Iteratoren

Sven Eric Panitz

7. April 2021

Inhaltsverzeichnis

1 Iterator und Iterable

1.1 Objekte für Schleifen

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 diesem Lehrbrief 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:

Java: Iteration1

```
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:

Java: GeradeZahlenKleiner10Iterierer

```
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.

Java: GeradeZahlenKleiner10Iterierer

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

Mit einem Objekt dieser Klasse können wir jetzt eine Schleife durchlaufen, ebenso wie

in der ersten Schleife oben:

Java: GeradeZahlenKleiner10Iterierer

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

1.2 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:

Java: IntIterierer

```
package name.panitz.pmt.iteration;
import java.util.function.Consumer;

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

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.

Java: IntIterierer

```
  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.

Java: GeradeZahlenIterierer

```
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.

Java: GeradeZahlenIterierer

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

1.3 Generizität

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:

Java: GenerischerIterierer

```
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 bezeichnet.

Die neue Implementierung der Iterationsklasse sieht entsprechend wie folgt aus:

Java: GeradeZahlenIterierer2

```
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.

Java: GeradeZahlenIterierer2

```
  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.

1.4 Die Standardschnittstelle `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:

Java: PmtIterator

```
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()`:

Java: PmtIterator

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

Man sieht genau die Bedeutung der Methode `next()`. Zum einen wird der aktuelle Schleifenwert 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.

Java: PmtIterator

```
  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.

Java: IntegerRangeIterator

```
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.

Java: IntegerRangeIterator

```
  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:

Java: IntegerRangeForEach

```
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));
  }
}
```

1.5 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.

Java: IntegerRange

```
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:

Java: IntegerRange

```
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.

Java: IntegerRange

```
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:

Java: IntegerRange

```
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:

Java: IntegerRange

```
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.

Java: IntegerRange

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

1.6 Iteratoren als innere Klassen

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 ist dann die Iteratorklasse komplett versteckt:

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

Java: IntegerRange

```
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:

Java: IntegerRange

```
public Iterator<Integer> iterator(){
  return new IntegerRangeIterator();
}
```

Das Iterator-Objekt ist dabei von einer inneren Klasse, die beschreibt, wie durch die Elemente iteriert wird. Die innere Klasse kann auf Felder des Objektes der äußeren Klasse mit `IntegerRange.this` zugreifen.

Java: IntegerRange

```
  private class IntegerRangeIterator implements Iterator<Integer>{
    int from=IntegerRange.this.from;
    int to=IntegerRange.this.to;
    int step=IntegerRange.this.step;
    public boolean hasNext(){
      return from < to;
    }
    public Integer next(){
      int result  = from;
      from = from + step;
      return result;
    }
  }
}
```

1.7 Iteratoren als anonyme innere Klassen

Wer es noch kürzer liebt, braucht für die private innere Klasse im obigen Beispiel noch nicht einmal mehr einen Namen, sondern kann diese direkt als anonyme Klassen definieren.

Zusätzlich leisten wir uns den Luxus einer Record-Klasse, um mit möglichst wenig Code auszukommen.

Java: AnonIntegerRange

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

public record AnonIntegerRange(int from, int to, int step)
                               implements Iterable<Integer>{
  public Iterator<Integer> iterator(){
    return new Iterator<>(){
      int from=AnonIntegerRange.this.from;
      int to=AnonIntegerRange.this.to;
      int step=AnonIntegerRange.this.step;
      public boolean hasNext(){ return from < to;}
      public Integer next(){
        int result  = from;
        from = from + step;
        return result;
      }
    };
  }
}
```

2 Aufgaben

Für diesen Lehrbrief sind eine Vielzahl von iterierbaren Objekten zu implementieren. Wir werden dabei einige Klasse und Schnittstellen benötigen, die zunächst einmal importiert werden.

Java: Iteratoren

```java
package name.panitz.util;
import java.util.Iterator;
import java.math.BigInteger;
import java.util.function.Function;
import java.util.function.Predicate;
```

Um mit einer Quelltextdatei auszukommen, ist die komplette Aufgabe in einer Schnittstelle gebündelt und die einzelnen Aufgaben sind als innere statische Klassen dieser Schnittstelle zu implementieren.

Java: Iteratoren

```java
public interface Iteratoren  {
```

Die Lösungen lassen sich dann driekt in der JShell ausprobieren, wenn man alle statischen Eigenschaften, also auch die statischen Klassen der Schnittstelle als erstes importiert:

Shell

```
jshell --enable-preview --class-path classes
|  Welcome to JShell -- Version 15.0.1
|  For an introduction type: /help intro

jshell> import static name.panitz.util.Iteratoren.*

jshell>
```

Aufgabe 1 Schreiben Sie noch einmal, wie bereits in diesem Lehrbrief vorgemacht, eine Iterable-Klasse, die es es erlaubt über einen Zahlenbereich zu iterieren. Nennen Sie die Klasse `IntRange` und sehen Sie viele verschiedene Konstruktoren vor. Berücksichtigen Sie jetzt auch noch, dass mit einem negativen Schritt von einer größeren Zahl rückwärts zu einer kleineren Zahl iteriert werden kann:

Außerdem soll es möglich sein ohne Ende potentiell unendlich oft zu iterieren. Auch wenn die größte oder kleinste int-Zahl erreicht wurden und mit dem Zahlenüberlauf weiter gearbeitet wird.

- `IntRange()`: es wird endlos über alle int Werte beginnend bei 0 iteriert.

- `IntRange(int from)`: es wird endlos über alle int Werte beginnend bei `from` iteriert.
- `IntRange(int from, int to)`: es wird beginnend bei `from` bis einschließlich `to` iteriert.
- `IntRange(int from, int to, int step)`: es wird beginnend bei `from` bis einschließlich `to` in `step` Schritten iteriert.
  Wenn `step` eine negative Zahl ist, wird so lange iteriert, bisdie nächste Zahl kleiner als `to` ist, wenn `step` positiv ist, solange bis die nächste Zahl größer als to ist.

Java: Iteratoren

```java
public static class IntRange implements Iterable<Integer>{
  int from;
  int to;
  int step;
  boolean infinite;
  public IntRange(int from,  int to,  int step){/*ToDo*/}
  public IntRange(int from,  int to){/*ToDo*/}
  public IntRange(int from){/*ToDo*/}
  public IntRange(){/*ToDo*/}
}
```

Ein Beispielaufruf:

Shell

```
jshell> new IntRange(1,100,10).forEach(x->System.out.print(x+" "))
1 11 21 31 41 51 61 71 81 91
jshell> new IntRange(100,1,-9).forEach(x->System.out.print(x+" "))
100 91 82 73 64 55 46 37 28 19 10 1
```

Aufgabe 2 Schreiben Sie eine Klasse `Fib`, die `Iterable<BigInteger>` so implementiert, dass beim Aufruf der Methode `next` nacheinander die Fibonaccizahlen zurück gegeben werden. Dabei soll die erste Fibonaccizahl die 0 sein.
Es soll also über $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ iteriert werden.

Java: Iteratoren

```
public static class Fib implements Iterable<BigInteger> {
  @Override public Iterator<BigInteger> iterator(){
    return null; /*ToDo*/
  }

  public static void main(String[] args){
    new Fib().forEach(x-> System.out.println(x));
  }
}
```

Ein Beispielaufruf:

Shell

```
jshell> var fib = new Fib().iterator()
fib ==> name.panitz.util.Iteratoren$Fib$MyIt@46f7f36a

jshell> new IntRange(1,20).forEach(i->System.out.print(fib.next()+" "))
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Aufgabe 3 Schreiben Sie eine Klasse `ArrayIterable`, die im Konstruktor einen Array erhält und bei der Iteration über die einzelnen Arrayelemente iteriert.

Java: Iteratoren

```
public static class ArrayIterable<A> implements Iterable<A>{
  A[] as;
  public ArrayIterable(A[] as){
    this.as = as;
  }
}
```

Ein Beispielaufruf:

Shell

```
jshell> new ArrayIterable<>(new String[]{"Hello","World"}).
   ...>   forEach(w->System.out.print(w.toUpperCase()+" "))
HELLO WORLD
```

Aufgabe 4 Schreiben Sie eine Klasse `IterableString`. Sie soll im Konstruktor ein String-Objekt erhalten. Die Klasse soll `Iterable<Character>` so implementieren, dass über die einzelnen Zeichen des Strings iteriert wird.

Java: Iteratoren

```
public static class  IterableString implements Iterable<Character> {
  public IterableString(String str){
  }
  public static void main(String[] args){
    for (char c:new IterableString("Hello world!")){
      System.out.println(c);
    }
  }
}
```

Ein Beispielaufruf:

Shell

```
jshell> new IterableString("Hello").forEach(c->System.out.print(c+" "))
H e l l o
```

Aufgabe 5 Schreiben Sie eine Klasse `Lines`. Sie soll `Iterable<String>` so implementieren, dass nacheinander die Zeilen eines im Konstruktor übergebenen Strings bei der Iteration durchlaufen werden.

Beachten Sie: Strings, die mit einem Zeilenende beginnen, sollen eine erste Zeile aus dem leeren String haben. String, die mit einem Zeilenende enden, sollen eine letzte Zeile mit einem Leerstring haben. Damit hat der String, der aus einem einzigen Zeilenende besteht, zwei Zeilen mit leeren Strings.

Java: Iteratoren

```
public static class Lines implements Iterable<String> {
  static String NEW_LINE = System.getProperty("line.separator");
  public Lines(String str){
    /*ToDo*/
  }
  @Override public Iterator<String> iterator() {
    return     /*ToDo*/;
  }
  public static void main(String[] args) {
    for (String s:new Lines("hallo"+NEW_LINE+"welt!"))
      System.out.println(s);
  }
}
```

Ein Beispielaufruf:

Shell

```
jshell> new Lines("\nHello\nWorld").forEach(l->System.out.println(l))

Hello
World
```

Aufgabe 6 Schreiben Sie eine Klasse `Words`. Sie soll `Iterable<String>` so implementieren, dass nacheinander die Wörter eines im Konstruktor übergebenen Strings bei der Iteration durchlaufen werden. Wörter werden durch Leerzeichen, Zeilenenden und Tabulatorzeichen getrennt. Es soll keine leeren Wörter geben.

Java: Iteratoren

```
public static class Words implements Iterable<String>{
  public Words(String text){
  }
}
```

Ein Beispielaufruf:

Shell

```
jshell> new Words("words don't    come  easy\nto   me    ").
   ...>   forEach(w->System.out.print(w+" "))
words don't come easy to me
```

Aufgabe 7 Schreiben Sie eine generische Klasse `IndexIterable<A>`. Im Konstruktor soll ein Objekt des Typs `java.util.function.Function<Long,A>` übergeben werden. Beim i-ten Aufruf der Methode `next()` soll der erzeugte Iterator diese Funktion benutzen, um das nächste Element für den Index i zu erzeugen. Der erste Aufruf habe den Index 1.

Java: Iteratoren

```
public class IndexIterable<A> implements Iterable<A> {
  public IndexIterable(Function<Long, A> f) {
  }
}
```

Ein Beispielaufruf:

Shell

```
jshell> var sqs = new IndexIterable<>(x->x*x).iterator()
sqs ==> name.panitz.util.Iteratoren$IndexIterable$1@67424e82

jshell> new IntRange(1,20).forEach(i->System.out.print(sqs.next()+" "))
1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400
```

Aufgabe 8 Schreiben Sie eine generische Klasse `GenerationIterable<A>`. Im Konstruktor soll ein Obejekt `f` des Typs `java.util.function.Function<A,A>` und ein Element `a` des Typs `A` übergeben werden. Beim Aufruf der Methode `next()` soll der erzeugte Iterator die folgende Folge generieren: $a, f(a), f(f(a)), f(f(f(a))), \ldots$.

Java: Iteratoren

```
public static class GenerationIterable<A> implements Iterable<A> {
  A a;
  Function<A,A> f;
  public GenerationIterable(A a, Function<A,A> f){
    this.a = a;
    this.f = f;
  }
}
```

Ein Beispielaufruf:

Shell

```
jshell> var xs = new GenerationIterable<>(5,x->-x).iterator()
xs ==> name.panitz.util.Iteratoren$GenerationIterable$MyIt@246b179d

jshell> new IntRange(1,20).forEach(i->System.out.print(xs.next()+" "))
5 -5 5 -5 5 -5 5 -5 5 -5 5 -5 5 -5 5 -5 5 -5 5 -5
```

Aufgabe 9 Benutzen Sie die Klasse `GenerationIterable` um einen Iterator zu erzeugen, der über alle ungeraden Zahlen iteriert. Machen Sie hierzu den geeigneten Aufruf von `super` im Konstruktor.

Java: Iteratoren

```
public static class OddIterable extends GenerationIterable<Long>{
  public OddIterable() {
    //hier der korrekte Aufruf....
  }
}
```

Ein Beispielaufruf:

Shell

```
jshell> var xs = new OddIterable().iterator()
xs ==> name.panitz.util.Iteratoren$GenerationIterable$MyIt@b81eda8

jshell> new IntRange(1,20).forEach(i->System.out.print(xs.next()+" "))
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
```

Aufgabe 10 Schreiben Sie eine Klasse, die aus einem Iterable-Objekt und einer Zahl n ein neues Iterable-Objekt erzeugt, das nach n Iterationsschritten kein neues nächstes Element mehr liefert. Es limitiert das Iterable auf maximal n-Objekte.

Java: Iteratoren

```
public static record Limit<A>(Iterable<A> itA, long n)
                                        implements Iterable<A>{
  public Iterator<A> iterator(){
    return new Iterator<>(){
      Iterator<A> it = itA.iterator();
      int i = 0;
       /* ToDo */
    };
  }
}
```

Jetzt lassen sich potentiell unendlich lange Iterables auf endlich viele Elemente limitieren. Zum Beispiel die zuvor implementierten ungeraden Zahlen.

Ein Beispielaufruf:

Shell

```
jshell> new Limit<>(new
↪  OddIterable(),10).forEach(x->System.out.print(x+" "))
1 3 5 7 9 11 13 15 17 19
```

Aufgabe 11 Schreiben Sie eine Klasse, die aus einem Iterable-Objekt ein neues Iterable-Objekt erzeugt. Beim Iterieren soll das ursprüngliche Iterable-Objekt verwendet werden und jeweils auf das durch `next` erhaltene Objekt eine Funktion angewendet werden.

Java: Iteratoren

```
public static record Maperable<A,R>(Iterable<A> itA, Function<A,R> f)
                                                    implements Iterable<R>{
  public Iterator<R> iterator(){
    return new Iterator<>(){
      Iterator<A> it = itA.iterator();
      /* ToDo */
    };
  }
}
```

Ein Beispielaufruf:

Shell

```
jshell> new Maperable<>(new
↪  Words("words don't come easy"),x->x.length()).
   ...>   forEach(x->System.out.print(x+" "))

5 5 4 4
```

Aufgabe 12 Schreiben Sie eine Klasse, die aus einem Iterable-Objekt ein neues Iterable-Objekt erzeugt. Beim Iterieren soll das ursprüngliche Iterable-Objekt verwendet werden und jeweils auf das durch `next` erhaltene Objekt eine Funktion angewendet werden.

Die eigentliche Iteratorklasse braucht hierzu ein Feld, in dem es den nächsten Wert, der das Prädikat erfüllt, speichert. Ansonsten ist es nihct möglich, in `hasNext()` zu sagen, ob es noch einen weiteren Wert gibt.

Java: Iteratoren

```
public static record Filterable<A>(Iterable<A> itA, Predicate<A> p)
                                          implements Iterable<A>{
  public Iterator<A> iterator(){
    return new MyIterator();
  }
  private class MyIterator implements Iterator<A>{
    Iterator<A> it = itA.iterator();
    A theNext = null;
    MyIterator(){getTheNext();}
    void getTheNext(){
         /* ToDo store the element,
            for which the predicate p holds in theNext */
    }
    public boolean hasNext(){return theNext!=null;}
    public A next(){
      var result = theNext;
      getTheNext();
      return result;}
  }
}
```

Ein Beispielaufruf:

Shell

```
jshell> new Filterable<>(new IntRange(1,100),x->x%9==0).
   ...> forEach(x-> System.out.print(x+" "))
9 18 27 36 45 54 63 72 81 90 99
```

Java: Iteratoren

```
}
```