LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Einführung in die Informatik: Programmierung und Softwareentwicklung
Wintersemester 2018/19

# Kapitel 13: Listen und Generisches Programmieren

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 16. Januar 2019

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11

## Ziele

- Implementierungen für Listen kennenlernen
- Einfach verkettete und doppelt verkettete Listen verstehen
- Java Generics kennenlernen
- Generische Programme und generisches Programmieren in Java
- Generische Klassen, Interfaces und Methoden
- Subtypenbeziehung bei Verberbung und generischen Klassen
- Listen-Implementierungen in der Java-Bibliothek kennenlernen
- Durch Listen von Objekten iterieren können



## Listen und Listenoperationen

Eine **Liste** ist eine endliche Folgen von Elementen, deren Länge (im Gegensatz zu Arrays) durch Hinzufügen und Wegnehmen von Elementen geändert werden kann. Es handelt sich um eine **dynamische** Datenstruktur.

**Übliche Operationen auf Listen**

- Hinzufügen eines Elements am Anfang (ggf. auch am Ende) einer Liste
- Entfernen eines Elements am Anfang (ggf. auch am Ende) einer Liste
- Zugriff auf Element der Liste (z.B. erstes, letztes Element oder an einer bestimmten Position)
- Berechnen der Länge der Liste
- Prüfen auf leere Liste
- Listendurchlauf



## Implementierungsarten für Listen

Es gibt mehrere Möglichkeiten Listen zu implementieren:

- **Array-Listen**
- **Verkettete Listen**

## Array-Listen

- Die Listenelemente sind in einem Array gespeichert
- Einfacher Zugriff auf Listenelemente
- Bei Einfügeoperation wird die Größe des Arrays bei Bedarf angepasst durch Anlegen eines neuen Arrays und Umkopieren.
- Das Array ist meist etwas größer als die repräsentierte Liste (partielle Arrays!), so dass nicht bei jeder Einfügeoperation (am Ende eines Arrays) umkopiert werden muss.
- In der Java-Bibliothek: Klasse `ArrayList`



## Verkettete Listen

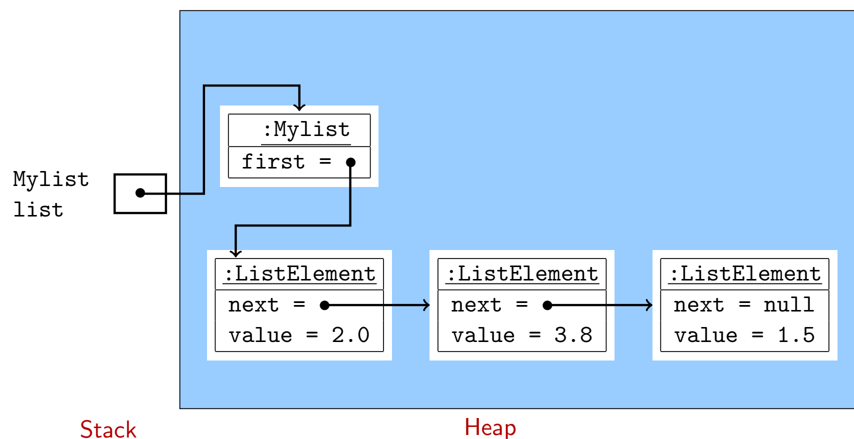
- Die Listenelemente werden als Kette von Objekten gespeichert.
- Die Größe der Liste ist dynamisch änderbar.
- Einfügen und Löschen wird einfacher
- Zugriff auf Elemente wird schwieriger
- In der Java-Bibliothek: Klasse LinkedList



## Einfach verkettete Listen

Repräsentiere Listen als Ketten von Objekten im Heap

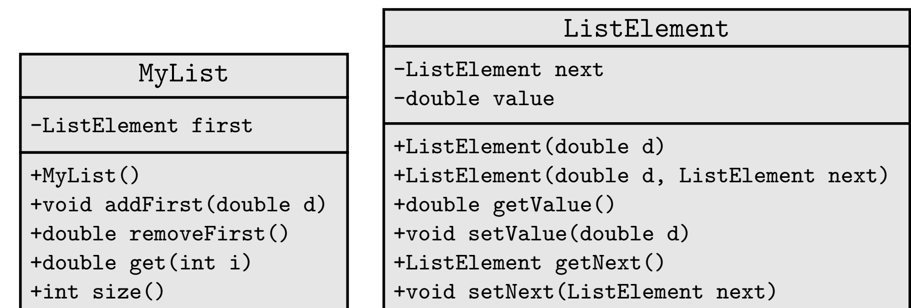
Beispiel: Repräsentation der Liste <`2.0, 3.8, 1.5`> von `double`-Werten:






## Klassen für verkettete Listen

- `MyList`: Klasse für Listen, die Methoden für die verschiedenen Operationen auf Listen bereitstellt. Objekte der Klasse `MyList` haben eine Referenz auf das erste Listenelement. Die Referenz ist `null`, wenn die Liste leer ist.
- `ListElement`: Klasse für Listenelemente. Jedes Objekt hat einen `double`-Wert und eine Referenz auf das nächste Listenelement. Die Referenz ist `null`, wenn es kein nächstes Element gibt.



## Beispiele für die Verwendung

```
MyList liste = new MyList(); // leere Liste erzeugen
liste.addFirst(10); // 10 anfuegen
System.out.println("Groesse:" + liste.size()); // Groesse ausdrucken
liste.removeFirst(); // Erstes entfernen
System.out.println("Groesse:" + liste.size()); // Groesse ausdrucken
liste.addFirst(20); // 20 vorne anfuegen
liste.addFirst(5);  // 5 vorne anfuegen
liste.addFirst(25); // 25 vorne anfuegen
liste.addFirst(1);  // 1 vorne anfuegen
// alle Elemente ausdrucken:
for (int i=0; i < liste.size(); i++) {
  System.out.print(liste.get(i) + ", ");
}
System.out.println();
System.out.println(liste.removeFirst()); // Erstes entfernen
// alle Elemente ausdrucken:
for (int i=0; i < liste.size(); i++) {
  System.out.print(liste.get(i) + ", ");
}
```



## Implementierung der Klasse ListElement

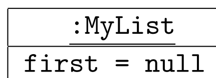
```
public class ListElement {
 private ListElement next;
 private double value;
 public ListElement(double value) {
  this.value = value;
 }
 public ListElement(double value, ListElement next) {
  this.value = value;
  this.next = next;
 }
 public double getValue() {
  return this.value;
 }
 public void setValue(double value) {
  this.value = value;
 }
 public ListElement getNext() {
  return this.next;
 }
 public void setNext(ListElement next) {
  this.next = next;
 }
}
```



## Implementierung der Klasse MyList

```
public class MyList {
    private ListElement first;
    public MyList() {
        this.first = null;
    }
    ...
```

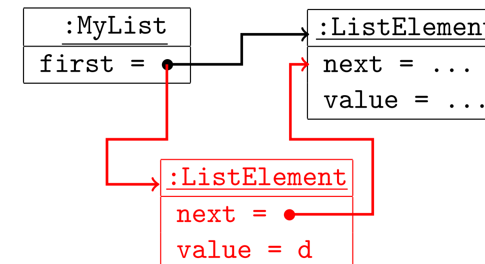
Der Konstruktor MyList() erzeugt eine leere Liste:






## Implementierung der Klasse MyList (2)

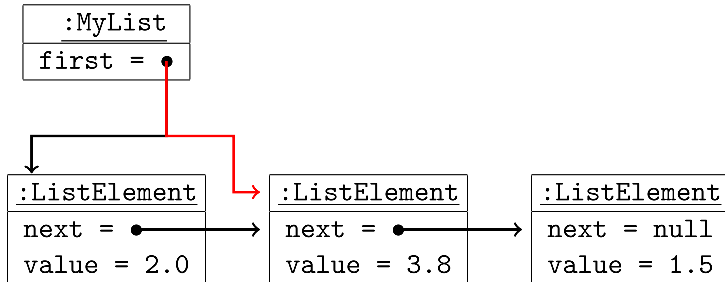
```
public void addFirst(double d) {
 ListElement newFirst = new ListElement(d,this.first);
 this.first = newFirst;
}
...
```




Beachte: Die Laufzeit von addFirst(...) ist konstant.

## Implementierung der Klasse MyList (3)

```
public double removeFirst() {
 if (first == null) {
  throw new NoSuchElementException("Die Liste ist leer!");
 }
 double value = this.first.getValue();
 this.first = this.first.getNext();
 return value;
}
```

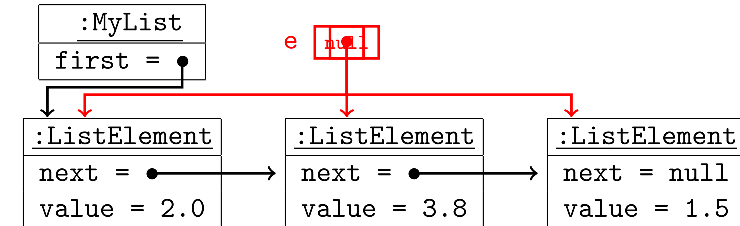



Beachte: Die Laufzeit von `removeFirst()` ist konstant.



## Implementierung der Klasse MyList (4)

```
public double get(int i) {
 ListElement e = this.first;
 while (e != null && i > 0) { e = e.getNext();
                              i--;           }
 if (e == null) {
  throw new IndexOutOfBoundsException("Falscher Index");}
 return e.getValue();
}
```




Beachte: Die Laufzeit von `get(i)` ist $\mathcal{O}(i)$ (bzw. im worst-case linear in der Listenlänge)



## Implementierung der Klasse MyList (5)

```
public int size() {
 int size = 0;
 ListElement e = this.first;
 while (e != null) {
  e = e.getNext();
  size++;
 }
 return size;
 }
```

Beachte: Die Laufzeit von `size()` ist linear in der Listenlänge

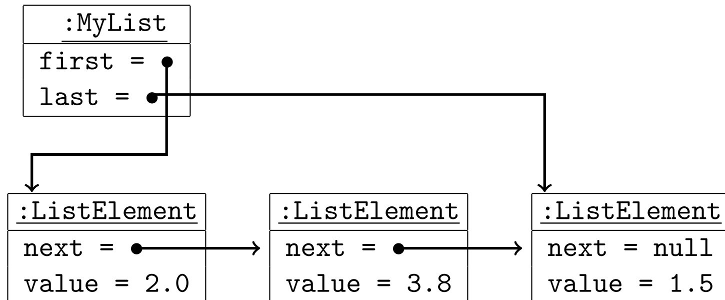


## Konstante Laufzeit für size()

```
public class MyList{
 private ListElement next;
 private int length; // Laenge mitspeichern
 public MyList() { first = null; length = 0; // neu: Laenge auf 0
 }
 public void addFirst(double d) {
  ListElement newFirst = new ListElement(d,this.first);
  this.first = newFirst; this.length++; // neu: Laenge erhoehen
 }
 public double removeFirst() {
  if (first == null) {
   throw new NoSuchElementException("Die Liste ist leer!");
  }
  double value = this.first.getValue();
  this.first = this.first.getNext(); this.length--; // neu
  return value;
 }
 public int size() {
  return this.length;
 }
```

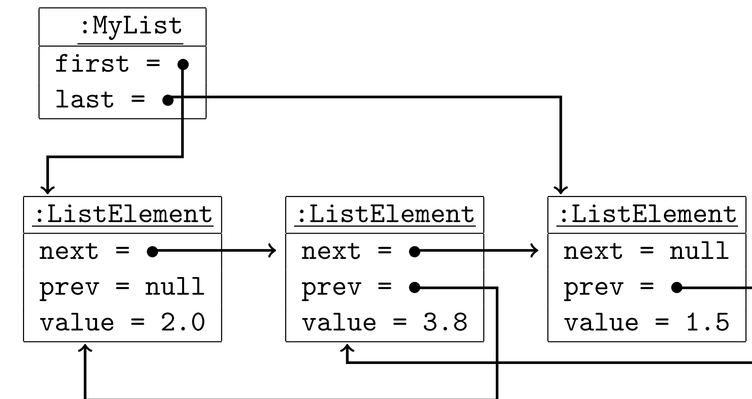
## Einfügen am Ende

- Zum Einfügen am Ende muss die ganze Liste durchlaufen werden, bis das Element gefunden ist, dessen `next`-Referenz auf `null` gesetzt ist.
- Mit einem zusätzlichen Zeiger zum letzten Element kann man auch am Ende in konstanter Zeit einfügen
- **Aber:** Löschen am Ende benötigt immer noch lineare Zeit, da man das vorletzte Element finden muss!






## Doppelt-verkettete Listen




Löschen am Ende geht nun auch in konstanter Zeit.



## Implementierung mit doppelter Verkettung (1)

```
public class ListElement {
 private ListElement next;
 private ListElement prev;
 private double value;
 public ListElement(double value) {this.value = value;}
 public ListElement(double value, ListElement next, ListElement prev) {
  this.value = value;
  this.next = next;
  this.prev = prev;
 }
 public double getValue() {return this.value;}
 public void setValue(double value) {this.value = value;}
 public ListElement getNext() {return this.next;}
 public void setNext(ListElement next) {this.next = next;}
 public ListElement getPrev() {return this.prev;}
 public void setPrev(ListElement prev) {this.prev = prev;}
}
```



## Implementierung mit doppelter Verkettung (2)

```
public class MyList {

 private ListElement first;
 private ListElement last;
 private int length;

 public MyList() {
   this.first = null;
   this.last = null;
   this.length = 0;
 }
 ...
```

## Implementierung mit doppelter Verkettung (3)

```
public void addFirst(double d) {
 // Neues Element erzeugen: next zeigt auf das alte erste Element,
 // prev zeigt auf null
 ListElement newFirst = new ListElement(d,this.first,null);
 // Wenn Liste nicht leer war, dann
 // prev-Zeiger des alten ersten Elements umhaengen
 if (this.first != null) {
   this.first.setPrev(newFirst);
 }
 // Zeiger auf erstes Element anpassen
 this.first = newFirst;
 // Wenn Liste leer war, dann auch Zeiger auf letztes Element anpassen
 if (this.last == null) {
   this.last = newFirst;
 }
 this.length++;
}
```



## Implementierung mit doppelter Verkettung (4)

```
public double removeFirst() {
 if (first == null) {
  throw new NoSuchElementException("Die Liste ist leer!");
 }
 double value = this.first.getValue();
 // Zeiger auf erstes Element anpassen
 this.first = this.first.getNext();
 // Wenn Liste leer wird, dann auch last-Zeiger anpassen
 if (this.first == null) {this.last = null;}
 this.length--;
 return value;
}
```



## Implementierung mit doppelter Verkettung (5)

```
public void addLast(double d) {
 // Neues Element erzeugen: next zeigt auf null,
 // prev zeigt auf das alte letzte Element
 ListElement newLast = new ListElement(d,null,this.last);
 // Wenn Liste nicht leer war, dann next Zeiger des letzten Elements
      anpassen
 if (this.last != null) {this.last.setNext(newLast);}
 // Zeiger auf letztes Element anpassen
 this.last = newLast;
 Wenn Liste leer war, dann auch first-Zeiger anpassen
 if (this.first == null) {this.first = newLast;}
 length++;                 // Laenge anpassen
}
```



## Implementierung mit doppelter Verkettung (6)

```
public double removeLast() {
 if (last == null) {
   throw new NoSuchElementException("Die Liste ist leer!");
 }
 double value = this.last.getValue();
 this.last = this.last.getPrev();
 // Wenn Liste leer wird, dann auch first Zeiger anpassen
 if (this.last == null) {this.first = null;}
 this.length--;
 return value;
}
```

## Implementierung mit doppelter Verkettung (7)

```
public double get(int i) {
 ListElement e = this.first;
 while (e != null && i > 0) {
  e = e.getNext();
  i--;
 }
 if (e == null) {
  throw new IndexOutOfBoundsException("Falscher Index");
 }
 return e.getValue();
}

public int size() {return this.length;}
}
```



## Implementierung mit doppelter Verkettung (7)

```
public void reverse() {
 // Umdrehen der Liste
 // aktuell zu bearbeitendes Element;
 ListElement e = this.first;
 // First und last zeiger vertauschen:
 this.first = this.last;
 this.last = e;
 // Elemente abarbeiten
 while (e != null) {
  // naechstes Element
  ListElement nexte = e.getNext();
  // next und prev vertauschen
  e.setNext(e.getPrev());
  e.setPrev(nexte);
  // zum naechsten Element gehen
  e = nexte;
 }
}
```



## Zusammenfassung der Zeitkomplexitäten

- Zugriff auf $i$-tes Element: linear
- Anfügen und Entfernen eines Elements am Anfang: konstant
- Anfügen eines Elements am Ende:
  - Einfach-verkettete Liste ohne last-Zeiger: linear
  - Einfach-verkettete Liste mit last-Zeiger:konstant
- Löschen eines Elements am Ende:
  - Einfach-verkettete Liste: linear
  - Doppelt-verkettete Liste: konstant

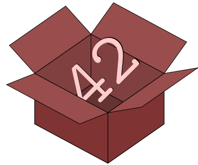


## Einschub: Java-Generics

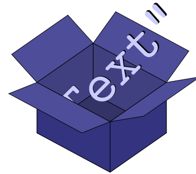
- Die Klasse `MyList` nimmt nur `Double`-Werte als Elementtypen
- Die Listen-Implementierungen in der Java Standardbibliothek erlauben beliebige Inhaltsobjekte
- Um das typsicher zu bewerkstelligen, werden dafür Java Generics verwendet.
- Im Folgenden: Erläuterung von Java Generics allgemein, danach: Rückkehr zu Listen

## Java Generics: Motivation

```
class IntegerBox {
   private Integer number;
   IntegerBox(Integer number) {
      this.number  = number;
   }
   public void set(Integer number) {
      this.number = number;
   }
   public Integer get() {
      return this.number;
   }
}
```




```
class StringBox {
   private String text;
   StringBox(String text) {
      this.text = text;
   }
   public void set(String text) {
      this.text = text;
   }
   public String get() {
       return this.text;
   }
}
```



Zweimal **nahezu der selbe** Code!



## Generisches Programmieren

**Generischer Programmcode**

**Eine** Implementierung kann für **alle / mehrere** Typen verwendet werden

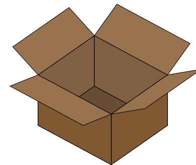
**Vorteile:**

- Keine Code-Duplikate
- Bessere Wartbarkeit
- Weniger fehleranfällig



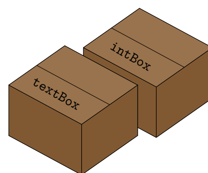
## Beispiel: Generisch Programmieren vor Java 5

```
class Box {
   private Object contents;
   Box(Object contents) {
      this.contents = contents;
   }
   public void set(Object contents) {
      this.contents = contents;
   }
   public Object get() {
      return this.contents;
   }
}
```



- Basisklasse `Object` als Inhaltstyp

```
Box textBox = new Box("Ein Text");
Box intBox  = new Box(42);
String text = (String)intBox.get();
               // Typfehler zur Laufzeit
```




- dynamische Typfehler zur Laufzeit möglich!

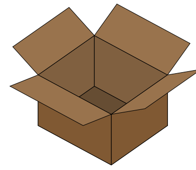


## Java Generics schaffen Abhilfe

- Mit Java 5 wurden Java Generics eingeführt
- Ziel: Typfehler zur **Compilezeit** statt zur **Laufzeit**
- Typparameter `<T>` für Klassen, Methoden, Interfaces
- Typvariable `T` kann dann wie ein konkreter Typ verwendet werden
- Beachte: Grunddatentypen können nicht für `T` eingesetzt werden (verwende `Integer` statt `int`)
- Typparameter sind wie Eingabeparameter, aber auf Typebene: Sie werden durch Typen und nicht durch Werte ersetzt
- Man spricht auch von Parametrischem Polymorphismus

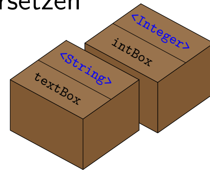
## Generische Klassen: Box mit variablem Inhalt

```
class Box<T> {
   private T contents;
   Box(T contents) {
      this.contents = contents;
   }
   public void set(T contents) {
      this.contents = contents;
   }
   public T get() {
      return this.contents;
   }
}
```



- Typparameter statt konkreter Typ
  - Instanzerzeugung: Parameter durch konkreten Typ ersetzen

```
Box<String>  textBox = new Box<String>("Ein Text");
Box<Integer> intBox  = new Box<Integer>(42);
String text = (String)intBox.get();
             // Typfehler zur Compilezeit
```



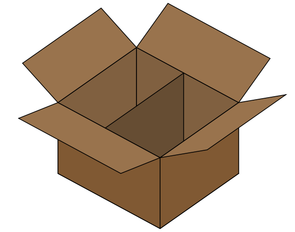



## Mehrere Typparameter: Beispiel Paare

```
class Pair<T1,T2> {
 private T1 fst;
 private T2 snd;

 Pair(T1 fst, T2 snd) {
    this.fst=fst;
    this.snd=snd;
 }
 public void setFst(T1 fst) {this.fst=fst;}
 public void setSnd(T2 snd) {this.snd=snd;}
 public T1 getFst() {return this.fst;}
 public T2 getSnd() {return this.snd;}

 Pair<T2,T1> swap() {
    return new Pair<T2,T1>(this.snd,this.fst);
 }
}
```

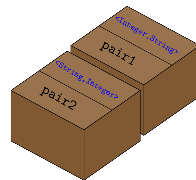




## Instanzerzeugung

Verschachtelte Typen sind möglich, z.B.:

```
Pair<Integer,String> pair1 =
   new Pair<Integer,String>(42,"Ein Text");
Pair<String,Integer> pair2 =
   pair1.swap();
```




```
Pair<Pair<Integer,String>,Pair<String,Integer>> pairOfPairs =
  new Pair<Pair<Integer,String>,Pair<String,Integer>>(pair1,pair2);
```

Kürzer mit dem **Diamant-Operator <>**:

```
Pair<Pair<Integer,String>,Pair<String,Integer>> pairOfPairs =
   new Pair<>(pair1,pair2);
```

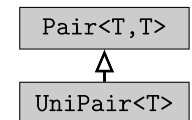
Compiler (Typchecker) leitet den Typ für `Pair<>` **automatisch** her



## Vererbung: Unterklassen können parametrisiert sein

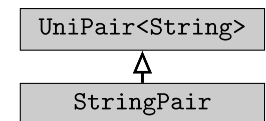
- Beispiel: Uniforme Paare

```
class UniPair<T> extends Pair<T,T> {
 UniPair(T fst, T snd) {super(fst,snd);}
}
```



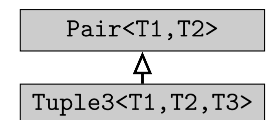

- Beispiel ohne Parameter: Paare von Strings

```
class StringPair extends UniPair<String> {
 StringPair(String fst, String snd) {
    super(fst,snd);
 }
}
```




- Beispiel mit mehr Parametern: 3-Tupel

```
class Tuple3<T1,T2,T3> extends Pair<T1,T2> {
 private T3 third;
 Tuple3(T1 fst,T2 snd, T3 third) {
    super(fst,snd);
    this.third=third;
 }
}
```



## Generische Interfaces

Typparameter sind auch für Interfaces erlaubt, z.B.

```
interface EQ<T> {
  boolean equalsTo(T t);
}
```

Äpfel mit Äpfeln und Birnen mit Birnen vergleichen:

```
class Apple implements EQ<Apple> {
 int weight;
 Apple(int weight) {
   this.weight=weight;}
 boolean equalsTo(Apple a) {
  return this.weight == a.weight};
}
```

```
class Pear implements EQ<Pear> {
 int weight;
 Pear(int weight) {
   this.weight=weight;}
 boolean equalsTo(Pear p) {
  return this.weight == p.weight;}
}
```

```
Apple apple1 = new Apple(50);
Apple apple2 = new Apple(55);
Pear  pear   = new Pear(45);
apple1.equalsTo(apple2);
apple1.equalsTo(pear); // Aepfel mit Birnen: Typfehler zur Compilezeit
```



## Einschränken des Typs: Bounds

- Idee: Begrenze die Typen, die für $T$ eingesetzt werden dürfen.
- Syntax allgemein: <T extends *Klasse* & *Interface*$_1$ &. . . & *Interface*$_n$>
- Semantik: Für T dürfen nur Typen eingesetzt werden, die
  - Unterklasse von *Klasse* sind und
  - alle *Interface*$_i$ implementieren.
- Dazu passt: <T> ist gleich zu <T extends Object>
- Beispiel: Kiste, die nur Zahlen beinhaltet:

```
class NBox<T extends Number> {
 private T contents;
 NBox(T contents) {this.contents=contents;}
}
```

Verwendung:

```
NBox<Integer> intBox = new NBox<>(10); // ok, da Integer < Number
NBox<String> textBox = new NBox<>("Ein Text"); // Compilezeit-Fehler
```



## Noch ein Beispiel: Paare vergleichen

Vergleiche $(x_1, y_1)$ mit $(x_2, y_2)$:

- Vergleiche $x_1$ mit $x_2$.
- Wenn $x_1 = x_2$, vergleiche $y_1$ und $y_2$.

Erfordert: Vergleichbarkeit der Komponenten muss gegeben sein

```
class CPair<T1 extends Comparable<T1>,T2 extends Comparable<T2>>
  implements Comparable<CPair<T1,T2>> {
   private T1 fst;
   private T2 snd;
   CPair(T1 fst, T2 snd) {this.fst = fst; this.snd=snd;}
   ...
   public T1 getFst() {return this.fst;}
   public T2 getSnd() {return this.snd;}

   public int compareTo(CPair<T1,T2> other) {
      int cmpFst = fst.compareTo(other.getFst());
      return cmpFst != 0 ? cmpFst : snd.compareTo(other.getSnd());
   }
}
```



## Generische Methoden

- auch Methoden können generisch sein und Typparameter verwenden
- Typparameter müssen vor dem Rückgabetyp angegeben werden
- Gültigkeitsbereich des Parameters ist die Methode

Beispiele:

- swap für Paare als statische Methode:

```
static <T1,T2> Pair<T2,T1> swap(Pair<T1,T2> pair) {
   return new Pair<T2,T1>(pair.snd,pair.fst);)
}
```

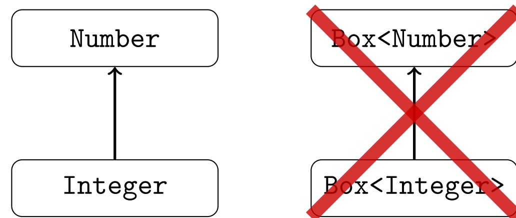
- Maximum zweier Objekte max als statische Methode:

```
static <T extends Comparable<T>> T max(T e1, T e2) {
   return e1.compareTo(e2) >= 0 ? e1 : e2;
}
```

## Generische Typen und Subtyping

Auch wenn $T_1$ echter Subtyp von $T_2$:

Klasse<$T_1$> ist **kein** Subtyp von Klasse<$T_2$>

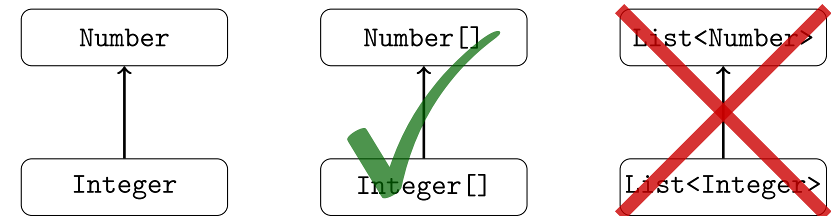



```
Integer myInteger       = 42;
Number  myNumber        = myInteger; // funktioniert
Box<Integer> integerBox = new Box<Integer>(42);
Box<Number>  numberBox  = integerBox; // Typfehler zur Compilezeit
```

Man sagt:
Die Subtypenbeziehung von Generics ist **nicht kovariant** (sondern invariant)



## Im Gegensatz: Kovariante Subtypenbeziehung bei Arrays




```
Integer[] ints = {1,2,3};
Number[]  nums = ints; // funktioniert
nums[2] = 2.5; // Laufzeitfehler
```

- Um solche Laufzeitfehler zu vermeiden, wurde für Generics die kovariante Subtypenbeziehung verboten.
- Container-Klassen wie List<T> aus Java Collections sind generisch.
- Mit Wildcards kann man ko- und kontravariant mit Generics vererben (betrachten wir nicht).



## Doppelt-verkette Listen mit generischem Inhalt

```
public class ListElement<E> {
 private ListElement<E> next;
 private ListElement<E> prev;
 private E value;
 public ListElement(E value) { this.value = value;}
 public ListElement(E value, ListElement<E> next, ListElement<E> prev)
 {
  this.value = value;
  this.next = next;
  this.prev = prev;
 }
 public E getValue() { return this.value; }
 public void setValue(E value) { this.value = value;}
 public ListElement<E> getNext() { return this.next; }
 public void setNext(ListElement<E> next) { this.next = next; }
 public ListElement<E> getPrev() { return this.prev; }
 public void setPrev(ListElement<E> prev) { this.prev = prev;}
}
```



## Doppelt-verkette Listen mit generischem Inhalt (2)

```
import java.util.NoSuchElementException;

public class MyList<E> {
 private ListElement<E> first;
 private ListElement<E> last;
 private int length;
 public MyList() {
  this.first = null;
  this.last = null;
  this.length = 0;
 }

 public void addFirst(E t) {
  ListElement<E> newFirst = new ListElement<E>(t,this.first,null);
  if (this.first != null) {this.first.setPrev(newFirst);}
  this.first = newFirst;
  if (this.last == null) {this.last = newFirst;}
  this.length++;
 }
 ...
```

## Doppelt-verkette Listen mit generischem Inhalt (3)

```
public E removeFirst() {
 if (first == null) {
  throw new NoSuchElementException("Die Liste ist leer!");
 }
 E value = this.first.getValue();
 this.first = this.first.getNext();
 if (this.first == null) {this.last = null;}
 this.length--;
 return value;
}
public void addLast(E t) {
 ListElement<E> newLast = new ListElement<E>(t,null,this.last);
 if (this.last != null) {this.last.setNext(newLast);}
 this.last = newLast;
 if (this.first == null) {this.first = newLast;}
 length++;
}
```



## Doppelt-verkette Listen mit generischem Inhalt (4)

```
public E removeLast() {
 if (last == null) {
  throw new NoSuchElementException("Die Liste ist leer!");
 }
 E value = this.last.getValue();
 this.last = this.last.getPrev();
 if (this.last == null) {this.first = null;}
 this.length--;
 return value;
}
public E get(int i) {
 ListElement<E> e = this.first;
 while (e != null && i > 0) {
  e = e.getNext();
  i--;
 }
 if (e == null) {
  throw new IndexOutOfBoundsException("Falscher Index");}
 return e.getValue();
}
```



## Doppelt-verkette Listen mit generischem Inhalt (5)

```
public int size() {
 return this.length;
}

public void reverse() {
 ListElement<E> e = this.first;
 this.first = this.last;
 this.last = e;
 while (e != null) {
   ListElement<E> nexte = e.getNext();
   e.setNext(e.getPrev());
   e.setPrev(nexte);
   e = nexte;
 }
}
}
```



## Doppelt-verkette Listen mit generischem Inhalt (6)

```
public class ListTest {
 public static void main(String[] args) {
    MyList<Integer> liste = new MyList<>(); // leere Liste erzeugen
    MyList<Integer>other= new MyList<>(); // noch eine leere Liste
    MyList<MyList<Integer>> listeVonListen = new MyList<>(); // leere
        Liste erzeugen
    liste.addFirst(1);
    liste.addFirst(2);
    liste.addFirst(3);
    liste.addFirst(4);
    liste.addFirst(5);
    listeVonListen.addFirst(liste);
    listeVonListen.addFirst(liste);
    listeVonListen.addFirst(liste);
    listeVonListen.addFirst(liste);
    other.addFirst(100);
    other.addFirst(200);
    listeVonListen.addFirst(other);
    ...
}
}
```

## Compilierung

**Typlöschung**: Der Compiler **entfernt** alle generischen Typparameter:

- Ersetze die Typvariablen durch
  - ihre Grenze, wenn es eine Bound gibt
  - `Object`, wenn es keine Bound gibt
  - erste Grenze, wenn es mehrere Bounds gibt
- Füge notwendige Typkonvertierungen ein.

Beispiele:

- `Box<Integer>`, `Box<Number>`, `Box<Box<Integer>>` werden alle zu `Box` gelöscht
- Typvariablen `T` für Typparameter `<T>` wird durch `Object` ersetzt
- Typvariablen `T` für Typparameter `<T extends Number>` wird durch `Number` ersetzt.
- Typvariablen `T` für Typparameter `<T extends Name`$_1$ `& ... & Name`$_n$`>` wird durch `Name`$_1$ ersetzt



## Beispiele

```
class Box<T> {
 private T contents
 Box(T contents) {
   this.contents = contents;
 }
 ...
}
```

wird zu

```
class Box {
 private Object contents
 Box(Object contents) {
   this.contents = contents;
 }
 ...
}
```

```
class NBox<T extends Number> {
 private T num
 NBox(T num) {
   this.num = num;
 }
 ...
}
```

wird zu

```
class NBox {
 private Number num
 NBox(Number num) {
   this.num = num;
 }
 ...
}
```



## Konsequenz der Typlöschung

Überladung für verschiedene Typparameter nicht möglich, z.B.

```
static void sortiere(MyList<Integer> l) {...}
static void sortiere(MyList<Double> l)  {...}
```

dürfen **nicht beide existieren**, da nach Typlöschung:

```
static void sortiere(MyList l) {...}
static void sortiere(MyList l) {...}
```

⇒ zwei Methoden mit gleicher Deklaration!



## Gründe für die Typlöschung

- Vermeide **Code-Bloating**:
  Nur ein Code **notwendig** für `List<Integer>`, `List<Number>`, ...
  (Im Unterschied zu C++-Templates!)
- Kompatibilität: nicht-generischer Code bleibt weiterhin **gültig**
- Das gilt auch für Java-Bytecode!
  ⇒ Schrittweise **Migration** zu generischem Code **möglich**.

Motto: **Evolution statt Revolution**

## Zusammenfassung zu Java Generics

- Java Generics ermöglichen generische Programme mit Typvariablen (parametrischer Polymorphismus)
- Generische Klassen, Interfaces und Methoden
- Sehr typsicher: Compilezeitfehler statt Laufzeitfehler
- Semantik: Die Typlöschung garantiert Abwärtskompatibilität und einfache Migration



## Listen in der Java-Bibliothek

Die Java-Bibliothek stellt Klassen für Listenimplementierungen zur Verfügung:

`LinkedList<E>`: doppelt verkettete Listen

`ArrayList<E>`: durch Arrays implementierte Listen

Beide Klassen sind generisch und können daher für beliebige Elementtyp verwendet werden.
Beispiele:

- `LinkedList<Point>`
- `LinkedList<Integer>`
- `LinkedList<Double>`
- `LinkedList<BankKonto>`
- `LinkedList<LinkedList<Double>>`



## Einige Methoden der Klasse `LinkedList<E>`

```
LinkedList<E>
---
...
---
+LinkedList()
void addFirst(E e)
+E removeFirst()
+void addLast(E e)
+E removeLast()
+E get(int i)
+int size()
+Iterator<E> iterator()
...
```



## Iteratoren

- Die Klassen für Listen in der Java-Bibliothek erlauben den Durchlauf von Listen mittels sogenannter **Iteratoren**
- Ein Iterator ist ein Objekt von dem man sich die Element der Liste nacheinander zurückgeben lassen kann
- Die Methode `iterator()` erzeugt einen Iterator für eine Liste.
- Mit der Methode `hasNext()` stellt der Iterator fest, ob es ein nächstes zu besuchendes Element gibt.
- Der Aufruf der Methode `next()` auf dem Iterator liefert das nächste Listenelement (z.B. zu Beginn das erste Element der Liste). Falls `next()` aufgerufen wird, wenn es kein nächstes Element gibt, wird eine `NoSuchElementException` geworfen.

```
LinkedList<E>
---
...
---
... +Iterator<E> iterator()
...
```

```
<<interface>>
Iterator<E>
---
+E next()
+boolean hasNext()
```

## Iteration mit Listen der Java-Standardbibliothek

Sei C eine Klasse und list eine Variable vom Typ LinkedList<C>.
Dannk kann man auf folgende Weise über die Liste list iterieren.

```
Iterator<C> it = list.iterator()
while (it.hasNext()) {
  C x = it.next();
  // Anweisungen
}
```

Es gibt eine **abkürzende Schreibweise** für die Art der Iteration:

```
for (C x : list) {
  // Anweisungen
}
```



## Beispiele

```
import java.util.*;
public class TestIterator {
  public static void main(String[] args) {
    LinkedList<Double> list = new LinkedList<>();
    list.addFirst(10.5);
    list.addFirst(100.9);
    list.addFirst(200.1);
    double sum = 0;
    Iterator<Double> it = list.iterator();
    while (it.hasNext()) {
     Double x = it.next();
     sum += x;
    }
    System.out.println(sum);
    sum = 0;
    for (Double x : list) { sum += x; }
    System.out.println(sum);
  }
}
```



## Vergleich: Verkettete Listen und Arrays

**Zeitkomplexitäten:** Zugriff auf i-tes Element:

- Verkettete Liste: linear
- Array: konstant

Anfügen und Entfernen eines Elements:

- Verkettete Liste: konstant
- Arrays: Beim ersten Element linear (Verschieben aller Einträge um 1), letztes Element konstant oder linear (wenn der Platz nicht reicht und Array umkopiert wird)

**Folgerung:**
Arrays eignen sich zur Behandlung von Sequenzen (Folgen) mit fester oder selten veränderter Anzahl von Elementen, während sich verkettete Listen besser eignen bei dynamischen Sequenzen, deren Länge sich häufig zur Laufzeit ändert.



## Zusammenfassung

- Implementierungen von einfach und doppelt-verketteten Listen
- Generische Klassen, Methoden, Interfaces
- Iteratoren
- Vor- und Nachteile von Listen und Arrays