vorfahren und dem Knoten selbst die nachfolgenden Geschwister berechnet und diese und deren Nachkommen genommen.

Die Achse *preceding*

Entsprechend bezeichnet die Achse *preceding* von allen Vorfahren die *preceding-sibling*-Knoten und deren Nachkommen. Dieses bedeutet, dass genau die Knoten beschrieben werden, die in der textuellen Darstellung beendet wurden, bevor der Ausgangsknoten begonnen hat..

$$\text{preceding}(n) = \{f | a \in \text{ancestor-or-self}(n), s \in \text{preceding-sibling}(a), f \in \text{descendant-or-self}(s)\}$$

Es werden also von allen Vorfahren und dem Knoten selbst die vorhergehenden Geschwister berechnet und diese und deren Nachkommen genommen.

Axes.java

```
public static List<Node> preceding(Node n){
  return preceding(n,new ArrayList<Node>());
}
public static List<Node> preceding(Node n, List<Node> result){
  for (Node a:ancestorOrSelf(n))
    for (Node s:precedingSibling(a))
      for (Node f:descendantOrSelf(s))
        result.add(f);
  return result;
}
```

Damit sind alle Achsen beschrieben, sie sich auf die Baumstruktur des Dokuments beziehen. Die noch ausbleibende Achse bezieht sich auf die Attribute eines XML-Dokuments, die wir nicht als Teil der Baumstruktur gesehen haben. Alle Knoten eines Dokuments lassen sich durch die Vereinigung der Achsen *preceding*, *ancestor*, *self*, *descendant* und

followong selektieren. Man kann auch als Faustregel sagen, es sind die vier Richtungen, die von einem Baumknoten beschritten werden können: links, über , unter und rechts. *preceding* sind die Knoten links von mir, *ancestor* sind die Knoten über mir, *descendant* sind die Knoten unter mir und *following* die Knoten rechts von mir.

Die Achse *attribute*

Die bisherigen Achsen selektierten Knoten aus der Baumhierarchie eines XML-Dokuments. Die letzten beiden Achsen beziehen sich auf die Attribute. Diese sind nicht teil der Kindknoten eines Elements. Daher werden über die bisherigen Achsen niemals Attribute selektiert. Hierfür gibt es die besondere Achse *attribute*, die die Liste aller Attribute eines Knoten selektiert.

Axes.java

```
  public static  List<Node> attribute(Node n){
   List<Node> result = new ArrayList<Node>();
   NamedNodeMap nl = n.getAttributes();
   if (null!=nl){
      for (int i:new IntegerRange(0,nl.getLength()-1)){
        result.add(nl.item(i));
      }
    }
    return result;
  }

  public  List<Node> select(Node n){
    switch (this){
      case child: return child(n);
      case descendant: return descendant(n);
      case descendant_or_self: return descendantOrSelf(n);
      case parent: return parent(n);
      case ancestor: return ancestor(n);
      case ancestor_or_self: return ancestorOrSelf(n);
      case following_sibling: return followingSibling(n);
      case preceding_sibling: return precedingSibling(n);
      case following: return following(n);
      case preceding: return preceding(n);
      case attribute: return attribute(n);
      default: return self(n);
    }
  }
}
```

Knotentest

Die Kernausdrücke in XPath sind von der Form:

axis::*nodeTest*

axis beschreibt dabei eine der 12 Achsen. *nodeTest* ermöglicht es, aus den durch die Achse beschriebenen Knoten bestimmte Knoten zu selektieren. Syntaktisch gibt es 8 Arten des Knotentests:

- `*`: beschreibt Elementknoten mit beliebigen Namen.
- *pref*:`*`: beschreibt Elementknoten mit dem Prefix *pref* und beliebigen weiteren Namen.
- *pref*:*name*: beschreibt Elementknoten mit dem Prefix *pref* und den weiteren Namen *name*. Der Teil vor den Namen kann dabei auch fehlen.
- `comment()`: beschreibt Kommentarknoten.
- `text()`: beschreibt Textknoten.
- `processing-instruction()`: beschreibt Processing-Instruction-Knoten.
- `processing-instruction(`*target*`)`: beschreibt Processing-Instruction-Knoten mit einem bestimmten Ziel.
- `node()`: beschreibt beliebige Knoten.

Ein XPath-Ausdruck bezieht sich immer auf einen Ausgabgsknoten im Dokument. Die Bedeutung eines XPath-Ausdrucks

axisType::*nodeTest*

ist dann:

Selektiere vom Ausgangsknoten alle Knoten mit der durch *axisType* angegebenen Achse. Von dieser Liste selektiere dann alle Knoten, für die der Knotentest erfolgreich ist.

Der folgende Ausdruck selektiert alle Nachkommen des Ausgangsknoten, die den Elementknoten mit den Tagnamen `code` sind:
`descendant-or-self::code`

Dieses sind in dem XML-Dokument, das dieses Skript beschreibt gerade alle Teile, in denen Java Quelltext steht.

Mehrere XPath-Ausdrücke können laut der XPath Definition nacheinander ausgeführt werden. Hierzu schreibt man zwischen den einzelnen XPath-Audrücken einen Schrägstrich. Der so zusammengesetzte Ausdruck selektiert wieder eine Liste von Knoten ausgehend von einem Ausgangsknoten. Die Bedeutung von einer solchen Verkettung von XPath-Ausdrücken ist:

Selektiere erst mit dem ersten XPath-Teilausdruck ausgehend von einem Startknoten die Liste der Ergebnisknoten. Dann nehme jeden der Knoten in dieser Ergebnisliste als neuen Ausgangsknoten für den zweiten XPath-Ausdruck und vereinige deren Ergebnissse.

Für das Beispiel des XML-Dokuments dieses Skriptes bezeichnet der Ausdruck `child::kapitel/child::se`

Den Text, der direkt unterhalb eines `code` Elements in steht, wobei das `code` Element Nachfahre eines `section` Elements ist, welches direktes Kind eines `kapitel` Elements ist, welches wiederum ein Kind des Ausgangsknotens ist.

Implementierung Nachdem wir bereits alle Achsen selbst implementiert haben, sollen auch die Knotentests implementiert werden. Hierzu können wir eine kleine Schnittstelle vorsehen, die einen Knotentest ausdrückt. Diese ist nichts weiter als ein für Knoten spezialisiertes Prädikat, so dass wir die allgemeine Schnittstelle `Predicate` erweitern können

NodeTest.java

```
package name.panitz.xml.xpath;
import org.w3c.dom.Node;
import org.w3c.dom.Text;
import java.util.function.Predicate;

public interface NodeTest extends Predicate<Node>{
```

Die einfachen Tests, ob es sich um Text, Kommentar oder einen beliebigen Knoten handelt, lassen sich über einen Lambda-Ausdruck definieren.

NodeTest.java

```
  static final NodeTest commentTest = n->n.getNodeType() ==
  ↪  Node.COMMENT_NODE;
  static final NodeTest textTest = n->n instanceof Text;
  static final NodeTest someNodeTest = n->true;
}
```

Für die weiteren Knotentest können Implementierungen dieser Schnittstelle vorgesehen werden. Die erste Klasse vereinigt die drei Knotentest, die sich auf die Elementknoten beziehen:

NameTest.java

```
package name.panitz.xml.xpath;
import org.w3c.dom.Node;

public class NameTest implements NodeTest{
  String prefix;
  String name;
  public NameTest(String prefix,String name){
    this.prefix = prefix;
    this.name = name;
  }
  public NameTest(String name){
    this.prefix = null;
    this.name = name;
  }
  public NameTest(){
    this.prefix = null;
    this.name = null;
  }

  public boolean test(Node n){
    if (n.getNodeType() != Node.ELEMENT_NODE
        && n.getNodeType() != Node.ATTRIBUTE_NODE) return false;

    if (null ==prefix && null==name) return true;
    if (null==prefix) return n.getNodeName().equals(name);
    if (null==name) return n.getPrefix().equals(prefix);
    return n.getPrefix().equals(prefix) && n.getLocalName().equals(name);
  }
  @Override public String toString(){
    if (null ==prefix && null==name) return "*";
    if (null==prefix) return name;
    if (null==name) return prefix+":*";
    return prefix+":"+name;
  }
}
```

Die folgende Klasse realisiert die Knotentests, die sich auf precessing instructions beziehen.

ProcessingInstructionTest.java

```
package name.panitz.xml.xpath;
import org.w3c.dom.Node;
import org.w3c.dom.ProcessingInstruction;

public class ProcessingInstructionTest implements NodeTest{
  String target;
  public ProcessingInstructionTest(){
    target = null;
  }
  public ProcessingInstructionTest(String target){
    this.target = target;
  }
  public boolean test(Node n){
    if (n.getNodeType() != Node.PROCESSING_INSTRUCTION_NODE) return
    ↪  false;
    if (null!=target){
      return target.equals(((ProcessingInstruction)n).getTarget());
    }
    return true;
  }
  @Override public String toString(){
    if (null==target) return "processing-instruction()";
    return  "processing-instruction("+target+")";
  }
}
```

Mit den Achsen von XPath und den Knotentests, lassen sich nun XPath-Ausdrücke als Paar von einer Achse und einem Test darstellen.

XPathExpr.java

```
package name.panitz.xml.xpath;
import org.w3c.dom.Node;
import java.util.List;
import java.util.ArrayList;

public class XPathExpr{
  Axes axis;
  NodeTest test;

  public XPathExpr(Axes axis, NodeTest test){
    this.axis = axis;
    this.test = test;
  }
  public List<Node> select(Node n){
    List<Node> result = new ArrayList<>();
    for (Node x:axis.select(n)){
      if (test.test(x)) result.add(x);
    }
    return result;
  }
  @Override public String toString(){
    return axis+"::"+test;
  }
}
```

Ein ganzer XPath-Ausdruck stellt eine Folge von einfachen Ausdrücken da, die nacheinander zur Selektion benutzt werden.

XPath.java

```
package name.panitz.xml.xpath;
import org.w3c.dom.Node;
import java.util.List;
import java.util.ArrayList;

public class XPath extends ArrayList<XPathExpr>{
  public List<Node> select(Node n){
    List<Node> result = new ArrayList<>();
    result.add(n);
    for (XPathExpr xpath:this){
      List<Node> result2 = new ArrayList<>();
      for (Node x:result){
        result2.addAll(xpath.select(x));
      }
     result = result2;
    }
    return result;
  }

  @Override public String toString(){
    StringBuffer result = new StringBuffer();
    boolean first = true;
    for (XPathExpr xpath:this){
      if (first){
        first = false;
      }else{
        result.append("/");
      }
      result.append(xpath.toString());
    }
    return result.toString();
  }

}
```

Abkürzende Schreibweise

In obiger Einführung haben wir bereits gesehen, dass XPath den aus dem Dateissystem bekannten Schrägstrich für Pfadangaben benutzt. Betrachten wir XPath-Ausdrücke als Terme, so stellt der Schrägstrich einen Operator dar. Diesen Operator gibt es sowohl einstellig wie auch zweistellig.

Die Grundsyntax in XPath sind eine durch Schrägstrich getrennte Folge von Kernausdrücke mit Achsentyp und Knotentest.

Der Ausdruck

`child::skript/child::*/descendant::node()/self::code/attribute::class`

beschreibt die Attribute mit Attributnamen `class`, die an einem Elementknoten mit Tagnamen `code` hängen, die Nachkommen eines beliebigen Elementknotens sind, die Kind eines Elementknotens mit Tagnamen `skript` sind, die Kind des aktuellen Knotens, auf dem der Ausdruck angewendet werden soll sind.

Vorwärts gelesen ist dieser Ausdruck eine Selektionsanweisung:
Nehme alle `skript`-Kinder des aktuellen Knotens. Nehme von diesen beliebige Kinder. Nehme von diesen alle `code`-Nachkommen. Und nehme von diesen jeweils alle `class`-Attribute.

Der einstellige Schrägstrichoperator bezieht sich auf die Dokumentwurzel des aktuellen Knotens.

Der doppelte Schrägstrich XPath kennt einen zweiten Pfadoperator `//`. Auch er existiert jeweils einmal einstellig und einmal zweistellig. Der doppelte Schrägstrich ist eine abkürzende Schreibweise, die übersetzt werden kann in Pfade mit einfachen Schrägstrichoperator.

- `//`*expr*: Betrachte beliebige Knoten unterhalt des Dokumentknotens, die durch *expr* charakterisiert werden.
- e_1`//`e_2: Betrachte beliebiege Knoten unterhalb der durch e_1 charkterisierten Knoten.und prüfe diese auf e_2.

Der Doppelschrägstrich ist eine abkürzende Schreibweise für: `/descendant-or-self::node()/`

Der Punkt Für die Selbstachse kann als abkürzende Schreibweise ein einfacher Punkt `.` gewählt werden, wie er aus den Pfadangaben im Dateisystem bekannt ist.

Der Punkt `.` ist die abkürzende Schreibweise für: `self::node()`.

Der doppelte Punkt Für die Elternachse kann als abkürzende Schreibweise ein doppelter Punkt `..` gewählt werden, wie er aus den Pfadangaben im Dateisystem bekannt ist.

Der Punkt `..` ist die abkürzende Schreibweise für: `parent::node()`.

Vereinfachte Kindachse Ein Kernausdruck, der Form `child::`*nodeTest* kann abgekürzt werden durch *nodeTest*. Die Kinderachse ist also der Standardfall.

Vereinfachte Attributachse Auch für die Attributachse gibt es eine abkürzende Schreibweise: ein Kernausdruck, der Form `attribute::`*pre*`:`*name* kann abgekürzt werden durch `@`*pre*`:`*name*.

Insgesamt läßt sich der obige Ausdruck abkürzen zu: `skript/*//code/@class`

XPath-Auswertung in Java

Im Java-API ist eine XPath-Implementierung enthalten. Die dazugehörigen Klassen befinden sich im Paket `javax.xml.xpath`. Die Implementierung basiert auf DOM, es werden also DOM-Knoten als Eingabe und genommen und auch u.a. eine DOM-Nodelist als Ergebnis berechnet.

Im Folgendem ein Beispiel, wie ein XPath-Ausdruck, der eine Knotenliste als Ergebnis hat, mit dem Java-API für XPath angewendet wird. Wir schreiben eine statische Methode, die einen XPath-Ausdruck als String übergeben bekommt und das Eingabedokument für diesen Ausdruck als einen Reader.

XPathTest.java

```
package name.panitz.xml;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.Reader;

import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;

import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

public class XPathTest {
  private static NodeList evalXPath(String xpathExpression, Reader
  ↪  reader)
      throws XPathExpressionException {
```

Zunächst einmal benutzt das XPath-API wieder das Factory-Pattern, so dass ein XPath-Ausdruck nicht direkt durch einem Konstruktoraufruf erzeugt wird, sondern durch eine Methode in einer sogenannten Factory-Klasse, ebenso wie auch der DOM-Parser erzeut wurde.

XPathTest.java

```
XPathFactory fact = XPathFactory.newInstance();
XPath xpath = fact.newXPath();
```

Die Schnittstelle `XPath` kennt eine Methode `evaluate`, die drei Parameter hat:

> der auszuwertende XPath-Ausdruck als String. die Eingabequelle für das XML-Dokuement, auf dem dieser Ausdruck ausgewertet werden soll. eine Konstante, die das erwartete Ergebnis anzeigt. Für die meisten XPath-Ausdrücke wird ein Objekt

des Typs `Nodelist` als Ergebnis erwartet. Es gibt aber auch XPath-Ausdrücke, die nur Strings, oder Zahlen als Ergebnis haben.

Das Ergebnis der Methode `evaluate` ist nur vom Typ `Object` und muss entsprechend erst noch mit einer Typzusicherung auf den eigentlichen Ergebnistypen überprüft werden.

In unserem Beispiel gehen wir davon aus, dass das Ergebnis eine `Nodelist` ist. Falls dieses nicht der Fall ist, wird eine Ausnahme geworfen.

XPathTest.java

```
  NodeList ns = (NodeList) xpath.evaluate(xpathExpression,
      new InputSource(reader), XPathConstants.NODESET);
  return ns;
}
```

Abschließend ein kleiner Testaufruf dieser Methode, der aus der XML-Datei, die diesem Skript zugrunde liegt, alle code-Elemente selektiert:

XPathTest.java

```
  public static void main(String[] args) throws XPathExpressionException,
      FileNotFoundException {
    String xpathExpression = "//code";
    Reader reader = new FileReader("skript.xml");

    NodeList ns = evalXPath(xpathExpression, reader);
    for (int i = 0; i < ns.getLength(); i++) {
      System.out.println(ns.item(i).getTextContent());
    }
  }
}
```

2.5.3 XSLT

XML-Dokumente enthalten keinerlei Information darüber, wie sie visualisiert werden sollen. Hierzu kann man getrennt von seinem XML-Dokument ein sogenanntes *Stylesheet* schreiben. XSL ist eine Sprache zum Schreiben von Stylesheets für XML-Dokumente. XSL ist in gewisser Weise eine Programmiersprache, deren Programme eine ganz bestimmte Aufgabe haben: XML Dokumente in andere XML-Dokumente zu transformieren. Die häufigste Anwendung von XSL dürfte sein, XML-Dokumente in HTML-Dokumente umzuwandeln.

Wir werden die wichtigsten XSLT-Konstrukte mit folgendem kleinem XML Dokument ausprobieren:

.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<?xml-stylesheet type="text/xsl" href="cdTable.xsl"?>
<cds>
 <cd>
  <artist>The Beatles</artist>
  <title>White Album</title>
  <label>Apple</label>
 </cd>
 <cd>
  <artist>The Beatles</artist>
  <title>Rubber Soul</title>
  <label>Parlophone</label>
 </cd>
 <cd>
  <artist>Duran Duran</artist>
  <title>Rio</title>
  <label>Tritec</label>
 </cd>
 <cd>
  <artist>Depeche Mode</artist>
  <title>Construction Time Again</title>
  <label>Mute</label>
 </cd>
 <cd>
  <artist>Yazoo</artist>
  <title>Upstairs at Erics</title>
  <label>Mute</label>
 </cd>
 <cd>
  <artist>Marc Almond</artist>
  <title>Absinthe</title>
  <label>Some Bizarre</label>
 </cd>
 <cd>
  <artist>ABC</artist>
  <title>Beauty Stab</title>
  <label>Mercury</label>
 </cd>
</cds>
```

Gesamtstruktur

XSLT-Skripte sind syntaktisch auch wieder XML-Dokumente. Ein XSLT-Skript hat feste Tagnamen, die eine Bedeutung für den XSLT-Prozessor haben. Diese Tagnamen haben einen festen definierten Namensraum. Das äußerste Element eines XSLT-Skripts hat den Tagnamen `stylesheet`. Damit hat ein XSLT-Skript einen Rahmen der folgenden Form:

.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:stylesheet>
```

Wir können mit einer *Processing-Instruction* am Anfang eines XML-Dokumentes definieren, mit welchem XSLT Stylesheet es zu bearbeiten ist. Hierzu wird als Referenz im Attribut `href` die XSLT-Datei angegeben.

.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<?xml-stylesheet type="text/xsl" href="cdTable.xsl"?>
<cds>
 <cd>........
```

Vorlagen (templates)

Das wichtigste Element in einem XSLT-Skript ist das Element `xsl:template`. In ihm wird definiert, wie ein bestimmtes Element transformiert werden soll. Es hat schematisch folgende Form:
`<xsl:template match="`*Elementname*`">`
zu erzeugender Code
`</xsl:template >`

Folgendes XSLT-Skript transformiert die XML-Datei mit den CDs in eine HTML-Tabelle, in der die CDs tabellarisch aufgelistet sind.

.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!-- Startregel für das ganze Dokument. -->
<xsl:template match="/">
  <html><head><title>CD Tabelle</title></head>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>


<!-- Regel für die CD-Liste. -->
<xsl:template match="cds">
  <table border="1">
    <tr><td><b>Interpret</b></td><td><b>Titel</b></td></tr>
    <xsl:apply-templates/>
  </table>
</xsl:template>

<xsl:template match="cd">
  <tr><xsl:apply-templates/></tr>
</xsl:template>

<xsl:template match="artist">
  <td><xsl:apply-templates/></td>
</xsl:template>

<xsl:template match="title">
  <td><xsl:apply-templates/></td>
</xsl:template>

<!-- Regel für alle übrigen Elemente.
     Mit diesen soll nichts gemacht werden -->
<xsl:template match="*">
</xsl:template>

</xsl:stylesheet>
```

Öffnen wir nun die XML Datei, die unsere CD-Liste enthält im Webbrowser, so wendet er die Regeln des referenzierten XSLT-Skriptes an und zeigt die so generierte Webseite als Tabelle an.

Läßt man sich vom Browser hingegen den Quelltest der Seite anzeigen, so wird kein

HTML-Code angezeigt sondern der XML-Code.

XSLT in Java

Ebenso wie für XPath gibt es auch eine Implementierung von XSLT in Java. Die entsprechenden Klassen befinden sich im Paket `javax.xml.transform`. Auch hier wird wieder das Factory-Pattern angewendet. Wir geben im Folgendem eine kleine Beispielanwendung, in der ein XSLT-Skript auf ein XML-Dokument angewendet wird.

Interessant ist, dass es auch möglich ist eine Instanz vom Typ `Transformer` ohne Angabe eines XSL-Skripts als Quelle zu erzeugen. Dieses entspricht dann der Identität, dass heißt, das transformierte Ausgabeausgabedokument ist gleich dem zu transformierenden Eingabedokument. Damit kann dieser Identittäts-Transformer dazu genutzt werden, um einen DOM-Baum wieder als textuelle Datei zu speichern.

XSLT.java

```
package name.panitz.xml.xslt;

import org.w3c.dom.Node;

import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.TransformerException;

import java.io.StringWriter;
import java.io.File ;

public class XSLT {
  static public String transform(File xslt,File doc){
    return transform(new StreamSource(doc),new StreamSource(doc));
  }

  static public String transform(Source xslt,Source doc){
    try{
      StringWriter writer = new StringWriter();
      Transformer t =
         TransformerFactory
          .newInstance()
          .newTransformer(xslt);
      t.transform(doc,new StreamResult(writer));
      return writer.getBuffer().toString();
    }catch (TransformerException _){
      return "";
    }
  }

  public static void main(String [] args)throws Exception {
    System.out.println(transform(new File(args[0]),new File(args[1])));
  }

}
```

2.5.4 Das SAX-API

Oft brauchen wir nie das komplette XML-Dokument als Baum im Speicher. Eine Großzahl der Anwendungen auf XML-Dokumenten geht einmal das Dokument durch, um ir-

gendwelche Informationen darin zu finden, oder ein Ergebnis zu erzeugen. Hierzu reicht es aus, immer nur einen kleinen Teil des Dokuments zu betrachten. Und tatsächlich hätte diese Vorgehensweise, bai allen bisher geschriebenen Programmen gereicht. Wir sind nie im Baum hin und her gegangen. Wir sind nie von einem Knoten zu seinem Elternknoten oder seinen vor ihm liegenden Geschwistern gegangen.

Ausgehend von dieser Beobachtuung hat eine Gruppe von Programmierern ein API zur Bearbeitungen von XML-Dokumenten vorgeschlagen, das nie das gesammte Dokument im Speicher zu halten braucht. Dieses API heißt *SAX*, für *simple api for xml processing*. *SAX* ist keine Empfehlung des W3C. Es ist außerhalb des W3C entstanden.

Die Idee von *SAX* ist ungefähr die, daß uns jemand das Dokument vorliest, einmal von Anfang bis Ende. Wir können dann auf das gehörte reagieren. Hierzu ist für einen Parse mit einem SAX-Parser stets mit anzugeben, wie auf das Vorgelesene reagiert werden soll. Dieses ist ein Objekt der Klasse `DefaultHandler`. In einem solchen *handler* sind Methoden auszuprogrammieren, in denen spezifiziert ist, was gemacht werden soll, wenn ein Elementstarttag, Elementendtag, Textknoten etc. vorgelsen wird. Man spricht bei einem SAX-Parser von einem ereignisbasierten Parser. Wir reagieren auf bestimmte Ereignisse des Parses, nämlich dem Starten/Enden von Elementen und so weiter.

Auch ein SAX-Parser liegt in Java nur als Schnittstelle vor und kann nur über eine statische Fabrikmethode instanziiert werden.

SaxParse.java

```
package name.panitz.xml.sax;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;
import org.xml.sax.*;
import java.io.File;

public class SaxParse{
  public static void parse(File file,DefaultHandler handler)
                                  throws Exception{
    SAXParserFactory.newInstance()
                    .newSAXParser()
                    .parse(file,handler);
  }
}
```

Als erstes Beispiel wollen wir unser altbekanntes Zählen der Knoten programmieren. Hierzu ist ein eigener `DefaultHandler` zu schreiben, der, sobald beim Vorlesen ihm der Beginn eines Elements gemeldet wird, darauf reagiert, indem er seinen Zähler um eins weiterzählt. Wir überschreiben demnach genau eine Methode aus dem `DefaultHandler`, nämlich die Methode `startElement`.

Count.java

```
package name.panitz.xml.sax;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class Count extends DefaultHandler {
  int result = 0;

  @Override
  public void startElement(String uri, String localName, String qName,
      Attributes attributes) throws SAXException {
    result++;
  }

  public static void main(String [] args) throws Exception{
    Count counter = new Count();
    SaxParse.parse(new java.io.File(args[0]),counter);
    System.out.println(counter.result);
  }
}
```

GetTagNames.java

```
package name.panitz.xml.sax;

import java.util.Set;
import java.util.TreeSet;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class GetTagNames extends DefaultHandler {
  Set<String> result = new TreeSet<>();

  @Override
  public void startElement(String uri, String localName, String qName,
      Attributes attributes) throws SAXException {
    result.add(qName);
  }

  public static void main(String [] args) throws Exception{
    GetTagNames tagCollector = new GetTagNames();
    SaxParse.parse(new java.io.File(args[0]),tagCollector);
    System.out.println(tagCollector.result);
  }

}
```

GetProgramCode.java

```
package name.panitz.xml.sax;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class GetProgramCode extends DefaultHandler {
  StringBuffer result = new StringBuffer();
  boolean isInCodeElement = false;

  @Override
  public void startElement(String uri, String localName, String qName,
      Attributes attributes) throws SAXException {
    if (qName.equals("code")) {
      isInCodeElement = true;
    }
  }

  @Override
  public void characters(char[] ch, int start, int length)
      throws SAXException {
    String text = new String(ch, start, length);
    if (isInCodeElement)
      result.append(text);
  }

  @Override
  public void endElement(String uri, String localName, String qName)
      throws SAXException {
    if (qName.equals("code")) {
      isInCodeElement = false;
    }
  }

  public static void main(String [] args) throws Exception{
    GetProgramCode codeExtractor = new GetProgramCode();
    SaxParse.parse(new java.io.File(args[0]),codeExtractor);
    System.out.println(codeExtractor.result);
  }
}
```

2.5.5 Das StAx-API

Wir haben bisher drei methodisch sehr unterschiedliche Herangehensweisen für die Programmierung von XML Dokumenten kennen gelernt:

- das DOM-API als ein Baum-API.
- das SAX-API als ein Ereignis-basiertes API.
- XPath als spezielle Programmiersprache, zur Beschreibung der Selektion von Knoten aus der Baumstruktur des Dokuments.

Als Abschluss betrachten wir noch eine vierte methodische Herangehensweise. Dabei schließen wir wieder den Kreis zum ersten Kapitel, nämlich zu den Iteratoren. Ein weiteres API zur XML Verarbeitung in Java ist das strombasierte API StAX. Dahinter verbirgt sich wie bei SAX, die Verarbeitung des Dokuments sequentiell in seiner textuellen Form. Damit steht wieder nicht die reine Baumstruktur direkt im Speicher zur Verfügung. Die sequentielle Verarbeitung wird durch einen Iterator über die verschiedenen XML-Ereignisse bewerkstelligt.

In Java finden sich die entsprechenden Definitionen als Schnittstellen im Paket `javax.xml.stream`. Die entscheidenen beiden Schnittstellen sind dabei:

- `XMLEventReader:` diese Schnittstelle beschreibt einen Iterator der beim Lesen eines XML-Dokuments über die einzelnen XML-Komponenten des Dokuments iteriert. Leider berücksichtigt diese Schnittstelle nicht die generischen Typen von Java, so dass die Methode `next()` als Rückgabetyp lediglich `Object` hat.
- `javax.xml.stream.events.XMLEvent`: diese Schnittstelle beschreibt allgemein ein XML-Konstrukt, auf dem beim Lesen gestoßen wird. Für die verschiedenen XML-Anteile gibt es Unterschnittstellen dieser Schnittstelle wie z.B. `StartElement`, `EndElement` oder `Characters`. In der Schnittstelle `XMLEvent` sind boolsche Methoden definiert, die testen, ob ein Objekt von einer dieser Unterschnittstellen ist. Es gibt z.B. die Methode `boolean isAttribute();`.

Die Anwendung des StAX-APIs geht sehr analog, zu der Anwendung der anderen beiden XML APIs.

StaxTest.java

```
package name.panitz.xml.stax;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.Iterator;

import javax.xml.stream.FactoryConfigurationError;
import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.events.XMLEvent;

public class StaxTest {

  public static void main(String[] args)
      throws FileNotFoundException
      , XMLStreamException
      , FactoryConfigurationError {
```

Ein StAX basierter Parser wird durch eine Factory instanziiert:

StaxTest.java

```
final XMLEventReader staxReader
  = XMLInputFactory
    .newInstance()
    .createXMLEventReader(new FileReader(args[0]));


int result=0;
```

Das so erhaltene `XMLEventReader` Objekt kann zum Iterieren durch das Dokument benutzt werden. Leider kennt das StAX-API nicht die Schnittstelle `Iterable`, so dass nicht direkt die for-each Schleife zum Iterieren angewendet werden kann. Wir können uns da mit der Erzeugung eines iterierbaren Objektes durch einen Lambda-Ausdruck behelfen:

StaxTest.java

```
Iterable<XMLEvent> it = () -> staxReader;
for (XMLEvent ev : it){
```

Für dieses Beispiel wollen wir die Anzahl der Elementknoten zählen. Ist das iterierte Objekt der Anfang eines Elements, also ein Start-Tag, dann zählen wir die Ergebnisvariable um eins hoch.

StaxTest.java

```
      if (ev.isStartElement()) result++;
    }
    System.out.println(result);
  }
}
```

3 Eine graphische Komponente für Baumstrukturen

Zum Abschluss dieses Kapitels über Baumstrukturen, sollen die Bäume auch in einer graphischen Komponente angezeigt werden können. Fast jede GUI-Bibliothek hat eine Komponente zur Darstellung von hierarchischen Strukturen, um zum Beispiel das Dateisystem anzeigen zu können. Im Swing-API ist dieses die Klasse `JTree`. Die Klasse `JTree` hat einen Konstruktor, dem man einen Baumknoten übergeben kann. Hierzu kennt das API eine Schnittstelle `TreeNode`, die allgemein die verlangten Eigenschaften eines Baumknotens in sieben Methoden beschreibt.

Allerdings kann man in der regel nicht davon ausgehen, dass eine Baumstruktur so entworfen wurde, dass sie diese Schnittstelle implementiert. Die Schnittstelle `Node` des DOM-APIs kennt die Schnittstelle `TreeNode` aus Swing genauso wenig, wie unsere Klasse `Tree`.

Deshalb gibt es eine zweite, flexiblere Möglichkeit, um der graphischen Komponente `JTree` die Baumdaten zu übergeben. Hierzu definiert Swing die Schnittstelle `TreeModel`.

MyTreeModel.java

```
package name.panitz.util;
import javax.swing.event.TreeModelListener;
import javax.swing.tree.TreeModel;
import javax.swing.tree.TreePath;

public class MyTreeModel implements TreeModel{
  Tree<?> tree;
  public MyTreeModel(Tree<?> tree) {
    this.tree = tree;
  }

  @Override
  public Object getRoot() {
    return tree;
  }

  @Override
  public Object getChild(Object parent, int index) {
    Tree<?> t = (Tree<?>)parent;
    return t.childNodes.get(index);
  }

  @Override
  public int getChildCount(Object parent) {
    Tree<?> t = (Tree<?>)parent;
    return t.childNodes.size();
  }

  @Override
  public boolean isLeaf(Object node) {
    Tree<?> t = (Tree<?>)node;
    return t.childNodes.isEmpty();
  }

  @Override
  public void valueForPathChanged(TreePath path, Object newValue) {
    throw new UnsupportedOperationException();
  }

  @Override
  public int getIndexOfChild(Object parent, Object child) {
    Tree<?> t = (Tree<?>)parent;
    int i = 0;
    for (Tree<?> c:t.childNodes){
      if (c==child) return i;
      i++;
    }
    return -1;
  }

  @Override
  public void addTreeModelListener(TreeModelListener l) {  }

  @Override
  public void removeTreeModelListener(TreeModelListener l) {  }
}
```

Bei genauer Betrachtung dieser Umsetzung stellt man fest, dass die Schnittstelle `TreeModel` leider kein Gebrauch von generischen Typen macht. Daher sind wir gezwungen mit Typzusicherungen zu arbeiten. Wir können aber eine abstrakte Klasse vorsehen, die Gebrauch von generischen Typen macht, und die `TreeModel` implementiert. Die Klasse bekommt einen Typparameter für den Typ der Baumknoten.

Die entsprechenden Typzusicherungen werden dann in der abstrakten Klasse durchgeführt. Zwei abstrakte Methoden bekommen dann eine Signatur, die direkt Parameter der Baumknoten erhält.

AbstractTreeModell.java

```java
package name.panitz.util;

import javax.swing.event.TreeModelListener;
import javax.swing.tree.TreeModel;
import javax.swing.tree.TreePath;

public abstract class AbstractTreeModell<T> implements TreeModel{
  T tree;

  public AbstractTreeModell(T tree) {
    super();
    this.tree = tree;
  }

  @Override
  public T getRoot() {
    return tree;
  }

  @Override
  public T getChild(Object parent, int index) {
    T t = (T)parent;
    return getChildAtIndex(t,index);
  }

  abstract public T getChildAtIndex(T parent, int index);


  @Override
  public int getChildCount(Object parent) {
    T t = (T)parent;
    return getChildSize(t);
  }

  abstract public int getChildSize(T t);

  @Override
  public boolean isLeaf(Object node) {
    return getChildCount(node)==0;
  }

  @Override
  public void valueForPathChanged(TreePath path, Object newValue) {
    throw new UnsupportedOperationException();
  }

  @Override
  public int getIndexOfChild(Object parent, Object child) {
    T t = (T)parent;
    T c = (T)child;
    return getChildIndex(t,c);
  }

  public int getChildIndex(T t, T c){
    for (int i=0; i<getChildCount(t);i++){
      if (c == getChild(t, i)) return i;
    }
    return -1;
  }
```

Um jetzt für eine Baumklasse ein Modell zu bekommen, reicht es aus, von der abstrakten Klasse abzuleiten und zwei naheliegende Methoden zu implementieren. Eine Methode, um die Anzahl der Kinder zu erfragen, eine um ein bestimmtes Kind zu erhalten.

TreeTreeModel.java

```
package name.panitz.util;
import javax.swing.JTree;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;

public class TreeTreeModel extends AbstractTreeModell<Tree<?>>{

  public TreeTreeModel(Tree<?> tree){
    super(tree);
  }

  @Override public Tree<?> getChildAtIndex(Tree<?> parent, int index) {
    return parent.childNodes.get(index);
  }

  @Override  public int getChildSize(Tree<?> parent) {
    return parent.childNodes.size();
  }

  public static void main(String[] args){
    javax.swing.JFrame f = new javax.swing.JFrame();
    JTree jt = new JTree(new TreeTreeModel(Tree.c1));
    jt.addTreeSelectionListener(new TreeSelectionListener() {
      @Override
      public void valueChanged(TreeSelectionEvent e) {
  Object n = e.getPath().getLastPathComponent();
        System.out.println(n);
      }
    });
    f.add(jt);
    f.pack();
    f.setVisible(true);
  }
}
```