Technische Hochschule **Rosenheim**
Technical University of Applied Sciences

Object-oriented programming
Chapter 4 – Iterators

Prof. Dr Kai Höfig

Types of classes

- The following classes are possible in Java:

  - (normal) class
    - A normal, non-static class per `.java` file
    - The class name must be the same as the file name; translation from `MyClass.java` to `MyClass.class` (bytecode).
    - It can contain any number of inner classes

```java
public class MyClass {
}
```

  - inner class
  - static inner class
  - anonymous inner class

Inner classes

- Any number of inner classes
- Visibility and validity analogous to attributes
- Within a normal class, outside of methods
- Inner can only exist in instance of outer
- In principle, can be nested as much as you want (inner in inner in inner ...)
- Access to all attributes of the outer instance
- In case of name conflicts, disambiguate with `<ClassName>.this.*`
- No `static` attributes in inner classes

```java
public class MyClass {
  private int attr;
  private static int ATTR=0;
  private class MyInnerClass{
    protected int attr;
    // protected static int ATTR=1; // compiler error!
    void innerMethod(){
      attr=2; // the attribute of MyInnerClass
      MyClass.this.attr=3; // the attribute of MyClass
    }
    }
}
```

Static inner classes

- Can be used without a direct instance of the outer class, but must be instantiated with new
- Consequently, no direct access to the non-static attributes of the outer class
- Access to the static attributes of the outer class

```
public class startupClasses {
  public static void main(String[] args) {
    MyClass.MyStaticInnerClass sik = new MyClass.MyStaticInnerClass();
    // MyClass.MyInnerClass ik = new MyClass.MyInnerClass(); // compiler error
    System.out.println(sik.method());
    System.out.println(sik.s);
    //System.out.println(MyClass.MyInnerClass.method()); // compiler error!
}
```

```
public class MyClass {
  private int attr;
  private static int ATTR=0;
…
  static class MyStaticInnerClass{
        String s = "hello";
    int method(){
      // MyClass.this.attr=0; // compiler error!
      ATTR = 10;
      return ATTR;
        }
    }
}
```

Anonymous inner class

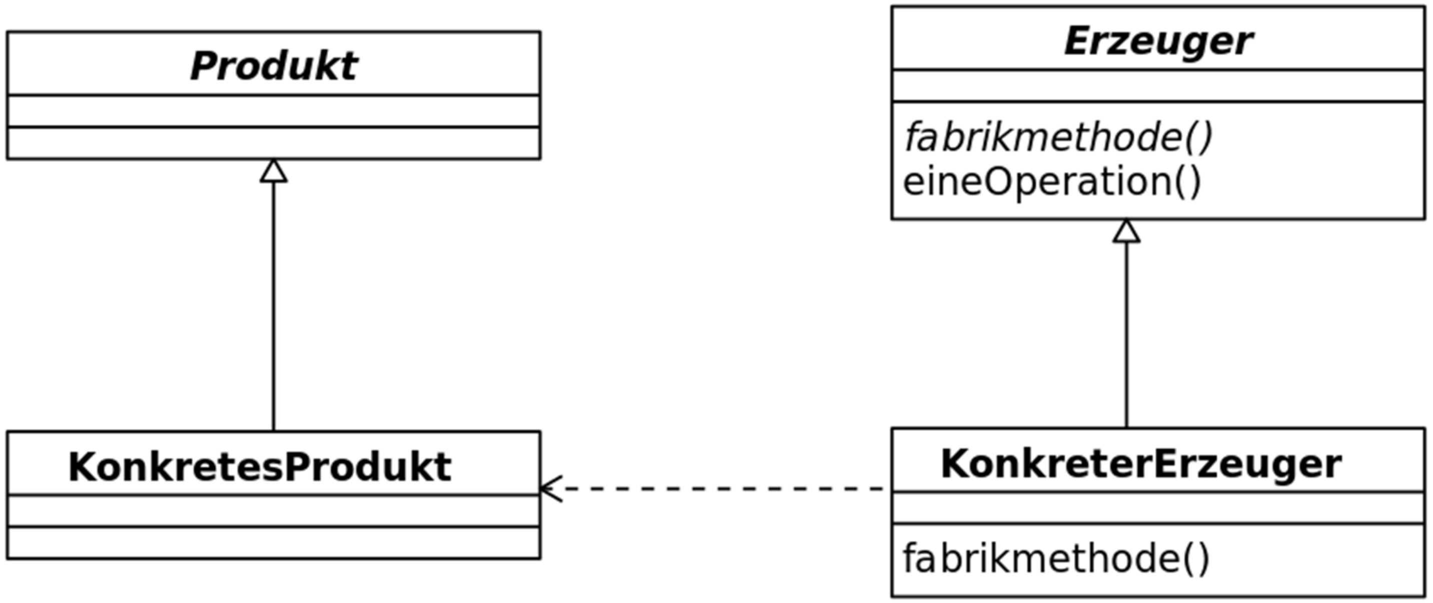
- Creates an object that implements an interface *ad hoc*
- But class declaration is unknown at runtime ("anonymous", i.e. has no name)
- Access to outer attributes only if they are practically final.

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

```
public class startupClasses {
  public static void main(String[] args) {
..
    int y=1; // or 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; // compiler error!
  }
}
```

Factory method

- The factory method is a so-called creation pattern
- A factory method returns elements to an interface
- This design pattern provides the foundation for *iterators*



For more basic design patterns go to: https://springframework.guru/gang-of-four-design-patterns/

Example factory method

```java
public interface Car {
  public void hoot();
}
```

```java
public interface CarFactory {
  public Car produce();
}
```

Factory method

```java
public class Lorry implements Car{
  @Override
  public void hoot() {
    System.out.println("Tooooooot");
    }
}

public class SmallCar implements Car{
  @Override
  public void hoot() {
    System.out.println("Hooot");
    }
}
```

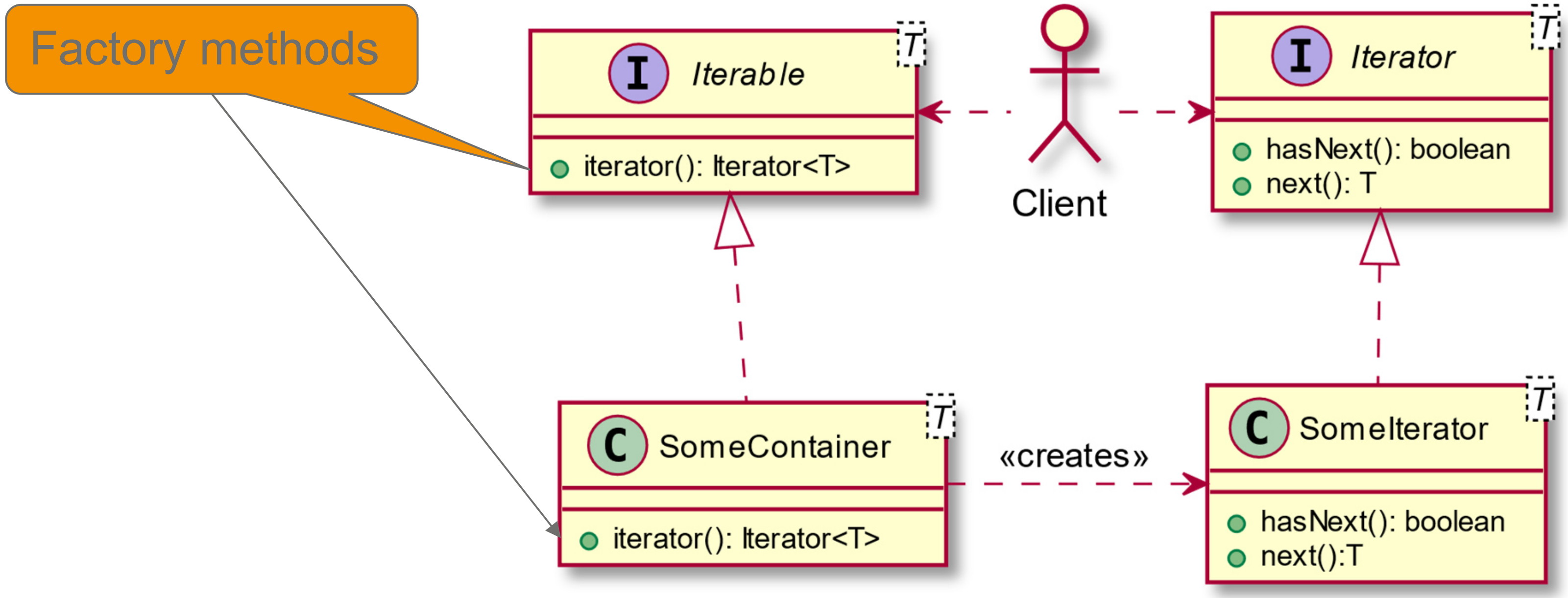
```java
public class LorryFactory implements CarFactory{
  @Override
  public Car produce() {
    return new Lorry();
    }
}

public class SmallCarFactory implements CarFactory{
  @Override
  public Car produce() {
    return new SmallCar();
    }
}
```

- What are the benefits? In the main, I can only use the interfaces again, the code is quickly interchangeable and expandable to other types of car and manufacturer.

```java
public class startupCarIndustry {
  public static void main(String[] args) {
    LorryFactory lwf = new LorryFactory();
    SmallCarFactory kwf = new SmallCarFactory();
        Car[] cars = {lwf.produce(),kwf.produce()};
    for(Car a : cars){
      a.hoot();
        }
    }
}
```

Iterator

- An `iterator` is supplied by the factory method of an `iterable`.
- Modelling the sequential access to a container structure
- Establishes a relationship between iterable and iterator
- Behavioural pattern
- Uses the factory method pattern



Using iterators

- Regular, with `while` loop*

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

- With `for-each` loop*

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

*`class IntSet implements Iterable<Integer>`

from `java.util.Iterator` and with `java.lang.Iterable`

Iteration

- In mathematics and computer science, iterating means repeating a certain action.
- In (mathematical) optimisation, this means the repeated application of a calculation rule to achieve optimum results, similar to a golfer who needs several, ever-shorter strokes for a hole.
- In computer science, iteration is usually meant in the algorithmic sense: the repeated execution of instructions. In the simplest sense, these are the `for`, `for-each` and `while` loops that repeat any applications:

Looking back: any while loop can be implemented as a for loop and vice versa

```
for (int i = 0; i < 3; i++) {
  if (i < 2)
    System.out.print((i+1) + ", ");
  else
    System.out.print(" or " + (i+1));
}
System.out.println(" - last chance - over!");
```

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

System.out.println(" - last chance - over!");
```

Iteration over simple data structures

- Today we want to look at a special application of iteration: traversal, i.e. running through data structures. What does this mean?
- From the last semester, we already know fields (arrays):

```java
// iteration over array
int[] a = {4, 1, 2, 7};


// output all elements of the array
for (i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

- If we now want to visit all the elements in our list, we would proceed in the same way as with the array:

```java
// all elements of a list
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));
```

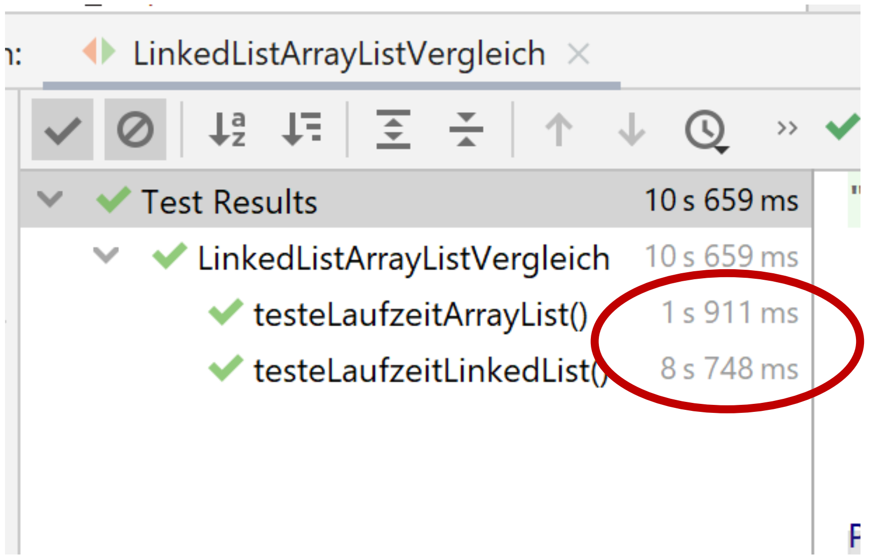
Disadvantage: we have to implement the methods for iteration again every time and they are sometimes not very efficient

Comparison of iteration over a chained list and an array

- However, we remember that ArrayList.get is implemented much more efficiently than LinkedList.get, namely in O(1) instead of O(n):

Chained list

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

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


LinkedListArrayListVergleich

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

Array list

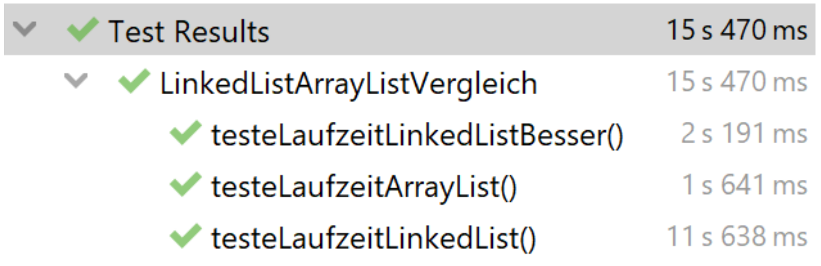
```
public T get(int i) {
  return zs[i];
}
```

See `LinkedListArrayListComparison` class in repository

So what could we do to improve that?

```
public class ListImpl<T> implements List<T> {
  …
  private Element head;
    Element next; // here we note what comes next for the iterator we have created
    …
  // we output the next elemen
  public T getNext(){
        Element e= next;
    if(next.next==null){
      next=null;
        }else{
      next=next.next;
        }
    return e.value;
    }

  // so that we can use a while loop
  public boolean hasNext(){
    if (next==null)
      return false;
    else
            return true;
    }

  // in case we want to run through again
  public void resetIterator(){
    this.next=head;
    }
  …
}
```

We note the element that we have output, which is also possible from the outside, but usually these classes are 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
```

Disadvantage: this is not standardised :/

Better: work with interfaces

- So we need something that always works the same way

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

And this thing should provide us with every data structure

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

Example with our `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;
        }
    }

  // create a new iterator
  public Iterator<T> iterator() {
    return new MyIterator();
    }


  private T[] zs;

  …
}
```

- Advantages:
  - Standardised interface for all our data structures; iterating is always the same.
  - Always the most powerful solution for any data structure
  - Easy to use

- We don't need a "reset" because we create a new `Iterator` object for every iteration

```
// now we use our Iterator and Iterable interface
Iterator<Integer> litit = lit.iterator();
while(litit.hasNext())
    System.out.println(litit.next().toString());
```

Example with our `ListImpl<T>`

- Even more compact by using an anonymous class, same benefits, same use

```java
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;
            }
        };
    }
}
```

```java
// now we use our Iterator and Iterable interface
Iterator<Integer> litit = lit.iterator();
while(litit.hasNext())
    System.out.println(litit.next().toString());
```

`java.util.Iterator` and `java.lang.Iterable`

- Java also offers these interfaces:

  https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html
  https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

- These are to be utilised in the same way (as outlined at the beginning of the lecture) and are implemented for many data structures of the Java Framework, e.g. the Java Collection Framework

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

Summary

- The iteration for container structures (such as lists or sets) is an abstraction that gives the user sequential access to the elements contained, without knowing the inner structure.
- The iterator as a behavioural pattern describes the connection between the interfaces `Iterator<T>` and `Iterable<T>`.
- The factory method is a creation pattern that is also found in the iterator pattern.
- Iterators for sequential data structures are generally easy to implement: they remember the current position.
- In contrast, iterators for tree structures, i.e. data structures whose elements have more than one successor, use an agenda: a list of items still to be visited.
- There are normal, inner, static inner and anonymous inner classes