title: "Iterator und Fabrik" permalink: /05-iterator/ mathjax: true

Iteration

In der Mathematik und Informatik bedeutet *iterieren* eine bestimmte Handlung zu wiederholen.

In der (mathematischen) Optimierung versteht man darunter die wiederholte Anwendung einer Rechenvorschrift um zum Optimum zu gelangen, ähnlich zu einem Golfspieler, welcher für eine Spielbahn mehrere, immer kürzere Schläge braucht.

In der Informatik ist meist die Iteration im algorithmischen Sinne gemeint: das wiederholte Ausführen von Anweisungen. Im einfachsten Sinne sind das die `for`, `for-each` und `while` Schleifen, welche beliebige Anwendungen wiederholen:

```
for (int i = 0; i < 3; i++) {
        if (i < 2)
                System.out.print((i+1) + ", ");
        else
                System.out.print(" oder " + (i+1));
}
System.out.println(" - letzte Chance - vorbei!");
```

Was wird hier wohl ausgegeben? (Lösung)

Wir erinnern uns auch, dass jede `for` Schleife als `while` Schleife formuliert werden kann, und umgekehrt:

```
int i = 0;
while (i++ < 2)
        System.out.println(i + ", ");
System.out.println(" oder " + i);

System.out.println(" - letzte Chance - vorbei!");
```

Iteration für einfache Datenstrukturen

Heute wollen wir uns mit einer speziellen Anwendung der Iteration befassen: das Traversieren, also Durchlaufen, von Datenstrukturen. Was ist damit gemeint?

Aus dem letzten Semester kennen wir bereits Felder (Arrays):

```
int[] a = {4, 1, 2, 7};

// alle Elemente des Arrays ausgeben
```

```
for (int i = 0; i < a.length; i++)
        System.out.println(a[i]);
```

Zu Anfang dieses Semesters hatten wir die Liste als sequenzielle Datenstruktur kennengelernt, welche wir dann einmal mit einem Array (`ArrayList`, Quelle) und einmal mit verketteter Liste (`LinkedList`, Quelle) implementiert hatten. Beide Klassen implementieren dabei die selbe Schnittstelle:

```
interface List<T> {
        void add(T o);
        T get(int i);
        int size();
}
```

Wollten wir nun alle Elemente der Liste besuchen, gingen wir analog zum Array vor:

```
List<String> li = ...;
for (int i = 0; i < li.size(); i++)
        System.out.println(li.get(i));
```

Wir erinnern uns dabei aber, dass `ArrayList.get` sehr viel effizienter als `LinkedList.get` implementiert ist, nämlich in $O(1)$ statt $O(n)$:

```
public T get(int i) {
        if (i >= length)
                throw new ArrayIndexOutOfBoundsException();
        // direkt den Wert im Array zurückgeben
        return a[i];
}
```

Gegenüber der verketteten Liste:

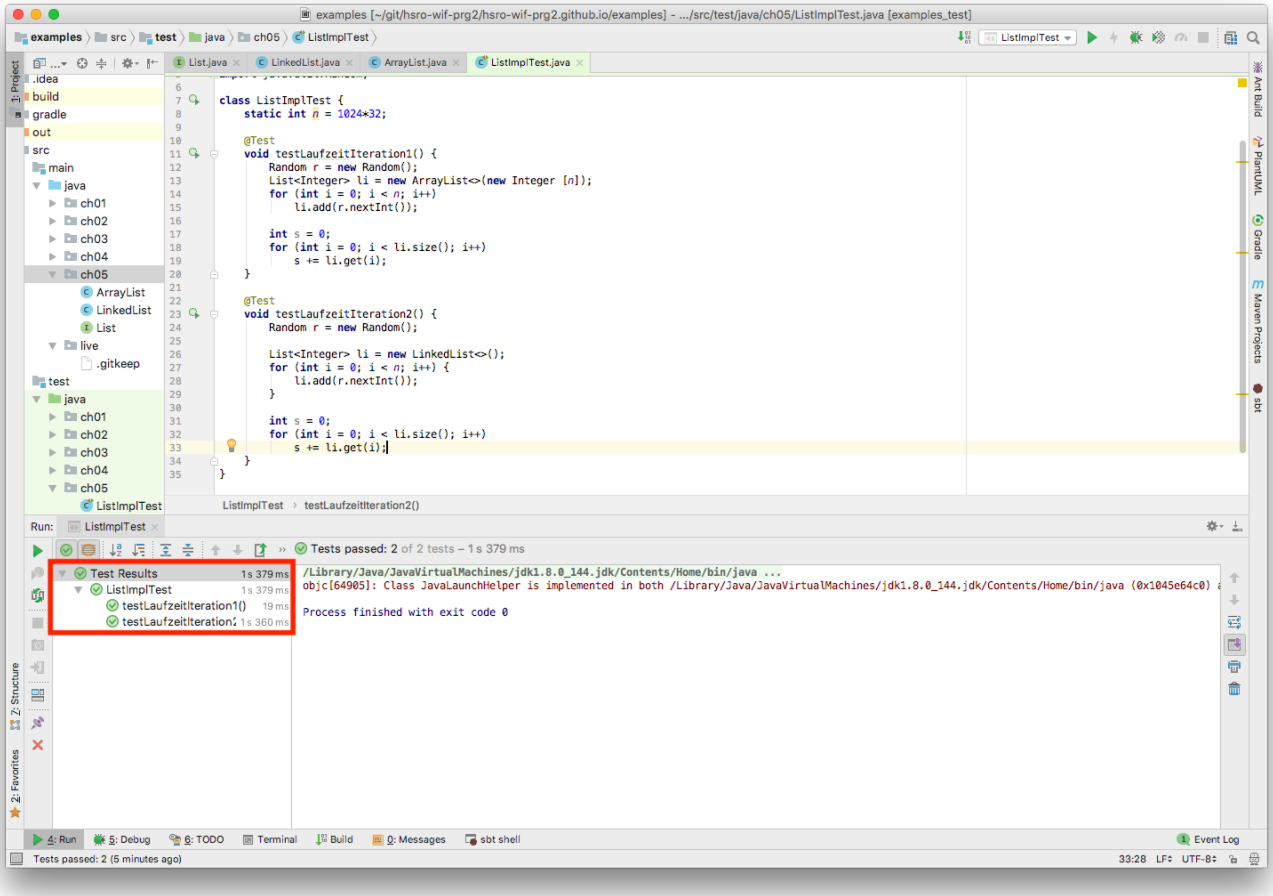
```
public T get(int i) {
        Element it = head;
        // "vorspulen" bis zum Zielelement
        while (i-- > 0)
                it = it.next;
        return it.value;
}
```

Ein kleiner Testcase um die Laufzeit zu vergleichen ergibt folgendes Bild:

```java
int n = 1024*32;  // wie viele Elemente?
Random r = new Random();
List<Integer> li = new ArrayList<>(new Integer [n]);
// alternativ: ... = new LinkedList<>();

// mit Zufallszahlen initialisieren
for (int i = 0; i < n; i++)
        li.add(r.nextInt());

// aufsummieren...
int s = 0;
for (int i = 0; i < li.length(); i++)
        s += li.get(i);
```



{: .center}

Die `ArrayList` ist hier mit 19ms um zwei Größenordnungen **schneller** als die `LinkedList` mit 1369ms -- logisch: ist ja `get` bei ersterer mit $O(1)$ nach *n* Aufrufen im ganzen $O(n)$, so ist letztere $O(n^2)$ da für jeden `get` Aufruf von vorne bis zur Zielposition gelaufen werden muss.

Nun ist die Iteration über eine Datenstruktur eine sehr häufige Geschichte. Wie könnte man diese wohl für eine verkettete Liste wie im obigen Beispiel beschleunigen?

Iterator

Wir stellen fest: Im Kern liegt das Problem darin, dass bei der Iteration auf der verketteten Liste immer wieder von vorne begonnen werden muss. Um das zu vermeiden, müsste man sich also merken, an welcher Position (`it`) man steht. Kennt man die innere Struktur der Liste, und ist der Zugriff von aussen möglich, so könnte man die Iteration "per Hand" machen, also in etwa so:

```
LinkedList<Integer> li = new LinkedList<>();

LinkedList<Integer>.Element it = li.head;
while (it != null) {
        System.out.println(it.value);
        it = it.next;
}
```

Nun ist aber sowohl die innere Klasse als auch das `head` -Element i.d.R. `private`. Und weiterhin sollte man ja immer nur gegen Schnittstellen, also unabhängig von der tatsächlichen Realisierung programmieren -- und das Interface `List` kennt eben gerade nur die Operationen auf einer Liste, aber nicht die Struktur.

Wir suchen also eine Abstraktion für die Iteration. Allgemein betrachtet brauchen wir zur Traversierung einer Datenstruktur ein Hilfsobjekt, das uns angibt, was das nächste Element ist, und ob es noch ein weiteres gibt. Dieser `Iterator` ist als Schnittstelle wie folgt festgelegt:

```
interface Iterator<T> {
        boolean hasNext();
        T next();
}
```

> Hinweis: Der in der Java API definierte `java.util.Iterator` (Quelle) verfügt noch über weitere Methoden, auf die wir hier aber nicht eingehen wollen.

Angenommen wir hätten also so ein `Iterator` -Objekt, so könnten wir wie folgt iterieren:

```
Iterator<Integer> it = ???;  // dazu später...

// solange es noch ein weiteres Element gibt
while (it.hasNext()) {
        // nächstes Element holen, weitergehen
        int val = it.next();  // it "merkt" sich aktuelle Position
        System.out.println(val);
}
```

Vielleicht nochmal zur Verdeutlichung gegenübergestellt, wie es ohne Iterator ging:

```
int i = 0;
while (i < li.length()) {
        int val = li.get(i); // autsch: Hier wird immer wieder von vorne begonnen
```

```
        i++;
        System.out.println(val);
}
```

Der Iterator übernimmt also die zwei zentralen Aufgaben der Iteration:

1. Er liefert das nächste Element, und
2. er merkt sich die aktuelle Position.

In der Regel braucht der Iterator Kenntnis der (und Zugriff auf) die innere Struktur und ist daher als (anonyme) innere Klasse realisiert.

Iterator für ArrayList

Wie sieht nun ein Iterator für die `ArrayList` aus?

- Die aktuelle Position ist ein Index, also z.B. ein `int`.
- Es gibt noch ein weiteres Element, wenn die aktuelle Position um mindestens eins kleiner ist als die Listenlänge.
- Zum nächsten Element gelangt man, indem man die aktuelle Position um eins erhöht.

```
class ArrayList<T> implements List<T> {
        // ...
        class MyIterator implements Iterator<T> {
                int pos = 0;
                public boolean hasNext() {
                        return pos < length;
                }
                public T next() {
                        if (!hasNext())
                                throw new NoSuchElementException();
                        T h = a[pos];
                        pos = pos + 1;
                        return h;
                }
        }

        // einen neuen Iterator erstellen
        public Iterator<T> iterator() {
                return new MyIterator();
        }
}
```

Wir stellen dabei Folgendes fest:

- Die innere Klasse `MyIterator` kann auf das `length` Attribut der `ArrayList` zugreifen; das macht den `hasNext`-Check einfach.
- In `next` wird zunächst das aktuelle Element in `h` gespeichert, dann die Position um eins erhöht, und abschließend `h` zurück gegeben.

Der Iterator kann nun wie folgt verwendet werden:

```
ArrayList<Integer> li = new ArrayList<>();

Iterator<Integer> it = li.iterator();
while (it.hasNext())
        System.out.println(it.next());
```

Iterator für LinkedList

Für die verkettete Liste sind nur kleine Änderungen notwendig:

1. Die aktuelle Position ist immer eine Referenz auf ein `Element`; zu Beginn ist diese `head`.
2. Es gibt nun kein weiteres Element, wenn die aktuelle Position `null` (sprich: "man hinten hinausgefallen") ist.

An diesem Beispiel können wir auch noch die *anonyme* innere Klasse wiederholen:

```
class LinkedList<T> implements List<T> {
        // ...
        public Iterator<T> iterator() {
                return new Iterator<T>() {
                        Element it = head;
                        public boolean hasNext() {
                                return it == null;
                        }
                        public T next() {
                                if (!hasNext())
                                        throw new NoSuchElementException();
                                T h = it.value;
                                it = it.next;
                                return h;
                        }
                };
        }
}
```

Es sei an dieser Stelle nochmal auf die Gemeinsamkeiten den beiden Implementierungen hingewiesen, zu sehen in einer Gegenüberstellung:

```
public T next() {
        if (!hasNext())
                throw new NoSuchElementException();

        /* verkettete Liste -- ArrayList     */
        T h = it.value;        T h = a[pos];   // aktuelles Element merken
        it = it.next;          pos = pos + 1;  // "eins weiter gehen"
```

```
        return h;
}
```

Iterable

Am obigen Beispiel haben wir gesehen, dass wir sowohl für die `ArrayList` als auch die `LinkedList` *unterschiedliche* Iteratoren implementieren können, in beiden Fällen hatten wir die Methode dazu `iterator` genannt. Da die Iteration natürlich wiederum ein abstraktes Konzept ist, gibt es in der Java API hierzu das Interface `java.lang.Iterable<T>`, welches eben diese Methode vorschreibt:

```
interface Iterable<T> {
        Iterator<T> iterator();
}
```

Implementiert eine Klasse also (unter Anderem) obige Schnittstelle, so kann man für sie entsprechend einen Iterator erhalten. Tatsächlich ist es auch in unserem Beispiel so, dass jede Liste einen Iterator liefern können sollte. Um dieses syntaktisch ausdrücken zu können gibt es nun das Schlüsselwort `extends`. Übertragen auf unser Beispiel bedeutet das: die Schnittstelle `List` *erweitert* (engl. extends) die Schnittstelle `Iterable`, bzw. in Java:

```
interface List<T> extends Iterable<T> {
        // implizit von Iterable: Iterator<T> iterator();
        void add(T o);
        T get(int i);
        int length();
}
```

Ein Interface kann dabei (im Prinzip) beliebig viele andere Interfaces erweitern (in Java nach `extends`, durch Kommata getrennt). Implementiert eine *Klasse* nun dieses erweiterte Interface, so muss sie entsprechend alle Methoden, also "die Summe der Interfaces" implementieren.

> Hinweis: In der Java API ist es genau genommen so, dass `List` das Interface `Collection` erweitert, welches wiederum `Iterable` erweitert.

Iterable und for-each

Seit Java 5 gibt es nun eine weitere syntaktische Besonderheit: Erfüllt eine Referenz das `Iterable` Interface, so kann es in der `for-each` Schleife verwendet werden.

```
List<Integer> li = new ArrayList<>();
// ... oder: = new LinkedList<>();

for (int i : li)
        System.out.println(i);
```

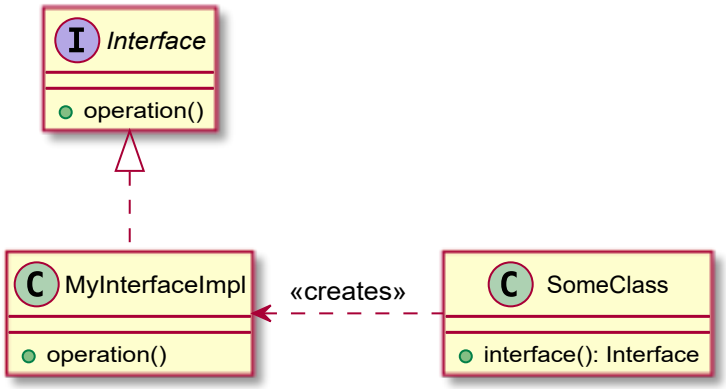
Diese Schreibweise ist oft kürzer und kompakter als eine `while` Schleife mit entsprechendem `Iterator`, allerdings steht das Iteratorobjekt auch nicht zur Verfügung, da es implizit verwaltet wird.

Design Patterns

In diesem Kapitel haben wir zwei *Entwurfsmuster* kennen gelernt. Diese *design patterns* sind Muster, welche in der Softwareentwicklung immer wieder vorkommen, und spätestens seit 1994 in dem Buch Design Patterns einen standardisierten Namenskatalog von 23 Mustern angehören.

Factory Method

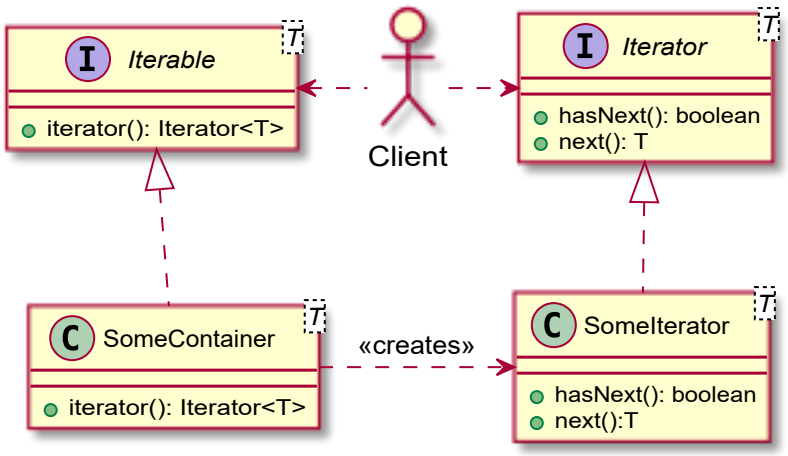
Das obige Muster, in dem eine Methode eine Instanz zu einem Interface zurück gibt, heisst auch *factory method* bzw. Fabrikmethode. Entscheidend für dieses (wie auch dem nah verwandten *factory*) Muster, dass dem Aufrufer der Methode nur das *Interface* als Rückgabetyp bekannt ist, nicht aber die der Instanz zu Grunde liegende *implementierende Klasse*. Diese ist in der Regel als innere oder anonyme innere Klasse realisiert.



Das Factory Method Muster ist dabei ein Erstellungsmuster (*creational pattern*), da es die *Erstellung* von Objekten betrifft.

Iterator

Der Iterator ist ein Verhaltensmuster (*behavioral pattern*), da es um die Modellierung des sequenziellen Zugriffs auf eine Containerstruktur geht. Es setzt die beiden Schnittstellen `Iterable` und `Iterator` in Beziehung, die wiederum von konkreten Klassen implementiert werden.



Der Iterator verwendet dabei das Factory Method Pattern.

Iteration für Sets

Die Iteration ist also ein abstraktes Konzept für sequenziellen Zugriff auf Datenstrukturen. Da auch das *Set* eine Datenstruktur ist, sollte man auch für dieses alle Elemente aufzählen können; gewünscht ist also eine Erweiterung des Interfaces `Set`:

```
interface Set<T> extends Iterable<T> {
        void add(T o);
        boolean contains(T o);
        int size();
}
```

Die Schwierigkeit ist nun: Ist das Set als Binärbaum implementiert, gibt es im Gegensatz zur Liste ja bis zu **zwei** Nachfolger pro "Besuchspunkt":

```
class SetImpl<T extends Comparable<T>> implements Set<T> {
        // ...
        private class Element {
                T value;
                Element left, right;
                // ...
        }

        Element root;
}
```

Wie kann man sich also die *Position* merken an der man steht, aber dazu auch, wo man *als nächstes* hingeht? Diese Frage lässt sich ja nun nicht (einfach) eindeutig beantworten.

Agenda

Man könnte die Frage aber auch *anders formulieren:* Welche Elemente *muss ich noch* besuchen? Anstatt genau eines (aktuellen) Elements merkt man sich also eine *Liste* von Elementen, welche noch zu besuchen sind -- die sogenannte *Agenda:*

- Zu Beginn legt man (sofern vorhanden) das `root` Element in die Agenda.
- Es gibt so lange noch ein nächstes Element, solange die Agenda nicht leer ist.
- Nimmt man ein Element aus der Agenda heraus, so prüft man, ob es einen linken und/oder rechten Nachfolger gibt, und legt diesen ggf. in die Agenda.

```
class SetImpl<T extends Comparable<T>> implements Set<T> {
        // ...
        public Iterator<T> iterator() {
                return new Iterator<T>() {
                        List<Element> agenda = new LinkedList<>();

                        // Defaultkonstruktor für anonyme innere Klassen
                        {
                                if (root != null)
                                        agenda.add(root);
                        }

                        public boolean hasNext() {
                                return agenda.length() > 0;
                        }

                        @Override
                        public T next() {
                                Element e = agenda.remove(0);
                                if (e.left != null)
                                        agenda.add(e.left);
                                if (e.right != null)
                                        agenda.add(e.right);
                                return e.value;
                        }
                };
        }
}
```

Wie verhält sich nun diese Agenda während der Lebenszeit des Iterators? Nehmen wir folgendes Beispiel, bei dem die Zahlen 4, 2, 6, 3, 1, 5 in einen Binärbaum eingefügt wurden.

```
root --> 4
        /  \
       2    6
      / \  /
     1  3  5
```

Dann verhält sich die Agenda (`[]`) bei den `.iterator()` bzw. `.next()` Aufrufen wie folgt (siehe Testcase `testSetIterable`) :

```
.iterator()
        .add(root) --> [4]     // initiale Agenda
.next()
        .remove(0) --> 4, []
        .add(2)    --> [2]
        .add(6)    --> [2, 6]
        return 4
.next()
        .remove(0) --> 2, [6]
        .add(1)    --> [6, 1]
        .add(3)    --> [6, 1, 3]
        return 2
.next()
        .remove(0) --> 6, [1, 3]
        .add(5)    --> [1, 3, 5]
        return 6
.next()
        .remove(0) --> 1, [3, 5]
        // weder .left noch .right
        return 1
.next()
        .remove(0) --> 3, [5]
        return 3
.next()
        .remove(0) --> 5, []
        return 5
.hasNext() ==> false!
```

Übersicht: Klassensyntax

Wir haben in den letzten Kapiteln nun verschiedene *Arten* von Klassen in Java kennen gelernt:

- (normale) Klasse: Gewöhnliche Implementierung, verwende `implements` zur Interfaceerfüllung
- *innere* Klasse: Modellierung von Klassen, welche Teilfunktionalität für eine einschließende Klasse erbringen, z.B. `Element` bei verketteter Liste oder Binärbaum.
- *statische innere* Klasse: Wie *innere*, aber wenn Objekte unabhängig der äußeren Klasse existieren können.
- *anonyme innere* Klasse: Eine Kurzschreibweise um Instanzen zu Interfaces zu erzeugen; häufig verwendet um `Iterator`- oder `Comparator`-Instanzen zu erstellen.

(Normale) Klassen

```
class MeineKlasse {
        // Attribute
}
```

- **Eine** normale, nicht-statische Klasse pro `.java` Datei
- Klassenname muss gleich dem Dateinamen sein; Übersetzung von `MeineKlasse.java` in `MeineKlasse.class` (Bytecode).
- Kann beliebig viele innere Klassen enthalten

Innere Klassen

```
class MeineKlasse {
        private MeineInnere {

        }

        private static MeineStatischeInnere {

        }
}
```

- Beliebig viele innere Klassen
- Sichtbarkeit und Gültigkeit analog zu Attributen
- **Innerhalb** einer normalen Klasse, **ausserhalb** von Methoden
- `Innere` kann **nur** in Instanz von `Aeussere` existieren
- Im Prinzip beliebig schachtelbar (innere in inneren in inneren, ...)

(Nicht-statische) Innere Klasse

```
class Aeussere {
        private String attribut = "A";
        private static String ATTRIBUT = "B";

        class Innere {
                String attribut = "X";

                // Compilerfehler! Innere können keine static haben
                static String ATTRIBUT = "Y";

                void zugriff() {
                        System.out.println(this.attribut);  // "X"
                        System.out.println(Aeussere.this.attribut);  // "A"
                }
        }
}
```

- Zugriff auf alle Attribute der `Aeussere` Instanz
- Bei Namenskonflikten mit `<KlassenName>.this.*` disambiguieren
- keine `static` Attribute in inneren Klassen

Statische Innere Klasse

```
class Aeussere {
        Strint attribut = "A";
        static class Statische {
                String attribut = "Z";
                void zugriff() {
                        System.out.println(attribut);  // "Z"

                        // Compilerfehler! Statische Innere hat keinen Aeussere-
Scope
                        System.out.println(Aeussere.this.attribut);
                }
        }
}
```

- Können ohne Instanz der äußeren Klasse verwendet werden

```
Aeussere.Statische inst = new Aeussere.Statische();
```

- Folglich kein direkter Zugriff auf die Attribute der äußeren Klasse

Anonyme Innere Klasse

```
interface Intf {
        void methode();
}
```

```
final int aussen = 10;
Intf inst = new Intf() {
        int attr;
        {
                // Konstruktor, wenn nötig
                attr = aussen;  // Zugriff nur auf quasi-final!
        }
        public void methode() {
                System.out.println(attr);
        }
};
```

- Erstellt ein Objekt das ein Interface implementiert
- Klassendefinition aber zur Laufzeit unbekannt ("anonym")
- Zugriff auf äußere Attribute nur, wenn diese quasi-final sind.

Zusammenfassung

- Die **Iteration** für Containerstrukturen (wie z.B. Listen oder Sets) ist eine Abstraktion, welche dem Benutzer sequenziellen Zugriff auf die enthaltenen Elemente gibt, ohne die innere Struktur zu kennen.
- Der **Iterator** als **Verhaltensmuster** (behavioral pattern) beschreibt dabei den Zusammenhang der Interfaces `Iterator<T>` und `Iterable<T>`.
- Die **Fabrikmethode** (factory method) ist ein **Erstellungsmuster** (creation pattern) welches sich auch im Iterator Muster wiederfindet.
- **Iteratoren für sequenzielle Datenstrukturen** sind im Allgemeinen einfach zu implementieren: sie merken sich die aktuelle Position.
- **Iteratoren für Baumstrukturen**, also Datenstrukturen deren Elemente mehr als einen Nachfolger haben, verwenden hingegen eine **Agenda:** eine Liste von noch zu besuchenden Elementen.

■