



Objektorientierte Programmierung

Kapitel 4 – Iteratoren

Prof. Dr. Kai Höfig

Arten von Klassen

- In Java sind folgende Klassen möglich:
 - (normale) Klasse
 - 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
- ```
public class MeineKlasse {
}
```
- innere Klasse
  - statische innere Klasse
  - anonyme innere Klasse

# Innere Klassen



- Beliebig viele innere Klassen
- Sichtbarkeit und Gültigkeit analog zu Attributen
- Innerhalb einer normalen Klasse, außerhalb von Methoden
- Innere kann nur in Instanz von Äußere existieren
- Im Prinzip beliebig schachtelbar (innere in inneren in inneren, ...)
- Zugriff auf alle Attribute der Äußere Instanz
- Bei Namenskonflikten mit `<KlassenName>.this.*` disambiguieren
- keine `static` Attribute in inneren Klassen

```
public class MeineKlasse {
 private int attr;
 private static int ATTR=0;
 private class MeineInnereKlasse{
 protected int attr;
 // protected static int ATTR=1; // Compilerfehler!
 void innerMethod(){
 attr=2; // das Attribut von MeineInnereKlasse
 MeineKlasse.this.attr=3; // das Attribut von MeineKlasse
 }
 }
}
```

# Statische Innere Klassen

- Können ohne direkte Instanz der äußeren Klasse verwendet werden, müssen aber mit new instanziiert werden
- Folglich kein direkter Zugriff auf die nicht statischen Attribute der äußeren Klasse
- Zugriff auf die statischen Attribute der äußeren Klasse

```
public class startupKlassen {
 public static void main(String[] args) {
 MeineKlasse.MeineStatischeInnereKlasse sik = new MeineKlasse.MeineStatischeInnereKlasse();
 // MeineKlasse.MeineInnereKlasse ik = new MeineKlasse.MeineInnereKlasse(); // Compilerfehler
 System.out.println(sik.method());
 System.out.println(sik.s);
 //System.out.println(MeineKlasse.MeineInnereKlasse.method()); // Compilerfehler!
 }
}
```

```
public class MeineKlasse {
 private int attr;
 private static int ATTR=0;
 ...
 static class MeineStatischeInnereKlasse{
 String s = "hallo";
 int method(){
 // MeineKlasse.this.attr=0; // Compilerfehler!
 ATTR = 10;
 return ATTR;
 }
 }
}
```

# Anonyme Innere Klasse



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

```
public class startupKlassen {
 public static void main(String[] args) {
 ..
 int y=1; // oder final int y=1;
 Intf i = new Intf(){
 int x = 0;
 {
 this.x=y;
 }
 public void method(){
 System.out.println("Hey! "+x+y);
 }
 };
 // y=3; // Compilerfehler!
 }
}
```

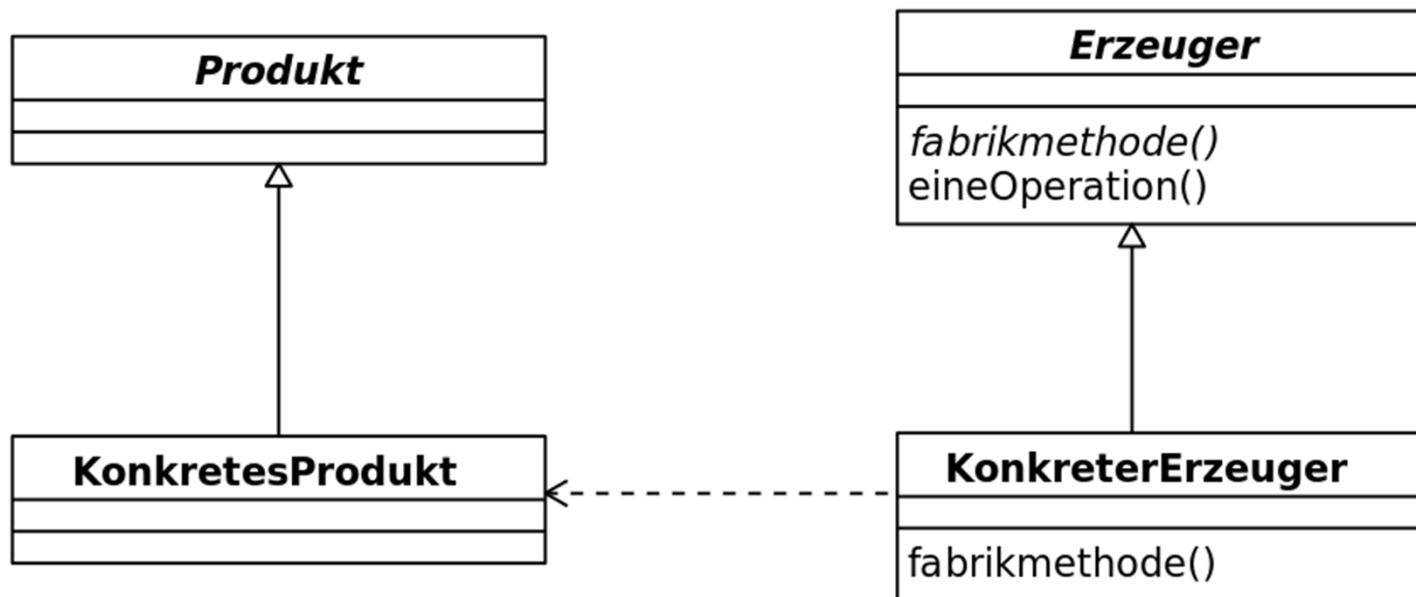
```
interface Intf {
 void method();
}
```

# Fabrikmethode

## *Factory Method*



- Die Fabrikmethode (engl. Factory method) ist ein s.g. Erstellungsmuster
- Eine Fabrikmethode liefert Elemente zu einem Interface zurück
- Dieses Design-Muster bietet die Grundlage für *Iteratoren*



Für weitere grundlegende Design Muster siehe hier: <https://springframework.guru/gang-of-four-design-patterns/>

# Beispiel Fabrikmethode



```
public interface Auto {
 public void hupen();
}
```

```
public class Lastwagen implements Auto{
 @Override
 public void hupen() {
 System.out.println("Tuuuuuuut");
 }
}
```

```
public class Kleinwagen implements Auto{
 @Override
 public void hupen() {
 System.out.println("Tröööt");
 }
}
```

- Was bringt's? In der Main kann ich wieder nur die Interfaces benutzen, der Code wird schnell austauschbar und erweiterbar auf weitere Arten von Autos und Herstellern.

```
public interface Autofabrik {
 public Auto produziere();
}
```

```
public class Lastwagenfabrik implements Autofabrik{
 @Override
 public Auto produziere() {
 return new Lastwagen();
 }
}
```

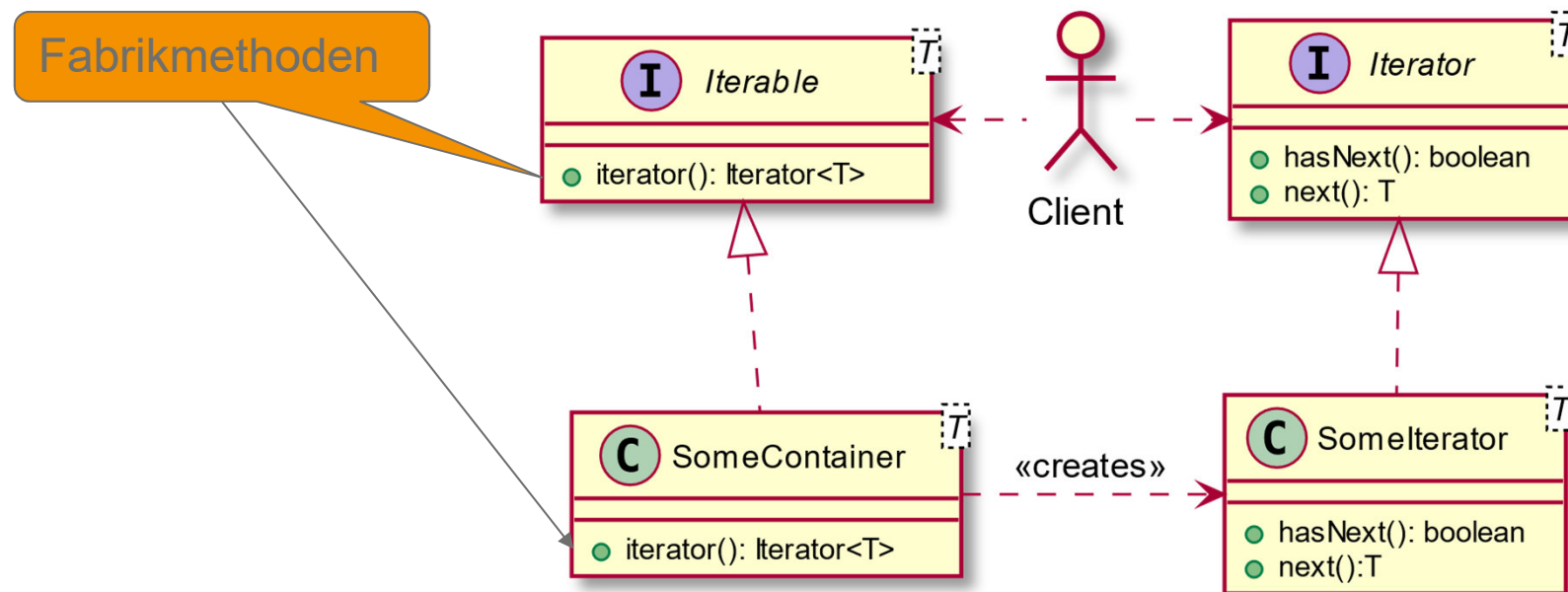
```
public class Kleinagenfabrik implements Autofabrik{
 @Override
 public Auto produziere() {
 return new Kleinwagen();
 }
}
```

Fabrikmethode

```
public class startupAutogewerbe {
 public static void main(String[] args) {
 Lastwagenfabrik lwf = new Lastwagenfabrik();
 Kleinagenfabrik kwf = new Kleinagenfabrik();
 Auto[] autos = {lwf.produziere(), kwf.produziere()};
 for (Auto a : autos){
 a.hupen();
 }
 }
}
```

# Iterator

- Ein `Iterator` wird von der Fabrikmethode eines `Iterables` geliefert.
- Modellierung des sequenziellen Zugriffs auf eine Containerstruktur
- Setzt `Iterable` und `Iterator` in Beziehung
- Verhaltensmuster (behavioral pattern)
- Verwendet dabei das Factory Method Pattern





# Verwendung von Iteratoren

- Regulär, mit `while` Schleife\*

```
IntSet is = new IntSet();
Iterator<Integer> it = is.iterator();
while(it.hasNext()){
 System.out.println(it.next());
}
```

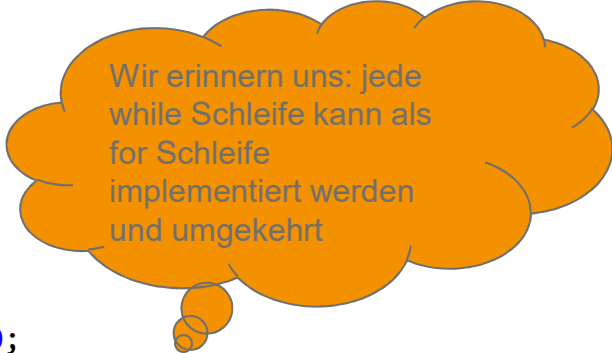
- Mit `for-each` Schleife\*

```
for(int i : is){
 System.out.println(i);
}
```

\*class IntSet implements Iterable<Integer>  
von java.util.Iterator und mit java.lang.Iterable

# 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:



Wir erinnern uns: jede while Schleife kann als for Schleife implementiert werden und umgekehrt

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

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

```
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!");
```

# Iteration über 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):

```
// Iteration über Array
int[] a = {4, 1, 2, 7};

// alle Elemente des Arrays ausgeben
for (i = 0; i < a.length; i++)
 System.out.println(a[i]);
```

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

```
// alle Elemente einer Liste
List<String> li = new ListImpl<String>();
li.add("str1");
li.add("str2");

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

Nachteil: die Methoden zur Iteration müssen wir jedes mal neu implementieren und sie sind mitunter nicht besonders effizient

# Vergleich Iteration über eine verkettete Liste und ein Array

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

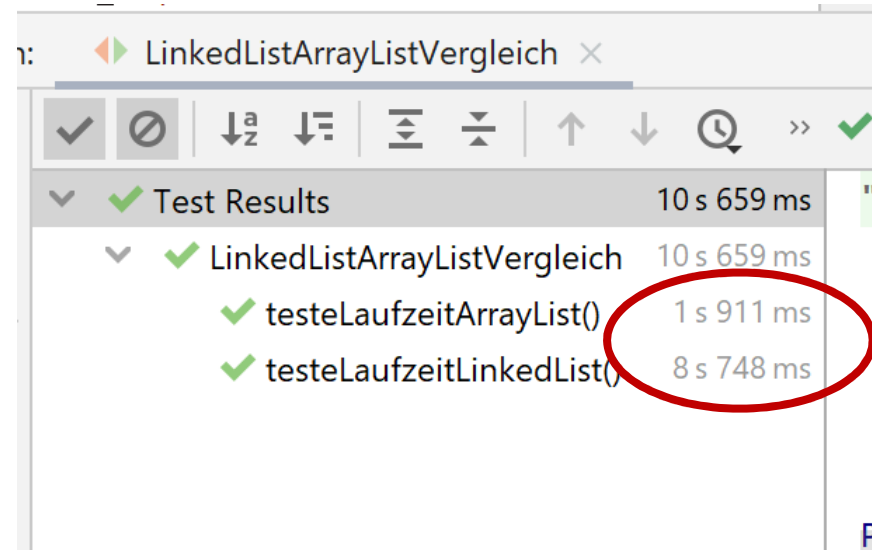
## Verkettete Liste

```
public T get(int i) { // !
 if (head == null)
 throw new NoSuchElementException();

 Element it = head;
 while (i-- > 0)
 it = it.next;
 return it.value;
}
```

## Array Liste

```
public T get(int i) {
 return zs[i];
}
```



| Test Results                 | 10 s 659 ms |
|------------------------------|-------------|
| LinkedListArrayListVergleich | 10 s 659 ms |
| testeLaufzeitArrayList()     | 1 s 911 ms  |
| testeLaufzeitLinkedList()    | 8 s 748 ms  |

Siehe Klasse

`LinkedListArrayListVergleich`  
im Repository

# Was könnte man also machen, um das zu verbessern?



```
public class ListImpl<T> implements List<T> {
 ...
 private Element head;
 Element next; // hier merken wir uns, was als nächstes dran ist für den selbst gebauten Iterator
 ...
 // wir geben das nächste Element aus
 public T getNext(){
 Element e= next;
 if(next.next==null){
 next=null;
 }else{
 next=next.next;
 }
 return e.value;
 }

 // damit wir eine while-schleife verwenden können
 public boolean hasNext(){
 if (next==null)
 return false;
 else
 return true;
 }

 // falls wir nochmal durchlaufen wollen
 public void resetIterator(){
 this.next=head;
 }
 ...
}
```

Wir merken uns das Element, das wir ausgegeben haben, das ginge auch von außen, aber idR. Sind diese Klassen ja private

|                                   |             |
|-----------------------------------|-------------|
| ▼ ✓ Test Results                  | 15 s 470 ms |
| ▼ ✓ LinkedListArrayListVergleich  | 15 s 470 ms |
| ✓ testeLaufzeitLinkedListBesser() | 2 s 191 ms  |
| ✓ testeLaufzeitArrayList()        | 1 s 641 ms  |
| ✓ testeLaufzeitLinkedList()       | 11 s 638 ms |

**Nachteil: das ist nicht standardisiert :/**

# Besser: mit Interfaces arbeiten

- Wir brauchen also ein Ding, was immer gleich funktioniert

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

Und dieses Ding soll uns jede Datenstruktur liefern

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

# Beispiel für unsere ListImplArray<T>



```
public class ListImplArray<T> implements List<T>, Iterable<T>{

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

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

 private T[] zs;

 ...
}
```

```
// Jetzt verwenden wir unser Iterator und Iterable Interface
Iterator<Integer> litit = lit.iterator();
while(litit.hasNext())
 System.out.println(litit.next().toString());
```

- Vorteile:
  - standardisiertes Interface für alle unsere Datenstrukturen; iterieren ist immer gleich.
  - Immer die performanteste Lösung für jede Datenstruktur
  - Einfache Verwendung
- Ein „reset“ brauchen wir nicht, da wir bei jeder Iteration ein neues Iterator Objekt erzeugen

# Beispiel für unsere ListImpl<T>

```
public class ListImpl<T> implements List<T>, Iterable<T> {
 ...
 private Element head;
 ...
 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;
 }
 };
 }
}
```

- Noch Kompakter durch die Verwendung einer Anonymen Klasse, gleiche Vorteile, gleiche Verwendung

```
// Jetzt verwenden wir unser Iterator und Iterable Interface
Iterator<Integer> litit = lit.iterator();
while(litit.hasNext())
 System.out.println(litit.next().toString());
```



# Java.util.Iterator und java.lang.Iterable

- Java bietet diese Interfaces ebenfalls an:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html>

- Diese sind in der Verwendung genauso zu handhaben (wie zu Beginn der Vorlesung skizziert) und sind für viele Datenstrukturen des Java Frameworks implementiert, z.B. das Java Collection Framework

<https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html>

# 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 erinnern 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.
- Es gibt normale, innere, statische innere und anonyme innere Klassen