TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 07: Collections

Java Oracle Tutorial: Collections

Karsten Weihe

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

In java.util:

- Collection
- Collections
- List
- Iterator

In java.util:

- Vector
- LinkedList
- ArrayList
- TreeSet
- HashSet

Wie der Name sagt, sind Collections Sammlungen von Elementen. Diese Elemente sind Objekte eines bestimmten Typs, und dieser Typ ist als generischer Typparameter offengehalten.

Im Package java.util finden sich die dafür bereitgestellten Klassen und Interfaces. Wie bekannt, steht die Abkürzung util für utilities, was man etwas frei als generell nützliche Sachen übersetzen könnte. Natürlich können Sie auch eigene Collection-Klassen implementieren.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

In java.util:

- Collection
- Collections
- List
- Iterator

In java.util:

- Vector
- LinkedList
- ArrayList
- TreeSet
- HashSet

Zentral ist das Interface Collection. Alle Collection-Klassen implementieren dieses Interface.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

In java.util:

- Collection
- Collections
- List
- Iterator

In java.util:

- Vector
- LinkedList
- ArrayList
- TreeSet
- HashSet

Nicht zu verwechseln mit einer Klasse namens Collections mit s hinten. Diese Klasse enthält ein paar nützliche Basisalgorithmen, wir kommen später darauf zurück, Stichwort Sortieren.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

In java.util:

- Collection
- Collections
- List
- Iterator

In java.util:

- Vector
- LinkedList
- ArrayList
- TreeSet
- HashSet

List ist ein Interface, das Collection erweitert. List wird von speziellen Collection-Klassen implementiert, die mehr an Funktionalität bieten als andere Collection-Klassen, nämlich eine Reihenfolge auf den Elementen, auch dazu gleich mehr.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

In java.util:

- Collection
- Collections
- List
- Iterator

In java.util:

- Vector
- LinkedList
- ArrayList
- TreeSet
- HashSet

Den Durchlauf durch die Elemente einer Collection kann man auf verschiedene Weise organisieren. Im Laufe der letzten Jahrzehnte hat sich – unabhängig von der Programmiersprache – eine besonders flexible Möglichkeit durchgesetzt, nämlich den Durchlauf nicht von der Collection-Klasse selbst, sondern von einer eigens dafür entwickelten, zusätzlichen Klasse durchführen zu lassen, für die sich der Name Iterator etabliert hat. Man iteriert dann über die Elemente mittels eines Iterators. Dieses Konzept und seine Umsetzung in Java als ein Interface namens Iterator werden wir ebenfalls besprechen.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

In java.util:	In java.util:
Collection	Vector
Collections	LinkedList
List	ArrayList
Iterator	TreeSet
	HashSet

Hier farblich unterlegt ein paar Beispiele für Collection-Klassen in java.util, also Klassen, die Interface Collection implementieren. In den verschiedenen Klassen sind die Sammlungen von Elementen unterschiedlich intern organisiert. Es gibt nicht die „eierlegende Wollmilchsau“, die für alle Zwecke gut geeignet ist, sondern verschiedene Collection-Klassen sind für unterschiedliche Zwecke besonders gut geeignet, deshalb diese Vielfalt.

Im Rahmen dieses Kapitels können wir auf die einzelnen Möglichkeiten, eine Sammlung von Elementen zu organisieren, nicht eingehen. Das wäre auch kein programmiersprachliches Thema mehr, sondern gehört in den Informatikbereich Algorithmen und Datenstrukturen.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<Number> c1 = new ArrayList<Number> ();
Collection<Number> c2 = new HashSet<Number> ();
Collection<Number> c3 = new Vector<Number> ();

for ( int i=0; i<100; i++ ) {
    c1.add ( new Integer(i) );
    c2.add ( new Double(i) );
    c3.add ( new Float(i) );
}
```

Aber da die Algorithmen und Datenstrukturen in Java gut in den Klassen eingekapselt sind, können wir uns hier auf die *Nutzung* dieser Klassen konzentrieren, ohne verstehen zu müssen, wie diese Klassen im private-Bereich nun genau aussehen.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<Number> c1 = new ArrayList<Number> ();
Collection<Number> c2 = new HashSet<Number> ();
Collection<Number> c3 = new Vector<Number> ();

for ( int i=0; i<100; i++ ) {
    c1.add ( new Integer(i) );
    c2.add ( new Double(i) );
    c3.add ( new Float(i) );
}
```

Wir richten eine Variable vom Interface Collection ein, instanziiert wieder mit der Klasse Number als Beispiel.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<Number> c1 = new ArrayList<Number> ();
Collection<Number> c2 = new HashSet<Number> ();
Collection<Number> c3 = new Vector<Number> ();

for ( int i=0; i<100; i++ ) {
    c1.add ( new Integer(i) );
    c2.add ( new Double(i) );
    c3.add ( new Float(i) );
}
```

Vom Interface Collection kann natürlich kein Objekt eingerichtet werden, wohl aber zum Beispiel von der Klasse ArrayList. Diese implementiert Interface Collection, so dass die Adresse des Objektes der Variablen c1 zugewiesen werden kann. Der Typparameter muss natürlich identisch sein mit dem bei der Definition von c1 als Collection.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<Number> c1 = new ArrayList<Number> ();
Collection<Number> c2 = new HashSet<Number> ();
Collection<Number> c3 = new Vector<Number> ();

for ( int i=0; i<100; i++ ) {
    c1.add ( new Integer(i) );
    c2.add ( new Double(i) );
    c3.add ( new Float(i) );
}
```

Genau dasselbe bei der Klasse HashSet.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<Number> c1 = new ArrayList<Number> ();
Collection<Number> c2 = new HashSet<Number> ();
Collection<Number> c3 = new Vector<Number> ();

for ( int i=0; i<100; i++ ) {
    c1.add ( new Integer(i) );
    c2.add ( new Double(i) );
    c3.add ( new Float(i) );
}
```

Oder auch bei der Klasse Vector.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<Number> c1 = new ArrayList<Number> ();
Collection<Number> c2 = new HashSet<Number> ();
Collection<Number> c3 = new Vector<Number> ();

for ( int i=0; i<100; i++ ) {
    c1.add ( new Integer(i) );
    c2.add ( new Double(i) );
    c3.add ( new Float(i) );
}
```

Der Konstruktor mit leerer Parameterliste richtet jeweils eine leere Sammlung von Elementen ein. Als nächstes füllen wir alle drei Objekte jeweils mit 100 Elementen.

***Nebenbemerkung*: Werte wie die Zahl 100 hier sollte man bekanntlich *nicht* im Programm explizit hinschreiben, sondern dafür eine Konstante mit einem sprechenden Namen definieren. Aber hier geht es ja nur um Illustration, daher wählen wir hier ausnahmsweise den einfachen Weg.**

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<Number> c1 = new ArrayList<Number> ();
Collection<Number> c2 = new HashSet<Number> ();
Collection<Number> c3 = new Vector<Number> ();

for ( int i=0; i<100; i++ ) {
    c1.add ( new Integer(i) );
    c2.add ( new Double(i) );
    c3.add ( new Float(i) );
}
```

Zum Einfügen eines neuen Elements bietet das Interface Collection die Methode add. Der formale Parameter der Methode add ist vom Typparameter, hier also vom Typ Number. Daher können wir so wie hier den aktualen Parameter vom Typ Integer wählen.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<Number> c1 = new ArrayList<Number> ();
Collection<Number> c2 = new HashSet<Number> ();
Collection<Number> c3 = new Vector<Number> ();

for ( int i=0; i<100; i++ ) {
    c1.add ( new Integer(i) );
    c2.add ( new Double(i) );
    c3.add ( new Float(i) );
}
```

Beziehungsweise Double.

Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<Number> c1 = new ArrayList<Number> ();
Collection<Number> c2 = new HashSet<Number> ();
Collection<Number> c3 = new Vector<Number> ();

for ( int i=0; i<100; i++ ) {
    c1.add ( new Integer(i) );
    c2.add ( new Double(i) );
    c3.add ( new Float(i) );
}
```

Beziehungsweise Float.

Nur um das Beispiel einfach zu halten, sind in jeder Collection nur Elemente derselben Klasse gespeichert, Integer, Double oder Float. Da der Typparameter Number ist, hätten wir genauso gut auch mischen können, also Integer, Double und Float in *einer* Collection.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

add	contains
addAll	containsAll
size	clear
isEmpty	remove

Hier eine Auswahl von Methoden im Interface Collection. Wir gehen nicht im Detail auf diese Methoden ein, sondern nur kurz und übersichtsweise.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

add	contains
addAll	containsAll
size	clear
isEmpty	remove

Methode add hatten wir auf der letzten Folie schon kennen gelernt. Jetzt schauen wir sie uns genauer an.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
boolean add ( T element ) throws
        UnsupportedOperationException,
        NullPointerException,
        ClassCastException,
        IllegalArgumentException,
        IllegalStateException
```

Das ist der Methodenkopf der Methode add von Interface Collection.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
boolean add ( T element ) throws
        UnsupportedOperationException,
        NullPointerException,
        ClassCastException,
        IllegalArgumentException,
        IllegalStateException
```

Wie wir schon gesehen haben, ist der Parameter das Element, das in die Collection eingefügt werden soll.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
boolean add ( T element ) throws
        UnsupportedOperationException,
        NullPointerException,
        ClassCastException,
        IllegalArgumentException,
        IllegalStateException
```

Zurückgeliefert wird genau dann true, wenn das Element tatsächlich eingefügt wurde und keine Exception geworfen wurde.

Hintergrund ist, dass manche Implementationen von Collection es nicht erlauben, dass zu einem Element, das schon in einer Collection vorhanden ist, noch einmal dasselbe oder ein wertgleiches Element eingefügt wird. Versucht man das dennoch, dann wird false zurückgeliefert. Bei Implementationen von Collection, in denen mehrere wertgleiche Elemente erlaubt sind, wird daher *immer* true zurückgeliefert.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
boolean add ( T element ) throws
        UnsupportedOperationException,
        NullPointerException,
        ClassCastException,
        IllegalArgumentException,
        IllegalStateException
```

***Vorgriff*: Diese Methode add ist ein Beispiel für Methoden, die laut offizieller Dokumentation nicht von jeder implementierenden Klasse wirklich wie beschrieben implementiert sein muss. Sie darf auch stattdessen so implementiert sein, dass sie nichts anderes tut, als eine UnsupportedOperationException zu werfen, um anzuzeigen, dass die Methode nur deshalb implementiert ist, weil sie nun einmal implementiert werden *muss*, damit die Klasse nicht abstrakt ist. Aber eigentlich ist sie ausgelassen, weil die Datenstrukturen, die die Menge der Elemente intern realisieren, eine solche Operation nicht erlauben. Dass Methoden nicht wie beschrieben implementiert werden müssen, sehen wir uns im Kapitel zu Polymorphie genauer an.**

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
boolean add ( T element ) throws
        UnsupportedOperationException,
        NullPointerException,
        ClassCastException,
        IllegalArgumentException,
        IllegalStateException
```

Manche Implementationen von Interface Collection akzeptieren nicht, dass null als Listenelement eingefügt wird. Ist der dynamische Typ von dieser Art und ruft man add dennoch mit aktualem Parameterwert null auf, so wird diese Exception geworfen.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
boolean add ( T element ) throws
        UnsupportedOperationException,
        NullPointerException,
        ClassCastException,
        IllegalArgumentException,
        IllegalStateException
```

Die Fälle, in denen eine Exception von einer dieser drei Exception-Klassen geworfen wird, interessieren in der FOP nicht. Daher beenden wir hier die Erläuterung von Methode add und fahren mit der nächsten Methode von Collection fort.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

add	contains
addAll	containsAll
size	clear
isEmpty	remove

Methode addAll erhält nicht ein einzelnes Element, sondern eine andere Collection als Parameter und fügt alle Elemente aus dieser Collection in das Collection-Objekt ein, mit dem die Methode aufgerufen wird.

Die throws-Klausel ist identisch zu der von add. Mit den throws-Klauseln der einzelnen Methoden beschäftigen wir uns hier nicht mehr weiter, Sie können die Details in der offiziellen Dokumentation von Interface Collection bei Bedarf problemlos nachschlagen.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

add	contains
addAll	containsAll
size	clear
isEmpty	remove

Die Methode size hat keine Parameter und gibt als int die Anzahl der Elemente aus, die momentan in der Collection gespeichert sind, ähnlich wie das Attribut length bei Arrays oder die Methode length bei Strings.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

add	contains
addAll	containsAll
size	clear
isEmpty	remove

Die ebenfalls parameterlose Methode isEmpty ist boolesch und liefert genau dann true zurück, wenn die Collection momentan keine Elemente enthält, also genau dann, wenn Methode size den Wert 0 zurückliefert.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

add	contains
addAll	containsAll
size	clear
isEmpty	remove

Die boolesche Methode contains hat einen Parameter vom Typ Object. Daher kann ein Objekt von einer beliebigen Klasse mit contains darauf getestet werden, ob es in der Collection mindestens einmal enthalten ist. Für den Test wird die Methode equals verwendet, die schon für Klasse Object eingerichtet und daher von allen Klassen ererbt oder überschrieben wird und somit für jeden möglichen aktualen Parameter vorhanden ist.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<String> coll = new  ..... ;
String str1 = new String ( “Hallo“ );
String str2 = new String ( “Hallo“ );
coll.add ( str1 );
if ( coll.contains(str2) )
```

Wie das mit contains genau funktioniert, sehen wir uns an einem kleinen, illustrativen Beispiel an.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<String> coll = new  ..... ;
String str1 = new String ( “Hallo“ );
String str2 = new String ( “Hallo“ );
coll.add ( str1 );
if ( coll.contains(str2) )
```

Wieder eine Variable von Interface Collection, der Typparameter ist instanziiert mit Klasse String.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<String> coll = new ..... ;
String str1 = new String ( “Hallo“ );
String str2 = new String ( “Hallo“ );
coll.add ( str1 );
if ( coll.contains(str2) )
```

Der dynamische Typ von coll spielt keine Rolle für das Beispiel und ist daher ausgelassen.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<String> coll = new  ..... ;
String str1 = new String ( “Hallo“ );
String str2 = new String ( “Hallo“ );
coll.add ( str1 );
if ( coll.contains(str2) )
```

Zwei String-Variable, die auf unterschiedliche, voneinander völlig unabhängige Objekte verweisen. Diese beiden Objekte enthalten aber dieselben fünf Zeichen.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<String> coll = new  ..... ;
String str1 = new String ( “Hallo“ );
String str2 = new String ( “Hallo“ );
coll.add ( str1 );
if ( coll.contains(str2) )
```

Eines der beiden String-Objekte wird in die Collection eingefügt.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<String> coll = new  ..... ;
String str1 = new String ( “Hallo“ );
String str2 = new String ( “Hallo“ );
coll.add ( str1 );
if ( coll.contains(str2) )
```

Und mit dem anderen String-Objekt wird nun Methode contains aufgerufen.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<String> coll = new  ..... ;
String str1 = new String ( “Hallo“ );
String str2 = new String ( “Hallo“ );
coll.add ( str1 );
if ( coll.contains(str2) )                    // true
```

Das Ergebnis ist true, denn Methode equals ist so für die Klasse String überschrieben, dass nicht die Objekte selbst identisch sein müssen, sondern es reicht, wenn die beiden Objekte dieselbe Zeichenkette enthalten, also wieder Wertgleichheit, nicht Objektidentität.

In diesem konkreten Beispiel wurde vor Anwendung der Methode contains nur ein einzelnes Objekt eingefügt. Davon hängt das Ergebnis selbstverständlich nicht ab: Auch wenn Sie eine Million Objekte vor diesem und noch einmal eine Million nach diesem eingefügt hätten, bevor Sie contains aufrufen, ist das Ergebnis natürlich dasselbe.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<String> coll = new  ..... ;
coll.add ( null );
if ( coll.contains(null) )
```

Noch eine kleine Variation dieses Beispiels, um zu sehen, was passiert, wenn null anstelle eines Objektes eingefügt und dann auf Enthaltensein getestet wird. Dafür muss natürlich eine Implementation von Interface Collection gewählt werden, die das Einfügen von null akzeptiert und eben *nicht* beim Versuch, null einzufügen, eine NullPointerException wirft.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<String> coll = new  ..... ;
coll.add ( null );
if ( coll.contains(null) )
```

Wir fügen also jetzt null anstelle eines Objektes ein.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Collection<String> coll = new  ..... ;
coll.add ( null );
if ( coll.contains(null) )                    //  true
```

Und befragen die Collection, ob mindestens ein Element mit Wert null enthalten ist. Auch in diesem Fall ist das Ergebnis wieder true.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

add	contains
addAll	containsAll
size	clear
isEmpty	remove

Zurück zu unserer Auswahl von Methoden im Interface Collection, wo wir bei contains stehengeblieben waren.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

add	contains
addAll	containsAll
size	clear
isEmpty	remove

Methode containsAll verhält sich zu contains im Prinzip wie addAll zu add. Diese Methode hat nicht nur ein einzelnes Element wie contains, sondern eine ganze Collection als formalen Parameter. Sie liefert genau dann true zurück, wenn contains für alle Elemente in dieser anderen Collection true zurückliefert.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

add	contains
addAll	containsAll
size	clear
isEmpty	remove

Die parameterlose Methode clear entfernt sämtliche momentan gespeicherten Elemente aus der Collection, mit der diese Methode aufgerufen wird. Unmittelbar nach Aufruf von clear ist die Collection, mit der clear aufgerufen wurde, also leer. Anders gesagt: isEmpty liefert true zurück unmittelbar nach Anwendung von clear.

Methoden von Collection

TECHNISCHE UNIVERSITÄT DARMSTADT

add	contains
addAll	containsAll
size	clear
isEmpty	remove

Die boolesche Methode remove hat wie contains einen Parameter von Typ Object und ist boolesch wie contains. Sie liefert true auch genau im selben Fall wie contains zurück, nämlich wenn der Parameter mindestens einmal via Methode equals in der Collection gefunden wird. Auch bei null ist alles dasselbe wie bei contains.

Wie der Name sagt, hat remove aber noch einen Seiteneffekt: Eines dieser Vorkommen des Parameters wird aus der Collection entfernt. Bei mehreren wertgleichen Vorkommen entscheidet die Implementation der Collection-Klasse, welches entfernt wird.

List

TECHNISCHE UNIVERSITÄT DARMSTADT

Auswahl zusätzliche Methoden von List:

indexOf

set

add

get

Das Interface namens List erweitert das Interface Collection, ebenfalls mit dem Elementtyp als Typparameter. Die entscheidende Erweiterung ist konzeptionell: In einer Collection gibt es erst einmal keine Reihenfolge der Elemente, ähnlich wie in einer mathematischen Menge. In einer List hingegen *ist* auf den Elementen eine Reihenfolge definiert. Wie in einem Array können die Elemente über einen Index 0, 1, 2, 3 und so weiter angesprochen werden. Dieser Index wird synonym auch Position genannt.

List

TECHNISCHE UNIVERSITÄT DARMSTADT

Auswahl zusätzliche Methoden von List:

indexOf

set

add

get

Die Methode indexOf hat einen Parameter vom Typ Object und liefert den ersten Index zurück, an dem der Parameter zu finden ist, wieder gefunden mittels Methode equals. Falls der Parameter *nicht* in der List gefunden wird, so wird der „unmögliche“ Index -1 zurückgeliefert.

List

TECHNISCHE UNIVERSITÄT DARMSTADT

Auswahl zusätzliche Methoden von List:

- indexOf
- set
- add
- get

Methode set müssen wir uns genauer anschauen.

List

TECHNISCHE UNIVERSITÄT DARMSTADT

```
T set ( int index, T element ) throws
        IndexOutOfBoundsException,
        UnsupportedOperationException,
        NullPointerException,
        ClassCastException,
        IllegalArgumentException
```

Methode set hat zwei Parameter: einen Index und einen Parameter vom Elementtyp. Der Wert, der in der Liste momentan an diesem Index gespeichert wird, wird durch den zweiten Parameter ersetzt.

List

TECHNISCHE UNIVERSITÄT DARMSTADT

```
T set ( int index, T element ) throws
        IndexOutOfBoundsException,
        UnsupportedOperationException,
        NullPointerException,
        ClassCastException,
        IllegalArgumentException
```

Rückgabe ist der alte Wert, also das Element, das vorher an diesem Index war.

List

TECHNISCHE UNIVERSITÄT DARMSTADT

```
T set ( int index, T element ) throws
        IndexOutOfBoundsException,
        UnsupportedOperationException,
        NullPointerException,
        ClassCastException,
        IllegalArgumentException
```

Falls der Index nicht im Indexbereich der Liste ist, wird wie bei Arrays eine IndexOutOfBoundsException geworfen.

List

TECHNISCHE UNIVERSITÄT DARMSTADT

```
T set ( int index, T element ) throws
        IndexOutOfBoundsException,
        UnsupportedOperationException,
        NullPointerException,
        ClassCastException,
        IllegalArgumentException
```

Zu den weiteren Exception-Klassen in der throws-Klausel von set haben wir schon bei Methode add von Collection alles für die FOP Notwendige gesagt.

List

TECHNISCHE UNIVERSITÄT DARMSTADT

Auswahl zusätzliche Methoden von List:

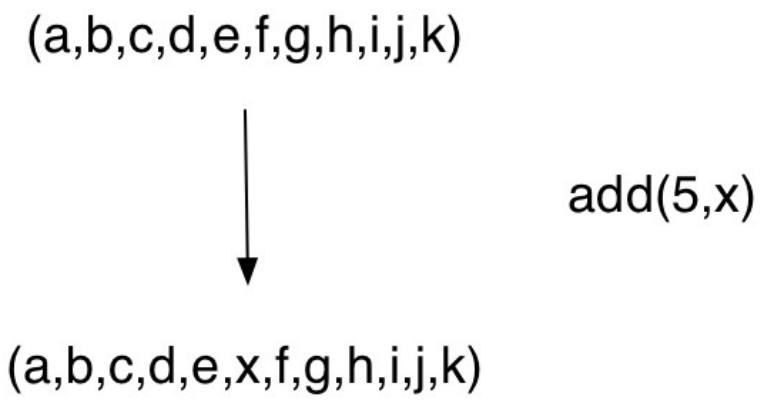
- indexOf
- set
- add
- get

Neben der von Collection ererbten Methode add, die wir schon gesehen haben, hat List eine weitere Methode add, die wie set neben dem einzufügenden Element noch einen Index als Parameter hat. Wie bei set ist der Index der erste der beiden Parameter.

Diese Methode add ist eigentlich in jeder Hinsicht identisch zu set, mit einer Ausnahme: Methode set *überschreibt* das Element am gegebenen Index mit dem neuen Wert, wohin gegen Methode add das neue Element an diesem Index *einfügt*, ohne ein anderes Element zu entfernen. Das neue Element wird also unmittelbar vor dem Element eingefügt, das bisher am angegebenen Index stand. Dieses und alle weiteren Elemente mit höherem Index rutschen entsprechend um 1 in ihrem Index weiter.

List

TECHNISCHE UNIVERSITÄT DARMSTADT



Ein einfaches, schematisches Beispiel, mit einzelnen Buchstaben als Listenelementen, das heißt, Character instanziiert den Typparameter: Oben ist die Liste *vor* Aufruf von add. Das Element f ist an Index 5. An Index 5 wird nun x eingefügt. Dadurch ist f von nun an an Index 6, g an Index 7 und so weiter.

List

TECHNISCHE UNIVERSITÄT DARMSTADT

Auswahl zusätzliche Methoden von List:

indexOf

set

add

get

Methode get schließlich hat einen ganzzahligen Index als Parameter und liefert das Element an dieser Position zurück. Falls der aktuale Parameterwert kein Index der Liste ist, wird eine IndexOutOfBoundsException geworfen.

TECHNISCHE UNIVERSITÄT DARMSTADT

Wildcards auf Collections und Listen

Im Abschnitt zu Wildcards in Kapitel 06 hatten wir angekündigt, dass wir beim Thema Collections noch einmal auf Wildcards zu sprechen kommen werden. Tatsächlich lassen sich hier die dort formulierten Empfehlungen für In-Parameter und Out-Parameter noch einmal gut motivieren.

Wildcards Collections / Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean containsNull ( Collection<? extends Number> coll ) {
    return coll.contains ( null );
}

public double getValue ( List<? extends Number> list, int index ) {
    return list.get(index).doubleValue();
}
```

Zwei Beispiele für In-Argumente, einmal Collection, einmal List.

Wildcards Collections / Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean containsNull ( Collection<? extends Number> coll ) {
    return coll.contains ( null );
}

public double getValue ( List<? extends Number> list, int index ) {
    return list.get(index).doubleValue();
}
```

In beiden Fällen wird jeweils nur eine Methode aufgerufen, die lesend auf die Collection beziehungsweise List zugreift.

Wildcards Collections / Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean containsNull ( Collection<? extends Number> coll ) {
    return coll.contains ( null );
}

public double getValue ( List<? extends Number> list, int index ) {
    return list.get(index).doubleValue();
}
```

Insbesondere im zweiten Fall sieht man deutlich, dass das extends im Typparameter der Liste kein Problem bei In-Parametern darstellt: Alle Klassen, die vom Typparameter Number direkt oder indirekt abgeleitet sind, haben die Methode doubleValue.

***Erinnerung*: Wo immer das extends kein Problem darstellt – also bei reinen In-Parametern –, ist es empfehlenswert, denn dann kann die Methode nicht nur für die nach extends angegebene Klasse, sondern auch für alle davon direkt oder indirekt abgeleiteten Klassen verwendet werden, im Beispiel also nicht nur für Listen von Number, sondern auch für Listen von Integer, Double und so weiter.**

Wildcards Collections / Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean containsNull ( Collection<?> coll ) {
   return coll.contains ( null );
}

public Object getValue ( List<?> list, int index ) {
   return list.get ( index );
}
```

Dasselbe noch einmal nun für „extends Object“ statt „extends Number“, wobei wir ja wissen, dass in diesem Fall auch einfach nur das Fragezeichen ohne „extends Object“ in die spitzen Klammern geschrieben werden kann. Natürlich muss sich dieses Beispiel auf Verwendungsmöglichkeiten beschränken, die schon für Object definiert sind. Das ist auf dieser Folie der Fall.

Wildcards Collections / Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void addPi ( Collection<? super Double> coll ) {
   coll.add ( Math.PI );
}

public void addSqrt2 ( List<? super Double> list, int index ) {
   list.add ( index, Math.sqrt(2) );
}
```

Im letzten Beispiel dieses Abschnitts, hier auf dieser Folie, sind die Collection beziehungsweise die List ein Out-Parameter der jeweiligen Beispielmethode, wieder oben eine Beispielmethode mit einer Collection, unten eine mit einer List.

Wildcards Collections / Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void addPi ( Collection<? super Double> coll ) {
   coll.add ( Math.PI );
}

public void addSqrt2 ( List<? super Double> list, int index ) {
   list.add ( index, Math.sqrt(2) );
}
```

In beiden Methoden rufen wir jeweils eine Methode auf, die nur schreibend auf die Collection beziehungsweise Liste zugreift. Einen lesenden Zugriff wie in den vorherigen Beispielen gibt es hier nicht.

Wildcards Collections / Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void addPi ( Collection<? super Double> coll ) {
   coll.add ( Math.PI );
}

public void addSqrt2 ( List<? super Double> list, int index ) {
   list.add ( index, Math.sqrt(2) );
}
```

Das passt wieder problemlos zu einer Beschränkung des Typparameters von Collection beziehungsweise List nach unten, denn ein double-Wert wird durch Boxing zu einem Objekt der Klasse Double, und das kann natürlich eingefügt werden in eine Collection oder Liste von Double, von jeder direkten oder indirekten Basisklasse von Double sowie von jedes Interface, das durch Double (indirekt) implementiert ist.

TECHNISCHE UNIVERSITÄT DARMSTADT

Listen von Lambda-Ausdrücken

***Erinnerung*: Im Kapitel 04c hatten wir einen Abschnitt zu Functional Interfaces und Lambda-Ausdrücken und hatten dort insbesondere auch gesehen, dass man unter anderem Arrays von Lambda-Ausdrücken haben kann.**

Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntPredicate [ ] predicates = new IntPredicate [ 6 ];

predicates [ 0 ] = n -> n % 2 == 1;
predicates [ 1 ] = n -> n > 0;
predicates [ 2 ] = n -> n * n < 100;
predicates [ 3 ] = predicates[0].negate();
predicates [ 4 ] = predicates[1].and ( predicates[2] );
predicates [ 5 ] = predicates[3].or ( predicates[4] );
```

Hier ist noch einmal eins-zu-eins das Beispiel für ein Array aus Lambda-Ausdrücken aus Kapitel 04c. Wir werden diese sechs Lambda-Ausdrücke jetzt alternativ in einer Liste speichern.

***Nebenbemerkung*: Auch wenn wir zwischenzeitlich in Kapitel 06 das generische Interface Predicate kennen gelernt haben, bleiben wir in diesem Abschnitt bei IntPredicate, damit alles möglichst analog bleibt.**

Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<IntPredicate> predicates = new ..... ();

predicates.add ( n -> n % 2 == 1 );
predicates.add ( n -> n > 0 );
predicates.add ( n -> n * n < 100 );
predicates.add ( predicates.get(0).negate() );
predicates.add ( predicates.get(1).and ( predicates.get(2) );
predicates.add ( predicates.get(3).or ( predicates.get(4) );
```

Die Transformation von Speicherung in einem Array zur Speicherung in einer Liste ist ziemlich schnörkellos.

Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<IntPredicate> predicates = new ..... ();

predicates.add ( n -> n % 2 == 1 );

predicates.add ( n -> n > 0 );

predicates.add ( n -> n * n < 100 );

predicates.add ( predicates.get(0).negate() );

predicates.add ( predicates.get(1).and ( predicates.get(2) );

predicates.add ( predicates.get(3).or ( predicates.get(4) );
```

Anstelle eines Array von IntPredicate eben eine leere Liste von IntPredicate.

Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<IntPredicate> predicates = new ..... ();

predicates.add ( n -> n % 2 == 1 );
predicates.add ( n -> n > 0 );
predicates.add ( n -> n * n < 100 );
predicates.add ( predicates.get(0).negate() );
predicates.add ( predicates.get(1).and ( predicates.get(2) );
predicates.add ( predicates.get(3).or ( predicates.get(4) );
```

Anstelle einer einfachen Zuweisung an die Arraykomponente muss jetzt bei Listen die Methode add aufgerufen werden.

Die von Collection ererbte Methode add hat ja keinen Index als Parameter. Bei einer Implementation von List gibt es die Garantie, dass das neue Element ganz am Ende der Liste angehängt wird, also hinter die letzte bisherige Position. Daher werden die sechs Lambda-Ausdrücke in dieser Liste in derselben Reihenfolge gespeichert wie zuvor im Array.

Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<IntPredicate> predicates = new ..... ();

predicates.add ( n -> n % 2 == 1 );
predicates.add ( n -> n > 0 );
predicates.add ( n -> n * n < 100 );
predicates.add ( predicates.get(0).negate() );
predicates.add ( predicates.get(1).and ( predicates.get(2) );
predicates.add ( predicates.get(3).or ( predicates.get(4) );
```

Methode get wird benötigt, um auf ein Listenelement zuzugreifen. Ansonsten aber alles ziemlich analog zu Arrays.

Natürlich kann man Lambda-Ausdrücke nicht nur in Listen, sondern auch in beliebige Collections einfügen. Das wird hier nicht extra gezeigt.

TECHNISCHE UNIVERSITÄT DARMSTADT

Sortieren mit Comparator

Wir haben jetzt alle Ingredienzen zusammen für ein absolut grundlegendes Beispiel für die Verwendung von Interface List, nämlich Sortieren, also die Umsortierung der Elemente in eine aufsteigende oder absteigende Reihenfolge nach einem beliebigem Kriterium. Für das Kriterium wird das Interface Comparator verwendet, das wir kurz vor Ende von Kapitel 06 schon als Fallbeispiel für Generics gesehen hatten.

Sortieren mit Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Student> list = new LinkedList<Student> ();

for ( int i = 0; i < 100000; i++ ) {
    Student s = new Student();
    s.enrollmentNumber = ( ( 5432 * i ) % 4321 );
    list.add ( s );
}

Collections.sort ( list, new EnrollmentNumberComparator() );
```

Wieder ein kleines, illustratives Beispiel. Dieses Beispiel reicht aus.

Sortieren mit Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Student> list = new LinkedList<Student> ();

for ( int i = 0; i < 100000; i++ ) {
    Student s = new Student();
    s.enrollmentNumber = ( ( 5432 * i ) % 4321 );
    list.add ( s );
}

Collections.sort ( list, new EnrollmentNumberComparator() );
```

Wir richten eine Liste der Klasse Student ein, die wir als Beispielmaterial für Comparator schon eingeführt hatten.

Sortieren mit Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Student> list = new LinkedList<Student> ();

for ( int i = 0; i < 100000; i++ ) {
    Student s = new Student();
    s.enrollmentNumber = ( ( 5432 * i ) % 4321 );
    list.add ( s );
}

Collections.sort ( list, new EnrollmentNumberComparator() );
```

Der dynamische Typ muss natürlich eine Klasse sein, die das Interface List implementiert.

***Vorgriff*: Wir realisieren gleich exemplarisch eine Klasse, die List implementiert und intern ziemlich genau so aussieht wie Klasse LinkedList intern aussehen dürfte.**

Sortieren mit Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Student> list = new LinkedList<Student> ();

for ( int i = 0; i < 100000; i++ ) {
    Student s = new Student();
    s.enrollmentNumber = ( ( 5432 * i ) % 4321 );
    list.add ( s );
}

Collections.sort ( list, new EnrollmentNumberComparator() );
```

Die Matrikelnummern werden zur Illustration ziemlich wild durcheinander gewählt.

Sortieren mit Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Student> list = new LinkedList<Student> ();

for ( int i = 0; i < 100000; i++ ) {
    Student s = new Student();
    s.enrollmentNumber = ( ( 5432 * i ) % 4321 );
    list.add ( s );
}

Collections.sort ( list, new EnrollmentNumberComparator() );
```

Und mit diesen Matrikelnummern werden die Student-Objekte dann in die Liste eingefügt. Vor- und Nachname sind für unser Beispiel irrelevant, weil darauf gar nicht zugegriffen wird, daher setzen wir Vor- und Nachname nicht.

Sortieren mit Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
List<Student> list = new LinkedList<Student> ();

for ( int i = 0; i < 100000; i++ ) {
    Student s = new Student();
    s.enrollmentNumber = ( ( 5432 * i ) % 4321 );
    list.add ( s );
}

Collections.sort ( list, new EnrollmentNumberComparator() );
```

Klasse Collections – mit s hinten – hat eine Klassenmethode namens sort, die die Liste – also den ersten Parameter der Methode – sortiert. Die Sortierlogik wird durch den zweiten Parameter definiert, der eine Comparator-Klasse mit demselben Typparamater wie die Liste sein muss. Wir wählen Klasse EnrollmentNumberComparator, also sind die Studierenden in der Liste nach Beendigung der Methode sort aufsteigend nach ihren Matrikelnummern sortiert – die kleinste Matrikelnummer zuerst, die größte als letztes.

TECHNISCHE UNIVERSITÄT DARMSTADT

Iterator

Bei der Übersicht über Klassen und Objekte haben wir das wichtige Thema Iteratoren schon kurz angesprochen. Jetzt behandeln wir das Thema im Detail.

Iterator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Iterator<Number>  it1 = c1.iterator ();
Iterator<Number>  it2 = c2.iterator ();
Iterator<Number>  it3 = c3.iterator ();

while ( it1.hasNext() )
    System.out.print ( it1.next().doubleValue() );
```

Wie immer ein kleines, illustratives Beispiel.

Iterator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Iterator<Number> it1 = c1.iterator ();
Iterator<Number> it2 = c2.iterator ();
Iterator<Number> it3 = c3.iterator ();

while ( it1.hasNext() )
    System.out.print ( it1.next().doubleValue() );
```

Zu jeder Klasse, die das Interface Collection implementiert, gibt es eine eigene Iterator-Klasse, die das generische Interface Iterator implementiert. Dieses Interface können wir daher zum formalen Parameter jedes Iterators machen.

***Nebenbemerkung*: Collection ist eine Erweiterung eines Interface namens Iterable. Alles in diesem Abschnitt über Interface Collection Gesagte ist eigentlich von Iterable ererbt.**

Iterator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Iterator<Number>  it1 = c1.iterator ();
Iterator<Number>  it2 = c2.iterator ();
Iterator<Number>  it3 = c3.iterator ();

while ( it1.hasNext() )
    System.out.print ( it1.next().doubleValue() );
```

Schon früher hatten wir c1 vom Typ ArrayList, c2 vomTyp HashSet und c3 vom Typ Vector beispielhaft eingerichtet.

Iterator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Iterator<Number>  it1 = c1.iterator ();
Iterator<Number>  it2 = c2.iterator ();
Iterator<Number>  it3 = c3.iterator ();

while ( it1.hasNext() )
    System.out.print ( it1.next().doubleValue() );
```

Das Interface Collection hat eine Methode iterator, die in Interface Iterable definiert und an Collection vererbt ist. Der Rückgabetyp ist Interface Iterator. Jede Klasse, die Iterable implementiert, liefert ein Objekt von ihrer eigenen spezifischen Iterator-Klasse zurück, aber wir können Iteratoren durch das Interface Iterator völlig gleich verwenden, die Unterschiede sind im dynamischen Typ sozusagen „versteckt“.

Iterator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Iterator<Number>  it1 = c1.iterator ();
Iterator<Number>  it2 = c2.iterator ();
Iterator<Number>  it3 = c3.iterator ();

while ( it1.hasNext() )
    System.out.print ( it1.next().doubleValue() );
```

Den ersten der drei Iteratoren verwenden wir jetzt beispielhaft nach dem typischen Muster, wie man generell mit Iteratoren umgeht.

Iterator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Iterator<Number>  it1 = c1.iterator ();
Iterator<Number>  it2 = c2.iterator ();
Iterator<Number>  it3 = c3.iterator ();

while ( it1.hasNext() )
    System.out.print ( it1.next().doubleValue() );
```

Methode next liefert ein bislang von diesem Iterator noch nicht geliefertes Element der Collection. Die Reihenfolge, in der die Elemente durch Aufrufe von next zurückgeliefert werden, ist bei Klassen, die Interface Collection implementieren, generell unbestimmt. Speziell bei Klassen, die Interface List implementieren, besteht aber die Verpflichtung, dass die Elemente streng nach Index durchlaufen werden, zuerst das Element an Index 0, dann das an Index 1 und so weiter.

Iterator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Iterator<Number>  it1 = c1.iterator ();
Iterator<Number>  it2 = c2.iterator ();
Iterator<Number>  it3 = c3.iterator ();

while ( it1.hasNext() )
    System.out.print ( it1.next().doubleValue() );
```

Der Elementtyp der Collection c1 ist ja Number. Daher haben alle Elemente die Methode doubleValue von Klasse Number.

Iterator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Iterator<Number>  it1 = c1.iterator ();
Iterator<Number>  it2 = c2.iterator ();
Iterator<Number>  it3 = c3.iterator ();

while ( it1.hasNext() )
    System.out.print ( it1.next().doubleValue() );
```

Methode hasNext ist boolesch. Der Rückgabewert ist genau dann true, wenn mindestens ein Element der Collection noch nicht durch diesen Iterator zurückgeliefert wurde. Betonung liegt auf *diesem* Iterator; wir können auch mehrere Iteratoren nacheinander oder sogar gleichzeitig auf derselben Collection haben.

Iterator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while ( it1.hasNext() )
    System.out.print ( it1.next().doubleValue() );
while ( it2.hasNext() )
    System.out.print ( it2.next().doubleValue() );
while ( it3.hasNext() )
    System.out.print ( it3.next().doubleValue() );
```

Hier sind noch einmal dieselben zwei Java-Zeilen von der letzten Folie.

Iterator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while ( it1.hasNext() )
    System.out.print ( it1.next().doubleValue() );
while ( it2.hasNext() )
    System.out.print ( it2.next().doubleValue() );
while ( it3.hasNext() )
    System.out.print ( it3.next().doubleValue() );
```

Völlig identisch der Durchlauf durch die zweite Collection mit dem zweiten Iterator.

Iterator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while ( it1.hasNext() )
    System.out.print ( it1.next().doubleValue() );
while ( it2.hasNext() )
    System.out.print ( it2.next().doubleValue() );
while ( it3.hasNext() )
    System.out.print ( it3.next().doubleValue() );
```

Und natürlich identisch auch die dritte Collection und der dritte Iterator.

TECHNISCHE UNIVERSITÄT DARMSTADT

Kurzform for-Schleife

Bei Arrays hatten wir eine Kurzform der for-Schleife kennengelernt für den Standardfall, dass man alle Komponenten nach ihrem Index durchläuft. Eine solche Kurzform gibt es auch für das Interface Iterable. Das Interface Iterable nimmt daher eine Sonderrolle in Java ein, indem diese spezielle Schreibweise bereitgestellt wird. Diese Sonderrolle erhalten dann natürlich auch alle von Iterable erbenden Interfaces wie Collection und List sowie alle Klassen, die eines dieser Interfaces implementieren.

Kurzform for-Schleife

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void writeAllElements
                    ( Collection<String> coll ) {
  for ( String str : coll )
      System.out.print ( str );
}
```

Dazu wieder ein kleines, rein illustrativ gemeintes Beispiel: Alle Elemente einer Collection sollen in der Reihenfolge, die von der Collection-Klasse vorgegeben wird, auf dem Bildschirm ausgegeben werden.

Kurzform for-Schleife

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void writeAllElements
                    ( Collection<String> coll ) {
  for ( String str : coll )
      System.out.print ( str );
}
```

Hier ist diese verkürzte for-Schleife. Beachten Sie die strikte Analogie zur verkürzten for-Schleife bei Arrays.

Kurzform for-Schleife

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void writeAllElements
                    ( Collection<String> coll ) {
  for ( String str : coll )
      System.out.print ( str );
}
```

Eine Variable vom Elementtyp der Collection enthält nacheinander Verweise auf die einzelnen String-Objekte, die in der Collection momentan gespeichert sind.

Kurzform for-Schleife

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void writeAllElements
                    ( Collection<String> coll ) {
  for ( String str : coll )
      System.out.print ( str );
}
```

Und nach dem Doppelpunkt kommt dann der Name der Collection.

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Standardoperationen filter, map und fold auf Java-Collection

In Kapitel 04c hatten wir diese drei Standardoperationen schon bei Racket auf Listen gesehen. Zum Vergleich sehen wir uns diese drei Operationen nun bei Java-Collections an.

Standardoperation filter

TECHNISCHE UNIVERSITÄT DARMSTADT

Objektmethode von Collection<T>:

boolean removeIf (Predicate<? super T> pred)

Zuerst filter: Für diese Operation hat Interface Collection tatsächlich schon eine Methode, die sehen Sie hier. Alle Elemente der Collection, die nicht das Prädikat passieren, werden aus der Collection entfernt. Das entspricht nicht genau der filter-Operation in Racket, denn dort wird eine neue Liste zurückgeliefert, nicht die Eingabeliste geändert. Zurückgeliefert wird von removeIf, ob mindestens ein Element aus der Liste entfernt wurde oder nicht.

***Vorgriff*: Für die anderen beiden Operationen, map und fold, hat das Interface Collection keine Methoden. Wir werden aber in Kapitel 08 sehen, dass auf Collection ein anderer Mechanismus möglich ist, nämlich Streams, der für alle drei Standardoperationen jeweils eine Methode bereitstellt.**

Standardoperation filter

TECHNISCHE UNIVERSITÄT DARMSTADT

Objektmethode von Collection<T>:

boolean removeIf (Predicate<? super T> pred)

Erinnerung an Kapitel 06, Abschnitt zu Wildcards: Ein Prädikat auf einem Supertyp von T kann ja problemlos auf Objekte vom Typ T oder Subtypen von T angewandt werden.

Standardoperation filter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void filter
                    ( Collection<T> coll, Predicate<T> pred ) {
  Iterator it = coll.iterator();
  while ( it.hasNext() ) {
     if ( ! pred.test ( it.next() )
       it.remove ();
}
```

Hier sehen wir eine mögliche Implementation der filter-Operation mittels Iterator, die wie Methode removeIf die Eingabeliste selbst verändert. Zur Vereinfachung ist die Methode void im Gegensatz zu removeIf. Als kleine Übung können Sie diese Methode so abändern, dass sie wie removeIf zurückliefert, ob Elemente entfernt wurden.

Standardoperation filter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <T> void filter
                ( Collection<T> coll, Predicate<T> pred ) {
  Iterator it = coll.iterator();
  while ( it.hasNext() ) {
     if ( ! pred.test ( it.next() )
      it.remove ();
}
```

Eine wichtige Feinheit hatten wir im Abschnitt zu Interface Iterator noch ausgelassen; hier brauchen wir sie und tragen sie deshalb nach: Wenn man mit einem Iterator über eine Collection läuft, dann darf diese Collection währenddessen eigentlich nicht geändert werden. Natürlich darf schon gar nicht das Element aus der Collection entfernt werden, auf dem der Iterator gerade steht.

Da aber genau dies so wie hier sehr häufig der Zweck dafür ist, dass man durch eine Collection läuft, wurde entschieden, dem Interface Iterator noch eine default-Methode namens remove mitzugeben, die das letzte mit Methode next zurückgelieferte Element aus der Collection entfernt. Nach einem Aufruf der Methode next darf daher nur einmal Methode remove aufgerufen werden, vor dem nächsten Aufruf von remove muss noch einmal ein next kommen.

Standardoperation filter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static  <T> Collection<T> filter
                ( Collection<T> coll, Predicate<T> pred ) {
  Collection<T> result = new LinkedList<T>();
  for ( T t : coll )
     if ( pred.test ( t ) )
        result.add ( t );
  return result;
}
```

Nun eine Variation der Methode filter, in der wie bei Racket nicht die Eingabeliste geändert, sondern eine davon separate Ergebnisliste erstellt wird. Daher wird eine Collection vom selben generischen Typ zurückgeliefert. Wenn die Dokumentation der Methode keine einschränkenden Forderungen stellt, kann man die implementierende Klasse frei wählen; wir wählen hier LinkedList.

Standardoperation filter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static  <T> Collection<T> filter
                    ( Collection<T> coll, Predicate<T> pred ) {
  Collection<T> result = new LinkedList<T>();
  for ( T t : coll )
     if ( pred.test ( t ) )
        result.add ( t );
  return result;
}
```

Da die Liste, über die der Iterator läuft, nicht geändert werden soll und daher Methode remove nicht gebraucht wird, kann auch einfach die verkürzte for-Schleife verwendet werden.

Standardoperation map

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <X,Y> Collection<Y> map
                ( Collection<X> coll, Function<X,Y> fct ) {
   LinkedList<Y> result = new LinkedList<Y>();
   for ( X x : coll )
      result.add ( fct.apply ( x ) );
   return result;
}
```

Mit dem soeben Gesagten zu filter und mit der Beschreibung von map in Kapitel 04c sollte diese Implementation einer Methode map in Java eigentlich problemlos zu verstehen sein. Da die beiden Listen prinzipiell von verschiedenen generischen Typen sind, stellt sich die Frage gar nicht erst, ob wie bei filter auch die Eingabeliste geändert werden kann; es geht eben nur so, dass eine neue Liste erstellt wird.

Standardoperation fold

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static <X,Y> Y fold
       ( Y init, BiFunction<X,Y,Y> fct, Collection<X> coll ) {
   Y tmp = init;
   for ( X x : coll )
      tmp = fct.apply ( x, tmp );
   return tmp;
}
```

Und abschließend noch eine Implementation von fold, die ebenfalls leicht zu verstehen sein sollte.

TECHNISCHE UNIVERSITÄT DARMSTADT

Maps

Noch eine wichtige Erweiterung des Konzepts von Elementesammlungen, genannt Map, was unter anderem *Abbildung* heißt, auch mathematische Abbildung, also Funktion in Sinne der Mathematik. Und genau darum geht es, wie wir jetzt sehen werden. Auch Interface Map findet sich im Package java.util. Natürlich darf das Interface Map nicht verwechselt werden mit der Standardoperation map, die wir eben gesehen hatten.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {

    private int n;

    public X ( int n ) {
        this.n = n;
    }

    public int getN () {
        return n;
    }
}
```

Wir benutzen dazu eine künstliche, rein illustrative Beispielklasse X mit einem int-Attribut n, das im Konstruktor initialisiert und von der zugehörigen get-Methode zurückgeliefert wird.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Map<String,X> map = new HashMap<String,X> ();
for ( int i = 0; i < 100; i++ ) {
   String key = “Nr. “ + i;
   X value = new X ( 2 * i + 3 );
   map.put ( key, value );
}
```

Jetzt das konkrete Beispiel für die Verwendung von Maps auf Basis der Beispielklasse X.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Map<String,X> map = new HashMap<String,X> ();
for ( int i = 0; i < 100; i++ ) {
    String key = "Nr. " + i;
    X value = new X ( 2 * i + 3 );
    map.put ( key, value );
}
```

Das Interface Map hat zwei Typparameter. Der erste ist der Schlüssel oder Englisch *Key*. Der zweite ist die Information, die zum Key gespeichert wird. Der englische Begriff für diese Information ist meist *Value*. Eine Map realisiert also eine Abbildung von den Keys in die Values.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Map<String,X> map = new HashMap<String,X> ();
for ( int i = 0; i < 100; i++ ) {
   String key = "Nr. " + i;
   X value = new X ( 2 * i + 3 );
   map.put ( key, value );
}
```

Der dynamische Typ ist natürlich wieder eine implementierende Klasse, auf die wir hier aber nicht weiter eingehen.

Vorgriff: Hashtabellen lernen Sie im zweiten Fachsemester in der Lehrveranstaltung Algorithmen und Datenstrukturen kennen.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Map<String,X> map = new HashMap<String,X> ();
for ( int i = 0; i < 100; i++ ) {
    String key = "Nr. " + i;
    X value = new X ( 2 * i + 3 );
    map.put ( key, value );
}
```

In unsere Map fügen wir nun zur Illustration einhundert Key-Value-Paare ein. Die Keys in einer Map müssen alle unterschiedlich sein. Wie es in der Mathematik für Abbildungen gefordert ist, wird jedem Key genau ein Wert zugewiesen. Die Values müssen *nicht* unterschiedlich sein; zwei Keys dürfen durchaus denselben Value haben, was mathematisch gesprochen nichts anderes heißt, als dass die Abbildung nicht injektiv sein muss.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Map<String,X> map = new HashMap<String,X> ();
for ( int i = 0; i < 100; i++ ) {
  String key = "Nr. " + i;
  X value = new X ( 2 * i + 3 );
  map.put ( key, value );
}
```

Der Einfachheit halber bilden wir die Keys auf einheitliche Weise aus den Zahlen 0 bis 99. Wie gesagt, kein Key darf zweimal vorkommen, das ist durch diese künstliche, nur beispielhaft gemeinte Bildungsweise gewährleistet.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Map<String,X> map = new HashMap<String,X> ();
for ( int i = 0; i < 100; i++ ) {
   String key = "Nr. " + i;
   X value = new X ( 2 * i + 3 );
   map.put ( key, value );
}
```

Der Value-Typ hat ja ein int-Attribut namens n. Dieser Wert wird ebenfalls der Einfachheit halber auf einheitliche Weise aus den Zahlen 0 bis 99 gebildet.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Map<String,X> map = new HashMap<String,X> ();
for ( int i = 0; i < 100; i++ ) {
   String key = “Nr. “ + i;
   X value = new X ( 2 * i + 3 );
   map.put ( key, value );
}
```

Mit Methode put von Interface Map fügen wir dieses Paar ein.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str1 = new String ( “Nr. 23“ );
X x1 = map.get ( str1 );
System.out.print ( x1.getN() );
String str2 = new String ( “Hallo“ );
X x2 = map.get ( str2 );
if ( x2 == null )
   System.out.print ( “Wie erwartet“ );
```

Und jetzt greifen wir einmal auf einen vorher eingefügten und einmal auf einen bis dato *nicht* eingefügten Key zu, um zu sehen, was dann jeweils passiert.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str1 = new String ( “Nr. 23“ );
X x1 = map.get ( str1 );
System.out.print ( x1.getN() );
String str2 = new String ( “Hallo“ );
X x2 = map.get ( str2 );
if ( x2 == null )
   System.out.print ( “Wie erwartet“ );
```

Diesen Key hatten wir eingefügt.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str1 = new String ( “Nr. 23“ );
X x1 = map.get ( str1 );
System.out.print ( x1.getN() );
String str2 = new String ( “Hallo“ );
X x2 = map.get ( str2 );
if ( x2 == null )
   System.out.print ( “Wie erwartet“ );
```

Das hat zur Konsequenz, dass Methode get den zugehörigen Value zurückliefert.

***Nebenbemerkung*: Ein Wörterbuch oder Lexikon würde man genau so wie auf dieser Folie gezeigt realisieren, auch mit HashMap oder ähnlicher Klasse. Ebenso eine Zugriffsmöglichkeit, bei der man in einem Studierendenverwaltungssystem eine Matrikelnummer eingibt, um die zugehörigen Stammdaten abzurufen, oder in einem Kontenverwaltungssystem eine Kontonummer eingibt, um die zugehörigen Kontodaten abzurufen. Die Suchwörter sind bei einem Wörterbuch die Keys, also String. Matrikelnummer und Kontonummer als Keys könnte man ebenfalls als Strings oder auch als int realisieren, es hängt von den konkreten Umständen ab, was sinnvoller ist. In einem Wörterbuch wäre der Value-Typ irgendeine Art von Texttyp, bei Studierenden die Klasse Student von vorher, natürlich um etliche weitere personenspezifische Attribute angereichert, analog Kontodaten.**

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str1 = new String ( “Nr. 23“ );
X x1 = map.get ( str1 );
System.out.print ( x1.getN() );          // 49
String str2 = new String ( “Hallo“ );
X x2 = map.get ( str2 );
if ( x2 == null )
   System.out.print ( “Wie erwartet“ );
```

Tatsächlich kommt die Zahl heraus, die wir – eingepackt in ein X-Objekt – zu str1 gespeichert hatten.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str1 = new String ( “Nr. 23“ );
X x1 = map.get ( str1 );
System.out.print ( x1.getN() );              //  49
String str2 = new String ( “Hallo“ );
X x2 = map.get ( str2 );
if ( x2 == null )
   System.out.print ( “Wie erwartet“ );
```

Nun noch zum Vergleich ein String, den wir definitiv *nicht* als Key in die Map eingefügt hatten.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str1 = new String ( “Nr. 23“ );
X x1 = map.get ( str1 );
System.out.print ( x1.getN() );              //  49
String str2 = new String ( “Hallo“ );
X x2 = map.get ( str2 );
if ( x2 == null )
    System.out.print ( “Wie erwartet“ );
```

Auch hier probieren wir mit Methode get, einen Value zu diesem Key von der Map zurückzubekommen.

Maps

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str1 = new String ( “Nr. 23“ );
X x1 = map.get ( str1 );
System.out.print ( x1.getN() );              //  49
String str2 = new String ( “Hallo“ );
X x2 = map.get ( str2 );
if ( x2 == null )
   System.out.print ( “Wie erwartet“ );
```

Aber stattdessen bekommen wir den Wert null, mit dem angezeigt wird, dass der aktuale Parameter von get *nicht* als Key in der Map zu finden ist.

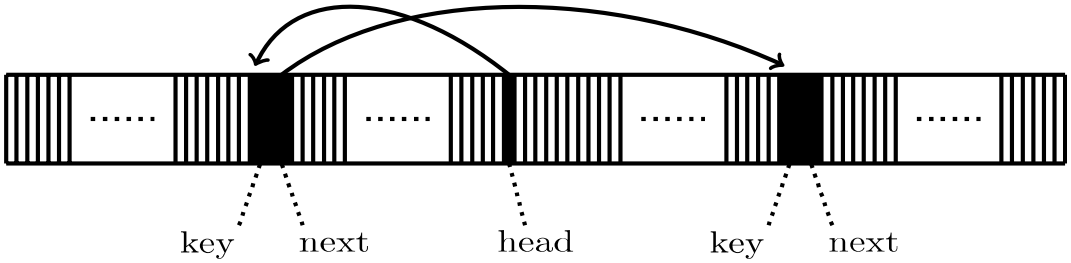
TECHNISCHE UNIVERSITÄT DARMSTADT

Eigene LinkedList-Klasse

Wir haben schon gesehen, dass es eine Implementation von List namens LinkedList im Package java.util gibt. Um zu studieren, wie generell Klassen aussehen, die Collection oder List implementieren, definieren wir als nächstes eine eigene Klasse MyLinkedList, die bis auf Feinheiten mehr oder weniger genauso aussieht wie die Implementation von java.util.LinkedList.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

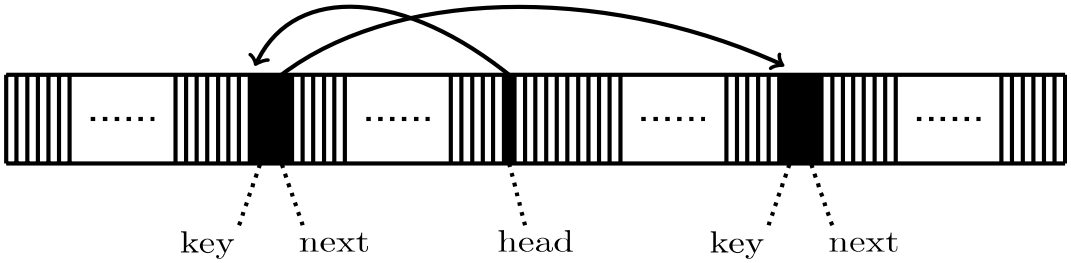



```
public class ListItem <T> {
  public T key;
  public ListItem<T> next;
}
```

```
ListItem<T> head = new ListItem<T>();
head.next = new ListItem<T>();
```

Wir betrachten hier eine wichtige Konstellation: Eine Klasse kann Referenzen auf die eigene Klasse als Attribute haben. Das geht dank der Trennung in Referenz und eigentlichem Objekt. Gäbe es diese Trennung nicht – wäre also das Attribut das Objekt selbst und nicht nur ein Verweis darauf –, dann würde ein Objekt der Klasse ein anderes Objekt derselben Klasse enthalten. Das würde aber wiederum ein Objekt derselben Klasse enthalten, das wiederum, und so weiter, ad infinitum.

Eigene LinkedList-Klasse

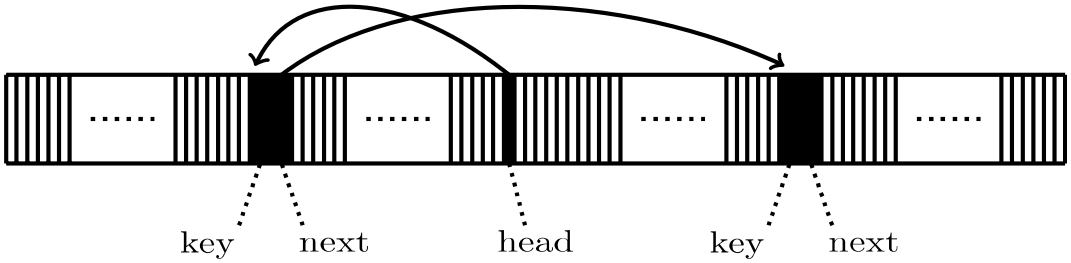



```
public class ListItem <T> {
  public T key;
  public ListItem<T> next;
}
```

```
ListItem<T> head = new ListItem<T>();
head.next = new ListItem<T>();
```

Die Klasse ListItem ist zweckmäßigerweise generisch, so wie die Listen-Klassen in java.util. Das eigentliche Listenelement wird typischerweise als Key bezeichnet.

Eigene LinkedList-Klasse



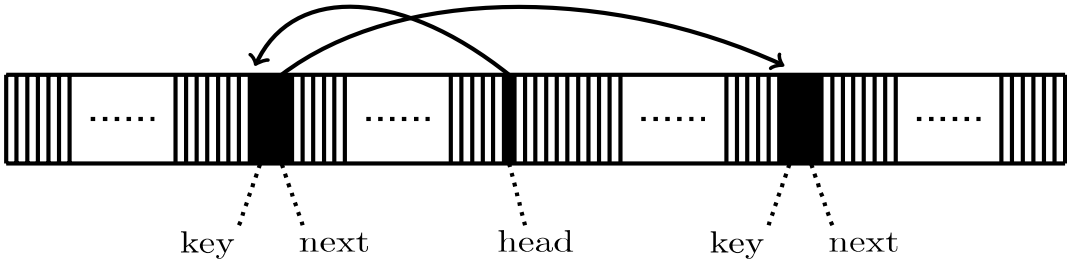

```
public class ListItem <T> {
  public T key;
  public ListItem<T> next;
}
```

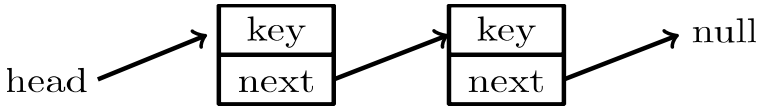
```
ListItem<T> head = new ListItem<T>();
head.next = new ListItem<T>();
```

Ein Objekt von Klasse ListItem hat ein Attribut von Klasse ListItem. Mit diesem zweiten Attribut werden die Listenelemente miteinander verkettet.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

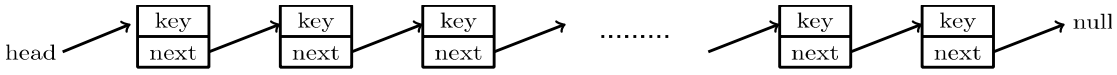




Wir lösen uns von der starren Darstellung im Speicher und betrachten Bilder in der Form wie unten. Der Inhalt unten ist derselbe wie der oben, nur übersichtlicher.

Eigene LinkedList-Klasse

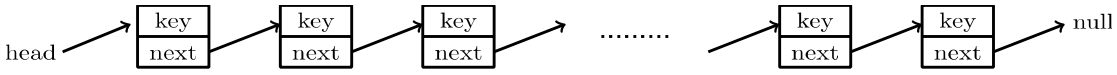
TECHNISCHE UNIVERSITÄT DARMSTADT



Wir können noch einen Schritt weitergehen und mehr als zwei Listenelemente miteinander verketten, sogar beliebig viele, tausende, millionen, milliarden... Genau das ist das Konzept verzeigerte Liste oder englisch linked list. Und bis auf Details sieht auch die Klasse LinkedList intern, im private-Bereich, genau so aus.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

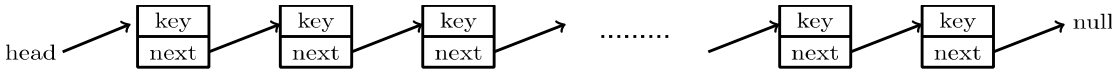



```
public class MyLinkedList <T> implements List<T> {
    private ListItem<T>  head;

    .....
}
```

So wie unten sieht dann unsere selbstgebastelte Klasse aus.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT



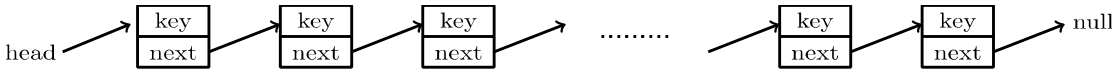

```
public class MyLinkedList <T> implements List<T> {
   private ListItem<T>  head;

   .........
}
```

Der Verweis auf den Kopf der Liste, also auf das erste Listenelement, ist dann natürlich ein private-Attribut. Direkter Umgang mit einer solchen verzeigerten Struktur ist zu fehleranfällig, da soll der Nutzer der Klasse gar nicht erst in Versuchung geführt werden, selbst auf diese Struktur zuzugreifen.

Durch diese Einkapselung in der Klasse MyLinkedList ist es auch kein Problem, dass die Attribute von Klasse ListItem public sind.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT




```
public class MyLinkedList <T> implements List<T> {
   private ListItem<T>  head;

   .....
}
```

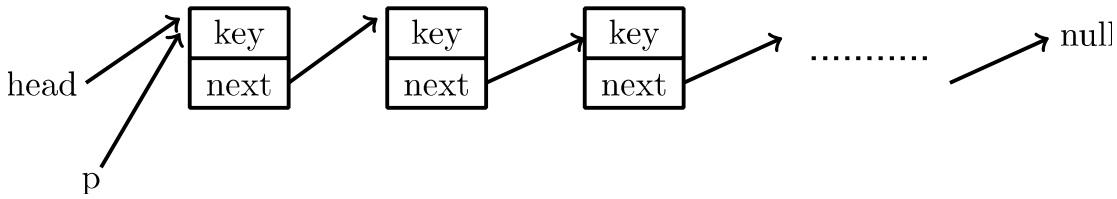
Es macht durchaus Sinn, weitere Attribute einzufügen, zum Beispiel ein int-Attribut, in dem die momentane Größe der Liste, also die momentane Anzahl Listenelemente gespeichert ist. Aber notwendig sind solche zusätzlichen Attribute nicht, und wir definieren in dieser einfachen Klasse daher auch keine weiteren.

Wir haben aber public-Methoden zu definieren. Wir werden allerdings nicht alle Methoden definieren, die Klasse LinkedList aus java.util hat, nur ein paar ausgewählte, und die auch nur vereinfacht: Die ganze Exception-Werferei lassen wir weitgehend aus, um uns auf die für diesen Abschnitt wesentlichen Aspekte zu konzentrieren.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Durchlauf durch *alle* Elemente:



ListItem<T> p = head;

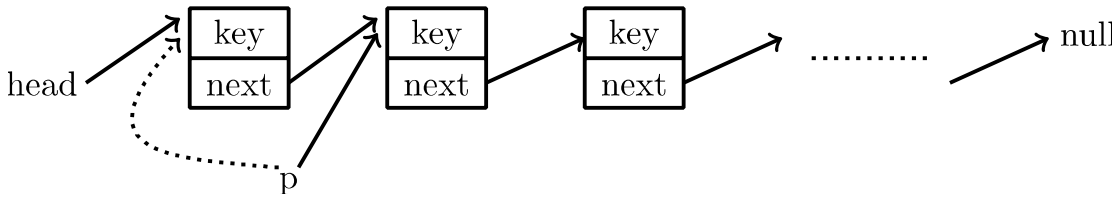
Das grundlegende Zugriffsmuster bei einer Liste ist der Durchlauf durch die Listenelemente − entweder bis zu einem bestimmten Element oder Durchlauf durch alle Elemente. Wir sehen uns zuerst den Fall an, dass die ganze Liste durchlaufen wird.

Dazu brauchen wir eine Laufvariable, so wie der Laufindex beim Array. Wir sprechen von Lauf*pointer* und nennen ihn einfach p. Er ist vom Typ der Listenelemente, denn er soll nacheinander auf die einzelnen Elemente der Liste verweisen.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Durchlauf durch *alle* Elemente:



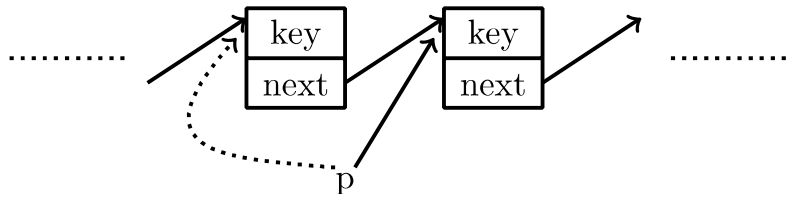
p = p.next;

Jeder Vorwärtsschritt sieht gleich aus: Der Wert in Attribut next des ersten Listenelements ist die Adresse des zweiten Listenelements. Wenn der Laufpointer p nun auf den Wert von p.next gesetzt wird, dann verweist p nicht mehr auf das erste, sondern auf das zweite Element.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Durchlauf durch *alle* Elemente:



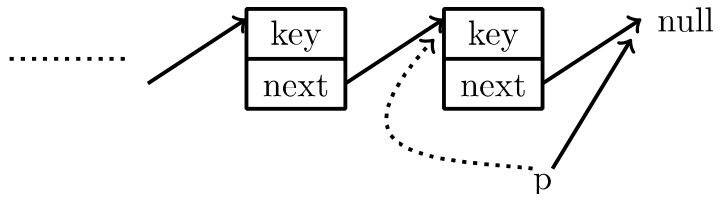
p = p.next;

Natürlich sieht der Vorwärtsschritt an späteren Positionen in der Liste immer völlig identisch aus: p wird auf p.next gesetzt.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Durchlauf durch *alle* Elemente:



p = p.next;

Und im allerletzten Schritt des Durchlaufs wird p zwangsläufig auf null gesetzt. Damit haben wir auch das Abbruchkriterium, nämlich dass p gleich null ist.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void processAll ( Consumer<T> consumer ) {
    for ( ListItem<T> p = head; p != null; p = p.next )
        consumer.accept ( p.key );
}
```

Klasse LinkedList hat die hier gezeigte Methode nicht, aber wir definieren sie trotzdem für MyLinkedList.

***Vorgriff*: Im Kapite 08l zu Streams und Files werden wir sehen, dass in Klasse LinkedList eine elegantere Methode implementiert ist, um alle Elemente einer Liste in einem Rutsch zu prozessieren.**

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void processAll ( Consumer<T> consumer ) {
    for ( ListItem<T> p = head; p != null; p = p.next )
        consumer.accept ( p.key );
}
```

Das generische Functional Interface Consumer aus java.util.function bietet mit der funktionalen Methode accept eine einheitliche Schnittstelle für den Fall, dass man Werte von T prozessiert, ohne dass ein Rückgabewert herauskommt.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void processAll ( Consumer<T> consumer ) {
    for ( ListItem<T> p = head; p != null; p = p.next )
        consumer.accept ( p.key );
}
```

Die for-Schleife zum Durchlauf durch die Liste demonstriert die Flexibilität des Konzepts for-Schleife. Sie ist hier in vielerlei Hinsicht ähnlich zur for-Schleife zum Durchlauf eines Arrays.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void processAll ( Consumer<T> consumer ) {
    for ( ListItem<T> p = head; p != null; p = p.next )
        consumer.accept ( p.key );
}
```

Dort, wo beim Durchlauf durch ein Array der Laufindex mit 0, also mit dem Index der ersten Arraykomponente initialisiert wird, wird hier der Lauf*pointer* mit der Adresse des ersten Listenelements initialisiert.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void processAll ( Consumer<T> consumer ) {
    for ( ListItem<T> p = head; p != null; p = p.next )
        consumer.accept ( p.key );
}
```

Und an der Stelle, an der beim Durchlauf durch ein Array der Laufindex inkrementiert wird, also von einer Arraykomponente zur nächsten gegangen wird, wird hier der Laufpointer von einem Listenelement zum nächsten umgesetzt.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void processAll ( Consumer<T> consumer ) {
    for ( ListItem<T> p = head; p != null; p = p.next )
        consumer.accept ( p.key );
}
```

Schließlich ist auch die Fortsetzungsbedingung analog zum Arraydurchlauf: Die Schleife wird beendet, sobald alle Elemente durchlaufen sind.

Beachten Sie, dass die Methode processAll auch im Fall einer leeren Liste das Richtige tut, denn dann ist die Fortsetzungsbedingung schon beim ersten Test nicht erfüllt, die Schleife wird kein einziges Mal durchlaufen, kein Element wird prozessiert, und das ist ja bei einer Liste ohne Elemente auch genau richtig.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class ScreenNumberWriter
                        implements Consumer<Number> {
    public void accept ( Number n ) {
        System.out.print ( n.doubleValue() );
    }
}
```

Um das Beispiel zu vervollständigen, noch eine beispielhafte Implementation von Consumer, das sicherlic nicht erklärt werden muss.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean contains ( T t ) {
    for ( ListItem<T> p = head; p != null; p = p.next )
        if ( p.key.equals(t) )
            return true;
    return false;
}
```

Jetzt eine Variation von processAll, die im Prinzip auch für List und sogar schon für Collection definiert ist. Wie schon gesagt, lassen wir das Thema Exceptions zur Vereinfachung hier aus. In dieser Variation wird die Liste nicht unbedingt bis zum Ende durchlaufen im Gegensatz zu processAll im letzten Beispiel.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean contains ( T t ) {
    for ( ListItem<T> p = head; p != null; p = p.next )
        if ( p.key.equals(t) )
            return true;
    return false;
}
```

Wir können den Kopf der for-Schleife exakt genauso wie soeben bei processAll gestalten.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean contains ( T t ) {
    for ( ListItem<T> p = head; p != null; p = p.next )
        if ( p.key.equals(t) )
            return true;
    return false;
}
```

Aber sobald das gesuchte Element gefunden ist, wird die Schleife und auch die gesamte Methode beendet, und korrekterweise wird true zurückgeliefert.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean contains ( T t ) {
    for ( ListItem<T> p = head; p != null; p = p.next )
        if ( p.key.equals(t) )
            return true;
    return false;
}
```

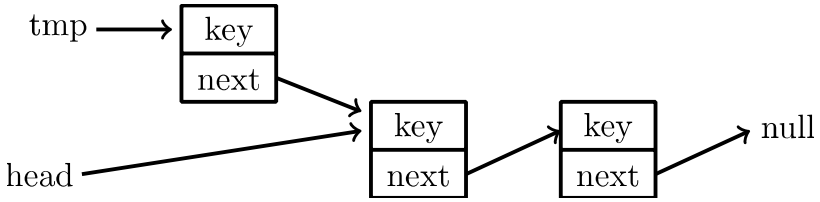
Natürlich kann es auch sein, dass das gesuchte Element gar nicht in der Liste ist. Dann wird die Schleife ohne return beendet, und die return-Anweisung *nach* der Schleife wird ausgeführt. Auch in diesem Fall ist die Rückgabe korrekt.

Beachten Sie, dass bei einer leeren Liste garantiert false herauskommt, weil die for-Schleife kein einziges Mal durchlaufen wird. Rückgabe false ist bei einer leeren Liste natürlich immer korrekt.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Neues Element einfügen:



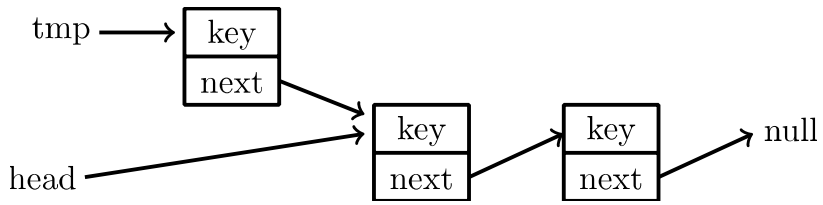

```
ListItem<T> tmp = new ListItem<T>();
tmp.key = key;
tmp.next = head;
```

Als nächstes die Methode add. Zuerst betrachten wir den einfacheren Fall, dass das neue Element ganze vorne an die Liste angehängt werden soll, das heißt, es soll nach dem Einfügen an Index 0 sein.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Neues Element einfügen:




```
ListItem<T> tmp = new ListItem<T>();
tmp.key = key;
tmp.next = head;
```

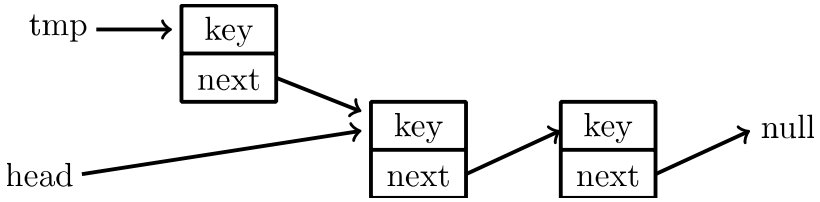
Für einen neuen Schlüsselwert in der Liste brauchen wir natürlich ein neues Listenelement.

***Achtung*: Im Abschnitt zu Einschränkungen der Generizität in Java weiter vorne im Kapitel hatten wir gesagt, dass keine Objekte des Typparameters erzeugt werden dürfen. Das ist an dieser Stelle auch nicht der Fall: Es wird kein T, sondern ein *ListItem* von T erzeugt.**

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Neues Element einfügen:



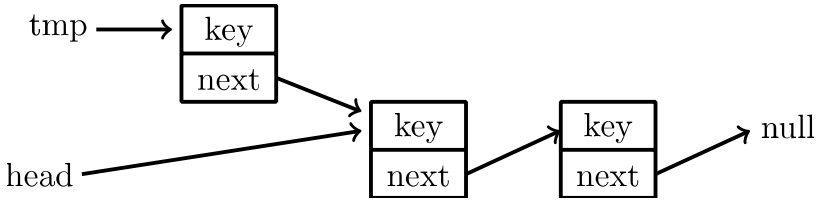

```
ListItem<T> tmp = new ListItem<T>();
tmp.key = key;
tmp.next = head;
```

Und das Listenelement bekommt den einzufügenden Schlüssel als Attribut key.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Neues Element einfügen:



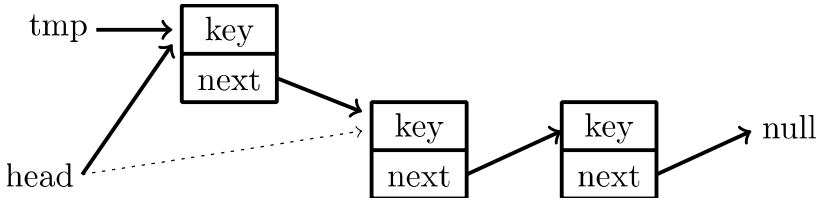

```
ListItem<T> tmp = new ListItem<T>();
tmp.key = key;
tmp.next = head;
```

Das Einfügen des neuen Elements am Anfang der Liste erfordert zwei Schritte, die unbedingt in dieser Reihenfolge auszuführen sind. Als erstes lassen wir das neue Element auf den bisherigen Listenkppf verweisen.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Neues Element einfügen:



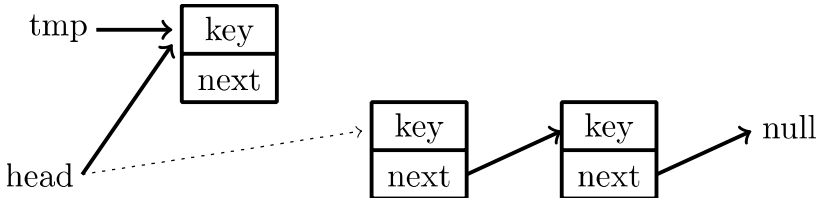

```
tmp.next = head;
head = tmp;
```

Und erst dann wird head auf den neuen Listenkopf gesetzt.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Neues Element einfügen:



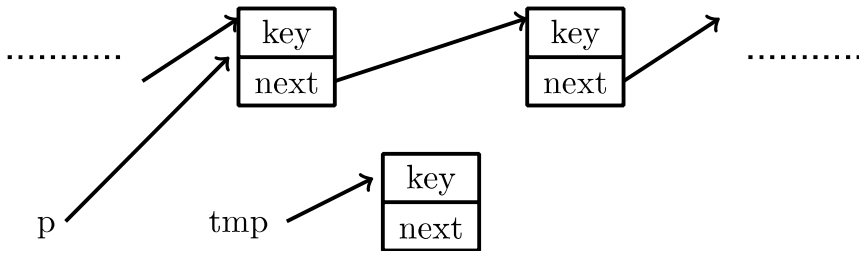

```
ListItem<T> tmp = new ListItem<T>();
tmp.key = key;
head = tmp;
```

Warum ist es wichtig, die Reihenfolge der beiden Schritte einzuhalten: Nun, wenn man zuerst den zweiten Schritt macht, überschreibt man den *einzigen* Verweis auf den ursprünglichen Listenkopf. Dieses Objekt und somit die gesamte bisherige Liste sind vom Programm aus nicht mehr erreichbar.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Neues Element einfügen:



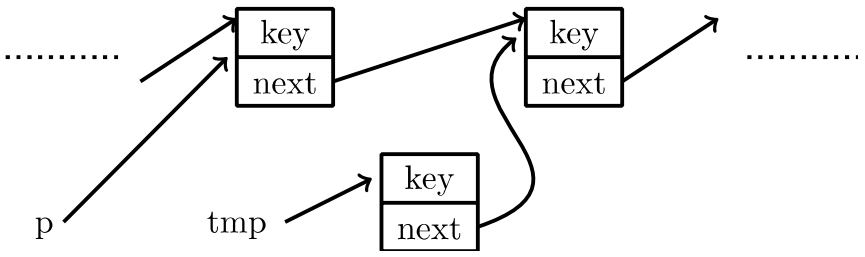

```
ListItem<T> tmp = new ListItem<T>();
tmp.key = key;
```

Jetzt fügen wir statt dessen ein Element irgendwo mitten in der Liste ein. Dazu müssen wir einen Laufpointer p wie gehabt einrichten und dann soweit in der Liste fortschalten, bis er auf dem Element steht unmittelbar vor der Stelle, an der das neue Element einzufügen ist. Also: Wenn das neue Element die Position n bekommen soll, dann muss p auf der Position n – 1 stehen.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Neues Element einfügen:



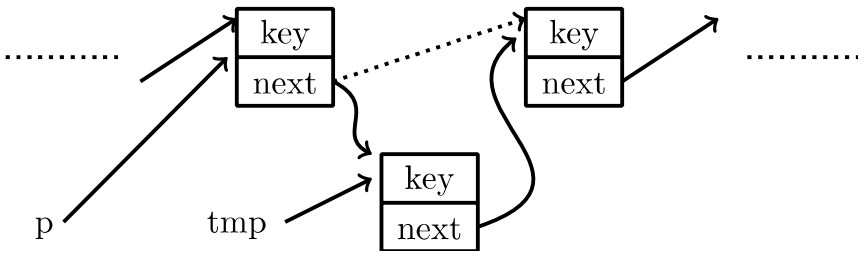
tmp.next = p.next;

Die beiden Schritte sind völlig analog zum Einfügen am Listenanfang. Zuerst wird das next-Attribut des neuen Listenelements auf die Adresse desjenigen Objekts gesetzt, das dem neuen Listenelement nach dem Einfügen unmittelbar folgen soll.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Neues Element einfügen:



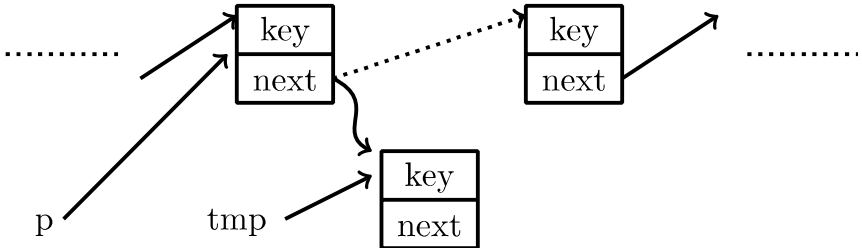
p.next = tmp;

Und erst als zweites wird dann der Verweis, der auf das neue Listenelement zeigen soll, gesetzt.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Neues Element einfügen:



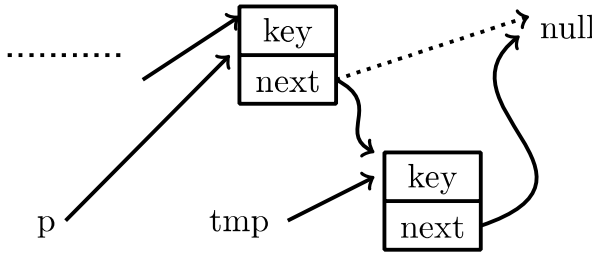

```
ListItem<T> tmp = new ListItem<T>();
tmp.key = key;
p.next = tmp;
```

Auch hier ist es aus demselben Grund wieder wichtig, den zweiten Schritt nicht vor dem ersten zu tun: Die Restliste würde wieder verlorengehen.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Neues Element einfügen:




```
tmp.next = p.next;
p.next = tmp;
```

Beachten Sie, dass der Fall, dass das neue Listenelement ganz am Ende anzuhängen ist, nicht gesondert betrachtet werden muss. Dieselben beiden Schritte funktionieren auch hier; dafür ist es völlig egal, ob p.next auf ein Objekt verweist oder den symbolischen Wert null hat.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void add ( int pos, T key ) throws IndexOutOfBoundsException {
  if ( pos < 0 )
    throw new IndexOutOfBoundsException ( "Wrong position: "+pos );
  ListItem<T> tmp = new ListItem<T>();
  tmp.key = key;
  .........
}
```

Jetzt haben wir uns bildlich klargemacht, wie das Einfügen eines neuen Elements in eine Liste aussehen müsste, und können uns an die Implementation der Methode add machen.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void add ( int pos, T key ) throws IndexOutOfBoundsException {
  if ( pos < 0 )
    throw new IndexOutOfBoundsException ( "Wrong position: "+pos );
  ListItem<T> tmp = new ListItem<T>();
  tmp.key = key;
  .........
}
```

Wie die entsprechende Methode von java.util.List bekommt unsere selbstgebastelte Methode als ersten Parameter die Position, an der das neue Element nach dem Einfügen stehen soll, und als zweiten Parameter den einzufügenden Wert selbst. Der Einfachheit halber ist unsere add-Methode void.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void add ( int pos, T key ) throws IndexOutOfBoundsException {
  if ( pos < 0 )
    throw new IndexOutOfBoundsException ( "Wrong position: "+pos );
  ListItem<T> tmp = new ListItem<T>();
  tmp.key = key;
  ..........
}
```

Wir nehmen nur eine der Exception-Klassen hinzu, die von Methode add von java.util.List potentiell geworfen werden. Die anderen Exception-Klassen fangen Fälle ab, die in einer echten Listen-Klasse sicherlich abgefangen werden sollten, in unserer beispielhaften Klasse MyLinkedList aber nur vom eigentlichen Thema ablenken würden und daher ausgelassen werden.

Ganz zu Anfang der Methode fangen wir schon einmal den ersten Fall einer falschen Position ab, nämlich dass der erste Parameter negativ ist. Den zweiten Fall, nämlich dass der erste Parameter zu groß ist und die Liste gar nicht eine solche Position zum Einfügen hat, müssen wir an einer späteren Stelle der Methode add behandeln.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void add ( int pos, T key ) throws IndexOutOfBoundsException {
  if ( pos < 0 )
    throw new IndexOutOfBoundsException ( "Wrong position: "+pos );
  ListItem<T> tmp = new ListItem<T>();
  tmp.key = key;
  ..........
}
```

Wir können auch schon einmal vorab das neue Element erzeugen und sein Attribut key auf den einzufügenden Wert setzen.

***Erinnerung*: Attribut next des neuen Listenelements ist null, weil jedes Attribut eines neu erzeugten Objektes mit dem Nullwert seines Typs initialisiert wird.**

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public void add ( int pos, T key ) throws IndexOutOfBoundsException {
  if ( pos < 0 )
    throw new IndexOutOfBoundsException ( "Wrong position: "+pos );
  ListItem<T> tmp = new ListItem<T>();
  tmp.key = key;
  ..........
}
```

Alles Weitere dann auf den nächsten Folien.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( pos == 0 ) {
  if ( head == null )
    head = tmp;
  else {
    tmp.next = head;
    head = tmp;
  }
  return;
}
```

```
if ( pos == 0 ) {
    tmp.next = head;
    head = tmp;
    return;
}
```

Als erstes betrachten wir den Fall, dass das neue Listenelement ganz vorne, also an Position 0 einzufügen ist. Wir schauen nus zwei äquivalente Varianten an.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( pos == 0 ) {
  if ( head == null )
    head = tmp;
  else {
    tmp.next = head;
    head = tmp;
  }
  return;
}
```

```
if ( pos == 0 ) {
    tmp.next = head;
    head = tmp;
    return;
}
```

Das sind genau die beiden Schritte, die wir uns an den schematischen Bildern für diesen Fall erarbeitet hatten.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( pos == 0 ) {
  if ( head == null )
    head = tmp;
  else {
    tmp.next = head;
    head = tmp;
  }
  return;
}
```

```
if ( pos == 0 ) {
    tmp.next = head;
    head = tmp;
    return;
}
```

Weiter ist nichts zu tun, um den Fall zu behandeln, das die Position 0 ist. Wir haben zwei Möglichkeiten, um zu verhindern, dass noch weitere Anweisungen ausgeführt werden, die nicht für diesen Fall vorgesehen sind: entweder hier ein return oder alles, was auf der nächsten Folie noch folgt, in einen großen else-Block packen. Wir entscheiden uns für das return.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( pos == 0 ) {
  if ( head == null )
    head = tmp;
  else {
    tmp.next = head;
    head = tmp;
  }
  return;
}
```

```
if ( pos == 0 ) {
    tmp.next = head;
    head = tmp;
    return;
}
```

Den Fall, dass die Liste leer ist, hatten wir bisher noch nicht in unsere Überlegungen einbezogen. Bei sehr vielen Operationen auf Listen muss dieser Fall separat betrachtet werden, das werden wir auch gleich beim Entfernen eines Elements sehen. Links sehen Sie die separate Behandlung: Wenn die Liste vor dem Einfügen leer war, besteht sie hinterher aus dem neuen Element. Aber bei dieser Einfügeoperation ist eine separate Behandlung einer leeren Liste nicht notwendig: Sie können sich leicht überlegen, dass der Code rechts im Falle einer leeren Liste exakt dasselbe Ergebnis hat wie der Code links, da die Zuweisung von head an tmp.next dann redundant ist – tmp.next ist schon null und wird nochmals mit null überschrieben.

Wenn die Position größer 0 und die Liste leer ist, dann wäre eine IndexOutOfBoundsException zu werfen. Wir werden gleich sehen, dass dieser Fall aber automatisch mitbehandelt wird bei dem Code zum Einfügen an einer anderen Position, so dass wir uns hier um den Fall, dass die Liste leer und die Position zum Einfügen größer 0 ist, nicht kümmern.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
  for ( ListItem<T> p = head; p != null; p = p.next ) {
        pos−−;
        if ( pos == 0 ) {
          tmp.next = p.next;
          p.next = tmp;
          return;
        }
  }
  throw new IndexOutOfBoundsException ( “Wrong position: “ + pos );
}
```

Das ist jetzt der Rest der Methode add inklusive schließender Klammer am Ende des Methodenrumpfes.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ListItem<T> p = head; p != null; p = p.next ) {
      pos−−;
      if ( pos == 0 ) {
        tmp.next = p.next;
        p.next = tmp;
        return;
      }
  }
  throw new IndexOutOfBoundsException ( “Wrong position: “ + pos );
}
```

Wieder die übliche Schleife zum Durchlauf durch alle Elemente der Liste.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
  for ( ListItem<T> p = head; p != null; p = p.next ) {
      pos--;
      if ( pos == 0 ) {
         tmp.next = p.next;
         p.next = tmp;
         return;
      }
  }
  throw new IndexOutOfBoundsException ( "Wrong position: " + pos );
}
```

Wir müssen den Durchlauf aber zwischendurch abbrechen, nämlich wenn wir die Position zum Einfügen erreicht haben. Genauer gesagt, muss der Laufpointer p auf dem Element an Index pos – 1 stehenbleiben. Wir müssen uns dazu überlegen, wo wir das Dekrement von pos im Schleifenrumpf platzieren und welchen Vergleichswert wir in der if-Abfrage von pos hernehmen.

Am einfachsten macht man sich das mit möglichst kleinem Wert für pos klar: Die Fälle, dass pos negativ oder gleich 0 ist, sind schon abgehandelt, 1 ist der kleinste Wert für pos, der durch den hier gezeigten Teil der Methode add zu behandeln ist. In diesem Fall, muss p auf dem ersten Listenelement stehenbleiben, es darf also nie p auf p.next gesetzt werden. Anders formuliert: Die if-Abfrage muss im ersten Schleifendurchlauf true ergeben.

Wenn wir versuchsweise pos mit 0 vergleichen, dann muss also vorher pos von 1 auf 0 heruntergezählt werden, also muss das Dekrement *vor* die if-Abfrage platziert werden.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
  for ( ListItem<T> p = head; p != null; p = p.next ) {
      pos−−;
      if ( pos == 0 ) {
         tmp.next = p.next;
         p.next = tmp;
         return;
      }
  }
  throw new IndexOutOfBoundsException ( “Wrong position: “ + pos );
}
```

Sie können den Gedankengang auch noch einmal mit pos gleich 2 und 3 wiederholen und werden sicherlich finden, dass es da immer noch passt und offensichtlich auch bei allen größeren Werten von pos. Also sind Dekrement und if-Abfrage so korrekt.

***Warnung*: Bei einer derart einfachen Schleife kann man einfach ´mal eben vom Fall 1 auf alle anderen Fälle schließen, aber bei komplizierteren Schleifen sind so simple Überlegungen nicht unbedingt korrekt!**

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
  for ( ListItem<T> p = head; p != null; p = p.next ) {
      pos−−;
      if ( pos == 0 ) {
        tmp.next = p.next;
        p.next = tmp;
        return;
      }
  }
  throw new IndexOutOfBoundsException ( “Wrong position: “ + pos );
}
```

Hier sind wir also jetzt in der Situation, dass p auf dem Element unmittelbar vor der Position steht, an der das neue Element einzufügen ist. Das ist genau die Situation in den schematischen Zeichnungen, und diese beiden Zeilen sind genau die beiden Schritte, die in diesem Fall das neue Element an der korrekten Position in der Liste einhängen.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
  for ( ListItem<T> p = head; p != null; p = p.next ) {
        pos−−;
        if ( pos == 0 ) {
          tmp.next = p.next;
          p.next = tmp;
          return;
        }
  }
  throw new IndexOutOfBoundsException ( “Wrong position: “ + pos );
}
```

Im Methodenrumpf haben wir zwei return-Anweisungen, eine weiter oben auf dieser Folie, die andere auf der vorangehenden Folie. Und wir hatten auch schon ganz am Anfang des Methodenrumpfes eine throws-Anweisung.

Durch diese drei Anweisungen ist gewährleistet, dass jetzt an dieser Stelle der Wert von pos immer noch größer 0 ist. Der Wert von pos wurde aber in der for-Schleife so oft um 1 heruntergezählt, wie es Elemente in der Liste gibt.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
  for ( ListItem<T> p = head; p != null; p = p.next ) {
      pos--;
      if ( pos == 0 ) {
        tmp.next = p.next;
        p.next = tmp;
        return;
      }
  }
  throw new IndexOutOfBoundsException ( "Wrong position: " + pos );
}
```

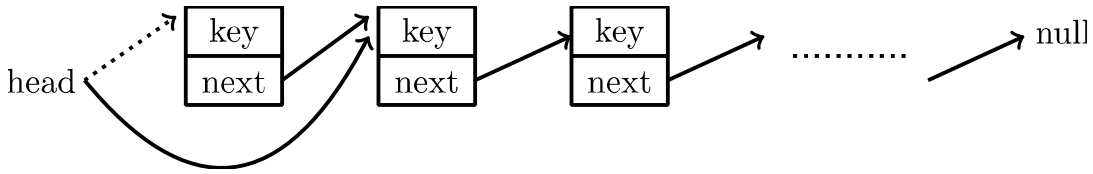
Daraus folgt logisch zwingend, dass der initiale Wert von pos zu groß war, es gibt keine solche Position in der Liste, auch nicht am Ende der Liste. Aus diesem Grund ist es korrekt, hier wieder eine IndexOutOfBoundsException zu werfen.

Beachten Sie, dass die Abarbeitung des Programms auch dann zu diesem Wurf einer Exception kommt, wenn die Position zum Einfügen größer 0 und die Liste leer ist. Denn dann wird die for-Schleife kein einziges Mal durchlaufen, weil die Fortsetzungsbedingung schon beim ersten Test gleich false ist. Diesen Fall – dass die Position größer 0 und die Liste leer ist – hatten wir bei der Behandlung des Falles, dass die Position gleich 0 ist, kurz erwähnt und auf später vertagt; jetzt ist auch dieser Punkt geklärt.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Element entfernen:



head = head.next;

Beim Entfernen eines Elements aus einer Liste müssen wir nun unbedingt zwischen Listenkopf und anderen Listenelementen unterscheiden.

Das Entfernen des Listenkopfes lässt sich in einer einzigen schematischen Zeichnung darstellen: Der bisherige Listenkopf wird sozusagen „übersprungen“.

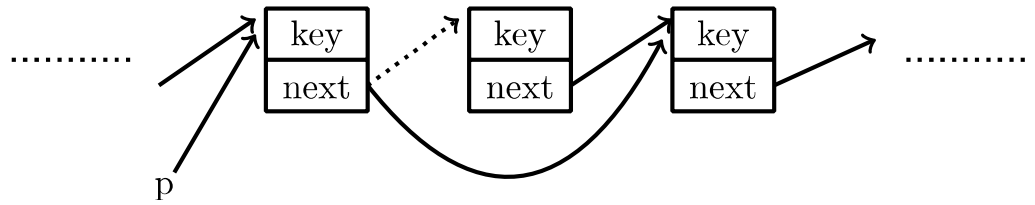
***Erinnerung*: In Kapitel 03b hatten wir gesehen, dass der Garbage Collector den Speicherplatz von solchen Objekten, die vom Programm aus nicht mehr erreichbar sind, irgendwann wieder für anderweitige Verwendung freigibt.**

***Warnung*: In vielen anderen Programmiersprachen, namentlich C und C++, gibt es keinen Garbage Collector in der Definition der Sprache, sondern spezielle Anweisungen, mit denen man den Speicherplatz manuell freigeben kann, bevor er nicht mehr erreichbar ist – aber auch unbedingt freigeben *muss*!**

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Element entfernen:



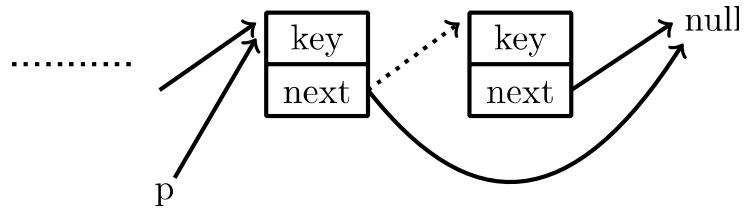
p.next = p.next.next;

Völlig analog funktioniert dieses Überspringen auch im Fall, dass ein Element irgendwo mitten in der Liste entfernt werden soll. Der Laufpointer muss auf dem Element *vor* dem zu entfernenden Element stehenbleiben, denn bei *diesem* Element muss der Inhalt des next-Attributs verändert werden.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Element entfernen:



p.next = p.next.next;

Und auch hier braucht wie beim Einfügen nicht der Fall gesondert betrachtet zu werden, dass wir genau am Listenende sind, denn die Anweisung zum Überspringen des Elements ist exakt dieselbe.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean remove ( Object obj ) {
  if ( head == null )
    return false;
  if ( obj.equals(head.key) ) {
    head = head.next;
    return true;
  }
  .....
```

Wie die einfachen Zeichnungen schon suggerieren, ist die Methode remove deutlich kürzer als die Methode add. Dennoch passt auch sie nicht auf eine einzige Folie mit einer vernünftigen Schriftgröße. Die Exceptions, die Methode remove potentiell wirft, sind für unsere Belange allesamt irrelevant und daher ausgelassen.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean remove ( Object obj ) {
  if ( head == null )
    return false;
  if ( obj.equals(head.key) ) {
    head = head.next;
    return true;
  }
  .....
```

Wir hatten schon gesehen, dass remove wie contains einen Parameter vom Typ Object hat. Da die Methode equals von Object zur Identifikation des zu löschenden Wertes verwendet wird, ist das aber kein Problem. Mit formalem Parametertyp Object ist die Methode maximal allgemein formuliert.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean remove ( Object obj ) {
  if ( head == null )
    return false;
  if ( obj.equals(head.key) ) {
    head = head.next;
    return true;
  }
  ..........
```

Wie beim Einfügen, müssen wir auch beim Entfernen – und natürlich auch bei jeder anderen Operation – den Fall mitbedenken, dass die Liste leer ist. Bei manchen Operationen wie Einfügen und Löschen muss man den Fall explizit angehen so wie hier. Aber beispielsweise bei processAll und contains war das nicht nötig: Bei einer leeren Liste wird die for-Schleife bei processAll und contains kein einziges Mal ausgeführt, und das Resultat ist jeweils korrekt.

Aber auch in solchen Fällen sollte man sich kurz Rechenschaft darüber ablegen, dass der Fall einer leeren Liste tatsächlich korrekt behandelt ist. Das hatten wir bei processAll und contains ja auch gemacht.

***Erinnerung an Racket*: Wir hatten immer wieder betont, dass gerade die Randfälle mitzubedenken sind, und eine leere Liste ist natürlich ein solcher Randfall, den wir bei Racket auch immer mit check-expect getestet hatten.**

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean remove ( Object obj ) {
  if ( head == null )
    return false;
  if ( obj.equals(head.key) ) {
    head = head.next;
    return true;
  }
  ..........
```

Hier wird der Fall geprüft und behandelt, dass das erste Element schon das gesuchte ist.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean remove ( Object obj ) {
  if ( head == null )
    return false;
  if ( obj.equals(head.key) ) {
    head = head.next;
    return true;
  }
  ..........
```

Das ist genau die Anweisung zum Überspringen des ersten Listenelements.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean remove ( Object obj ) {
  if ( head == null )
    return false;
  if ( obj.equals(head.key) ) {
    head = head.next;
    return true;
  }
  ..........
```

Danach ist in diesem Fall in der Methode nichts weiter zu tun, denn die Anforderung an remove ist, genau ein Vorkommen des gesuchten Objektes zu entfernen. Sollte es noch weitere wertgleiche Objekte in der Liste geben, sollen sie unangetastet bleiben.

Und da wir nun ein Element entfernt haben, wissen wir auch in diesem Moment schon den korrekten Rückgabewert, nämlich true. Daher ist es insgesamt korrekt, die Methode hier vorzeitig abzubrechen und true zurückzuliefern.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
    for ( ListItem<T> p = head; p.next != null; p = p.next )
        if ( obj.equals(p.next.key) ) {
          p.next = p.next.next;
          return true;
        }
    return false;
}
```

Nun der Rest der Methode remove.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
    for ( ListItem<T> p = head; p.next != null; p = p.next )
        if ( obj.equals(p.next.key) ) {
          p.next = p.next.next;
          return true;
        }
    return false;
}
```

Der nun sicherlich schon vertraute Durchlauf durch die Liste, ...

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
    for ( ListItem<T> p = head; p.next != null; p = p.next )
        if ( obj.equals(p.next.key) ) {
          p.next = p.next.next;
          return true;
        }
    return false;
}
```

... allerdings mit einem kleinen, aber entscheidenden Unterschied: Im Schleifenrumpf wird vorausgesetzt, dass nicht nur p, sondern auch p.next existiert. Daher muss die Schleife beendet werden, wenn p.next eben nicht mehr existiert, also gleich null ist. Das ist auch kein Problem, denn sobald p.next gleich null ist, ist ja kein Listenelement mehr an der Stelle p.next zu prüfen.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
    for ( ListItem<T> p = head; p.next != null; p = p.next )
        if ( obj.equals(p.next.key) ) {
          p.next = p.next.next;
          return true;
        }
    return false;
}
```

Da wir schon vorher, auf der letzten Folie, den Fall behandelt hatten, dass head gleich null ist, ist es kein Problem, dass wir im Schleifenkopf und im Schleifenrumpf im ersten Durchlauf über p auf head.next zugreifen.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
    for ( ListItem<T> p = head; p.next != null; p = p.next )
        if ( obj.equals(p.next.key) ) {
            p.next = p.next.next;
            return true;
        }
    return false;
}
```

Wir dürfen dann aber auch nicht vergessen, Attribut key nicht bei p, sondern bei p.next zu testen. Das heißt, beim ersten Schleifendurchlauf ist p auf Position 0 und Position 1 wird getestet, beim zweiten Durchlauf ist p auf Position 1 und Position 2 wird getestet, und so weiter. Position 0 braucht in der Schleife nicht getestet zu werden, das hatten wir vorher schon erledigt, auf der letzten Folie.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
    for ( ListItem<T> p = head; p.next != null; p = p.next )
        if ( obj.equals(p.next.key) ) {
          p.next = p.next.next;
          return true;
        }
    return false;
}
```

Das ist wieder die Anweisung zum Überspringen des zu entfernenden Elements.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
    for ( ListItem<T> p = head; p.next != null; p = p.next )
        if ( obj.equals(p.next.key) ) {
          p.next = p.next.next;
          return true;
        }
    return false;
}
```

Und auch hier ist es aus denselben Gründen wie eben beim Entfernen des Listenkopf auf der letzten Folie korrekt, die Methode sofort abzubrechen und true zurückzuliefern.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
    for ( ListItem<T> p = head; p.next != null; p = p.next )
        if ( obj.equals(p.next.key) ) {
          p.next = p.next.next;
          return true;
        }
    return false;
}
```

Durch die beiden vorangehenden return-Anweisungen – auf dieser und der vorherigen Folie – ist gewährleistet, dass die Abarbeitung der Methode genau dann an diese Stelle kommt, wenn das gesuchte Objekt weder der Listenkopf noch ein anderes Listenelement ist. Rückgabe false ist in diesem Fall korrekt, und damit ist auch die Methode remove fertig.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class MyLinkedListIterator <T> implements Iterator <T> {
   private ListItem<T> p;
   public MyLinkedListIterator ( ListItem<T> head ) {
      p = head;
   }
   ..........
}
```

Wir implementieren noch schnell eine passende Iteratorklasse und die Methode iterator von MyLinkedList; zuerst die Iteratorklasse.

Wie ListItem, sollte auch MyLinkedListIterator in der Klasse nicht public sein, denn die bloße Existenz der Klasse MyLinkedListIterator ist ein Implementationsdetail der Klasse MyLinkedList. Außerhalb der Klasse MyLinkedList werden Iteratoren von Klasse MyLinkedListIterator nur durch das Interface Iterator angesprochen, so dass es überhaupt keinen Grund gibt, MyLinkedListIterator außerhalb von MyLinkedList sichtbar werden zu lassen.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class MyLinkedListIterator <T> implements Iterator <T> {
  private ListItem<T> p;
  public MyLinkedListIterator ( ListItem<T> head ) {
     p = head;
  }
  ..........
}
```

Intern ist der Iterator einfach ein Verweis auf ein Listenelement wie bei den bisher gesehenen Methoden von MyLinkedList, nur jetzt nicht als Laufpointer im Rahmen einer for-Schleife, sondern als Attribut.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class MyLinkedListIterator <T> implements Iterator <T> {
   private ListItem<T> p;
   public MyLinkedListIterator ( ListItem<T> head ) {
      p = head;
   }
   ..........
}
```

Dieses Attribut wird dann im Konstruktor des Iterators auf den Kopf der Liste gesetzt.

Die Idee ist, dass p immer auf das nächste mit hasNext zurückzuliefernde Element verweist, also auf das erste in der Liste, dessen Key bis dahin noch *nicht* durch hasNext zurückgeliefert wurde.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class MyLinkedListIterator <T> implements Iterator <T> {
    private ListItem<T> p;
    public MyLinkedListIterator ( ListItem<T> head ) {
        p = head;
    }
    ..........
}
```

Die weiteren Methoden dann auf der nächsten Folie.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean hasNext() {
    return p != null;
}

public T next() {
   T key = p.key;
   p = p.next;
   return key;
}
```

Konkret fehlen noch die Methoden hasNext und next.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean hasNext() {
   return p != null;
}

public T next() {
   T key = p.key;
   p = p.next;
   return key;
}
```

Da ja p immer auf das erste noch nicht zurückgelieferte Listenelement verweisen soll, ist die Abfrage, ob es ein solches Listenelement überhaupt noch gibt, sehr einfach und sicherlich unmittelbar einsichtig.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean hasNext() {
    return p != null;
}

public T next() {
   T key = p.key;
   p = p.next;
   return key;
}
```

Methode next muss p um eine Position weiterschalten. Aber der Key des Elements, auf das p unmittelbar *vor* dieser Fortschaltung verwies, ist der, der zurückgeliefert werden muss. Dieser ist aber *nach* dieser Fortschaltung unerreichbar.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public boolean hasNext() {
   return p != null;
}

public T next() {
  T key = p.key;
  p = p.next;
  return key;
}
```

Deshalb muss die Rückgabe zuerst kurz zwischengespeichert werden, um sie nach der Fortschaltung mit return zurückzuliefern, denn die return-Anweisung ist ja per Definition die letzte Anweisung, die in einer Methode ausgeführt wird, sie kann nicht *vor* der Fortschaltung ausgeführt werden.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public Iterator<T> iterator() {
   return new MyLinkedListIterator ( head );
}
```

Die Methode Iterator von Klasse MyLinkedListIterator ist jetzt ganz einfach und sollte auch ohne weitere Erläuterungen verständlich sein. Damit ist Klasse MyLinkedList soweit fertig, wie gesagt, nur teilweise hier vorgestellt.

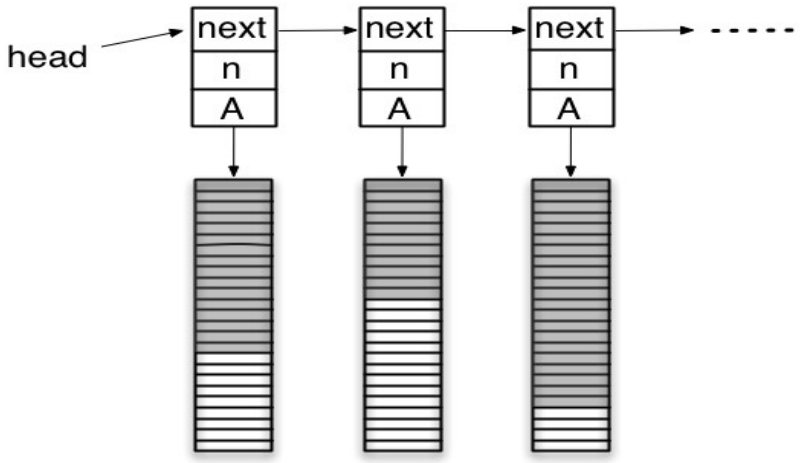
TECHNISCHE UNIVERSITÄT DARMSTADT

Alternative Definition von java.util.LinkedList (kursorisch)

Um ein erstes Gefühl dafür zu bekommen, was hinter der Fassade der Interface Collection und List so alles möglich ist, schauen wir uns kurz eine alternative Möglichkeit an, wie man LinkedList realisieren könnte. Wir belassen es aber mit schematischen Zeichnungen und setzen diese nur punktuell in Java-Quelltext um.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

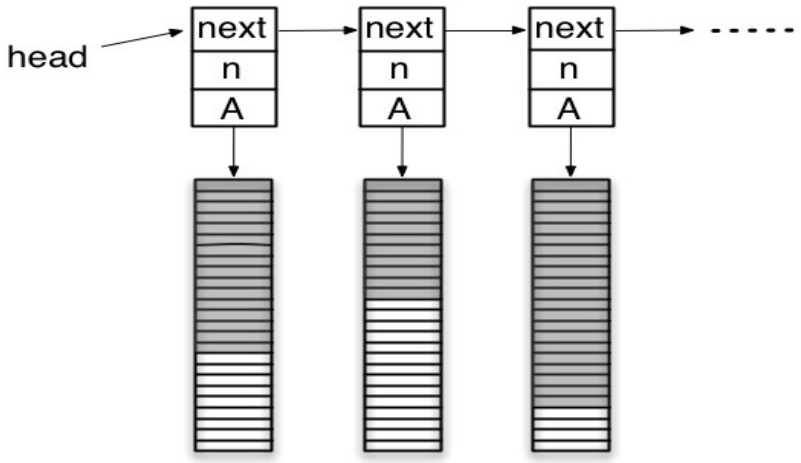


So kann man sich den private-Bereich dieser Klasse in etwa vorstellen: eine Liste von Listenelementen, die neben next jetzt kein Attribut key haben, sondern statt dessen zwei andere: einen Verweis auf ein Array sowie eine ganze Zahl n. Die Arrays haben alle dieselbe Größe.

Die Elemente der Liste sind in den Komponenten dieser Arrays abgespeichert, aber nicht alle Komponenten dieser Arrays sind umgekehrt auch Listenelemente. Die Idee ist, dass nur die ersten n Komponenten eines Arrays tatsächlich als Listenelemente zu verstehen sind, alle weiteren Komponenten des Arrays sind auf null gesetzt. Angedeutet wird dies im Bild durch die graue Unterlegung: Das sind die Arraykomponenten, die Listenelemente enthalten, und das zugehörige Attribut n enthält genau die Anzahl der grau unterlegten Komponenten im zugehörigen Array.

Alternative LinkedList

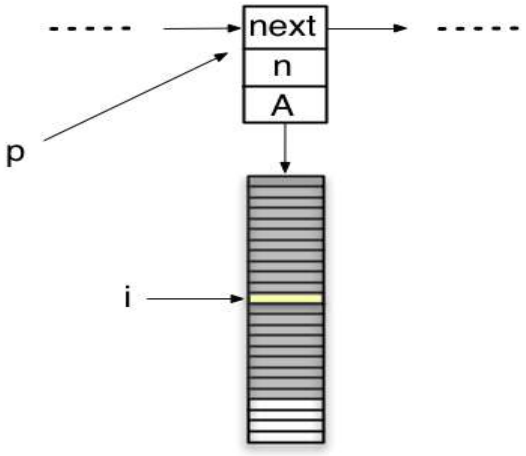
TECHNISCHE UNIVERSITÄT DARMSTADT



***Nebenbemerkung*: Wir hatten schon festgestellt, dass Arrays vom generischen Parametertyp in Java nicht erzeugt werden können. Deshalb muss man an dieser Stelle ein wenige „tricksen“, aber das betrachten wir hier nicht, darauf kommt es uns nicht an.**

Alternative LinkedList

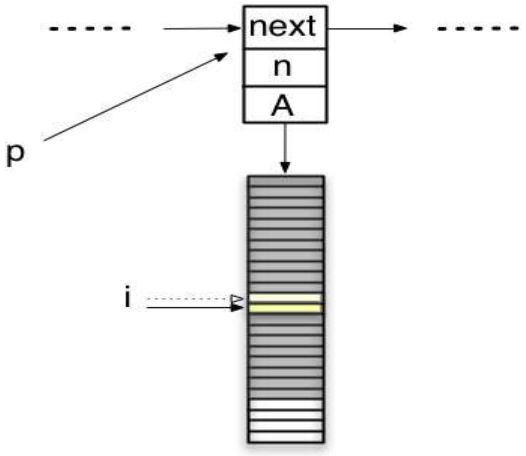
TECHNISCHE UNIVERSITÄT DARMSTADT



Die Listenelemente werden dann zweistufig durchlaufen, mit einem Laufindex i und einem Laufpointer p. Beide zusammen definieren, welches gerade das aktuelle Element ist, und beide zusammen werden benötigt, um auf das aktuelle Element zuzugreifen.

Alternative LinkedList

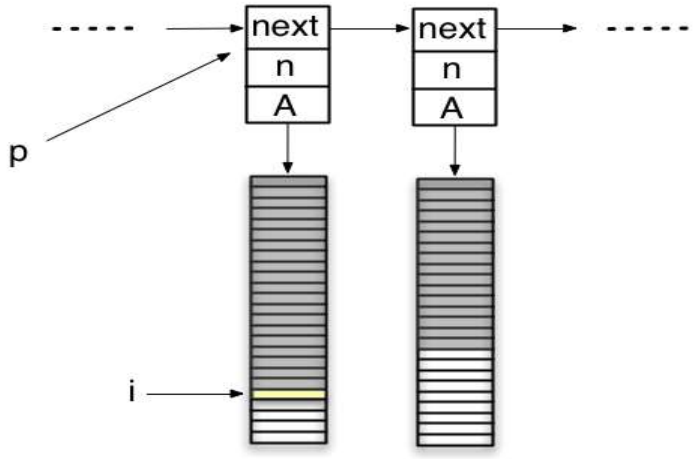
TECHNISCHE UNIVERSITÄT DARMSTADT



Solange der Laufindex kleiner als der zugehörige Wert n ist, bedeutet Vorwärtsgehen in der Liste einfach, den Laufindex um 1 zu erhöhen.

Alternative LinkedList

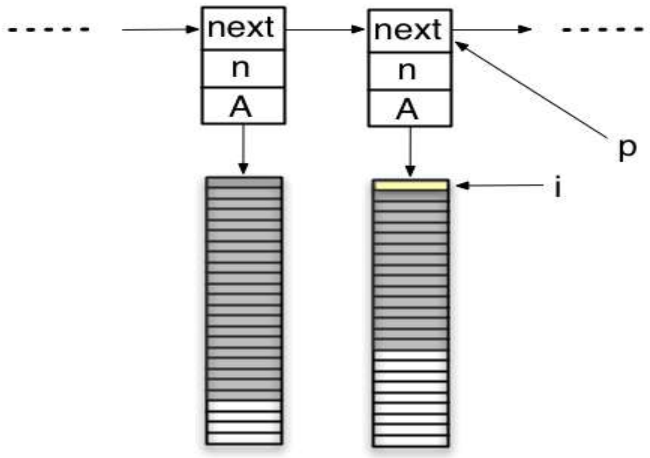
TECHNISCHE UNIVERSITÄT DARMSTADT



Ist hingegen der Laufindex i gleich dem zugehörigen Wert n, dann kann nicht einfach der Laufindex nochmals erhöht werden, denn danach kommen keine Listenelemente mehr in diesem Array.

Alternative LinkedList

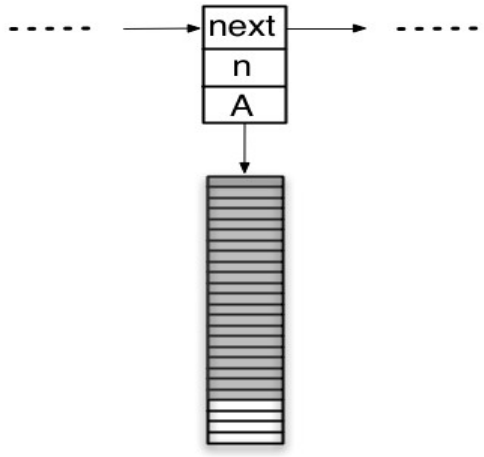
TECHNISCHE UNIVERSITÄT DARMSTADT



Statt dessen wird p um eine Position weitergesetzt, und der Laufindex wird auf 0 gesetzt. Das ist dann das nächste Element der Liste.

Alternative LinkedList

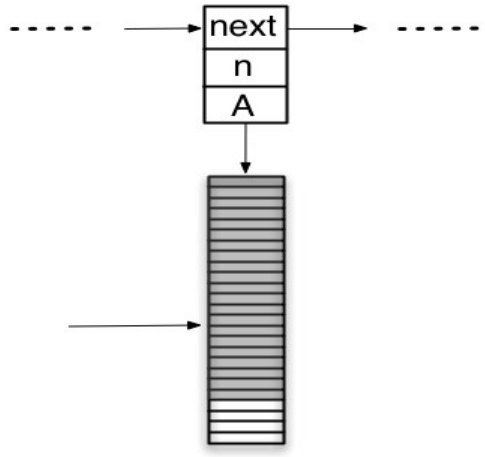
TECHNISCHE UNIVERSITÄT DARMSTADT



Als nächstes Einfügen eines neuen Listenelements.

Alternative LinkedList

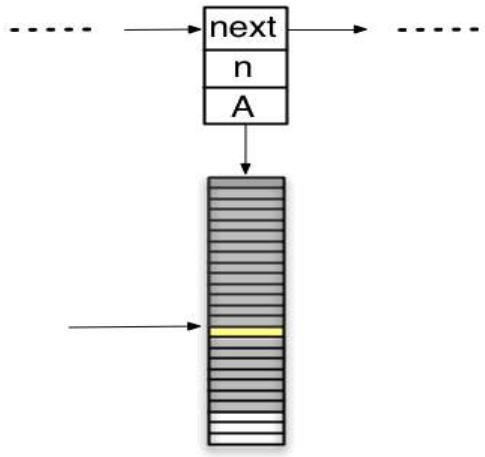
TECHNISCHE UNIVERSITÄT DARMSTADT



Hier sei die Position, an der das neue Listenelement eingefügt werden soll.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT



Wenn noch Platz im Array ist, also wenn n noch nicht gleich der Arraylänge ist, dann ist das Einfügen relativ einfach: Die nachfolgenden Arraykomponenten müssen jeweils um eine Position weiter nach hinten im Array verschoben werden. Den Quelltext für diese Verschiebeaktion schauen wir uns auf der folgenden Folie an.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( int j = p.n-1; j >= i; j-- )
    p.a[j+1] = p.a[j];
p.a[i] = t;
p.n++;
```

Das ist der gesamte Java-Code zum Einfügen eines neuen Elements in ein noch nicht volles Array.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( int j = p.n-1; j >= i; j-- )
    p.a[j+1] = p.a[j];
p.a[i] = t;
p.n++;
```

Die Schleife muss von hinten nach vorne laufen, vom letzten bis zum ersten zu verschiebenden Element, ...

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( int j = p.n-1; j >= i; j−− )
   p.a[j+1] = p.a[j];
p.a[i] = t;
p.n++;
```

... denn dann kann die Arraykomponente an Position j + 1 überschrieben werden, da sie ja im vorangegangenen Schleifendurchlauf schon nach Position j + 2 kopiert worden war.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( int j = p.n-1; j >= i; j-- )
    p.a[j+1] = p.a[j];
p.a[i] = t;
p.n++;
```

Insbesondere wurde die Arraykomponente an Index i nach Index i+1 kopiert, so dass der neu einzufügende Wert jetzt ohne Informationsverlust die Arraykomponente an index i überschreiben kann.

Alternative LinkedList

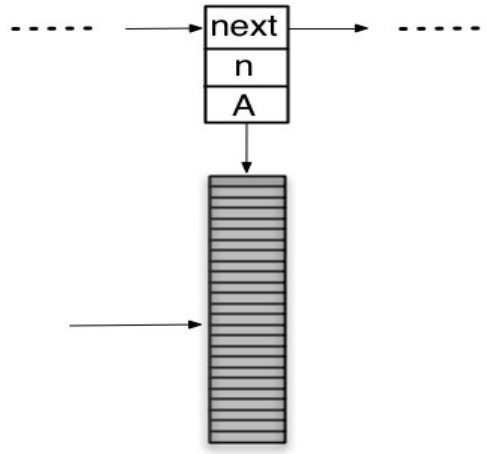
TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( int j = p.n-1; j >= i; j-- )
    p.a[j+1] = p.a[j];
p.a[i] = t;
p.n++;
```

Und natürlich darf nicht vergessen werden, den Zähler für dieses Array um eins hochzusetzen.

Alternative LinkedList

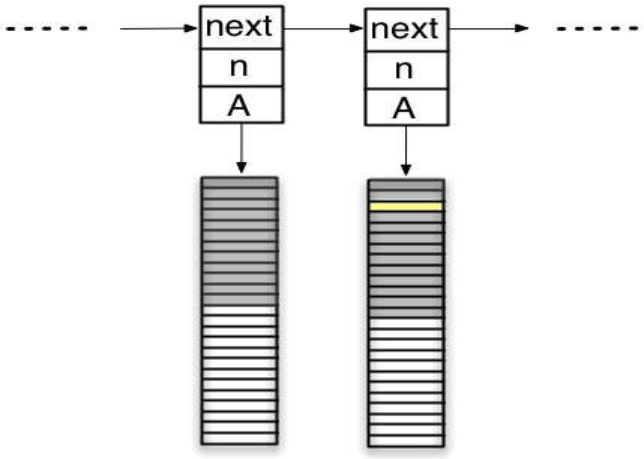
TECHNISCHE UNIVERSITÄT DARMSTADT



Nun kann aber auch der Fall eintreten, dass das Array voll ist, in das das neue Element einzufügen wäre.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT



In diesem Fall wird ein neues Element unmittelbar hinter das Element mit dem vollen Array in die Liste von Arrays eingefügt, und die Komponenten des alten Arrays werden teilweise in das neue Array verschoben. Egal ob die Position zum Einfügen im alten Array verblieben oder ins neue Array verschoben ist: In jedem Fall ist durch diese Aufteilung jetzt Platz in diesem Array, und das neue Element kann wie vorher einfach eingefügt werden.

Wir teilen die Arraykomponenten hier fifty-fifty auf. Das ist nicht unbedingt so notwendig, aber zweckmäßig: Wir wollen natürlich möglichst selten Arrays neu einrichten oder wieder entfernen, das kostet nur Laufzeit. Mit fifty-fifty dauert es sehr lange, bis einer der beiden Arrays voll und wieder aufgeteilt wird oder durch Entfernen von Elementen leer und zu entfernen ist.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
//  For even array sizes only!

    int m = p.n/2;
    for ( int j = 0;  j < m; j++ ) {
       p.next.a[j] = p.a[j+m];
       p.a[j+m] = null;
    }
    p.next.n = m;
    p.n = p.n – m;
```

Der Quelltext für das Verteilen der Arraykomponenten auf beide Arrays sieht so aus.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// For even array sizes only!

int m = p.n/2;
for ( int j = 0;  j < m; j++ ) {
    p.next.a[j] = p.a[j+m];
    p.a[j+m] = null;
}
p.next.n = m;
p.n = p.n – m;
```

Da es uns hier nur ums Prinzip geht, vereinfachen wir den Code etwas und akzeptieren, dass der vereinfachte Code auf dieser Folie nur bei geradzahligen Arraylängen funktioniert.

Als Übung können Sie sich einmal überlegen, was verändert werden müsste für ungeradzahlige Arraylängen.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// For even array sizes only!

int m = p.n/2;
for ( int j = 0;  j < m; j++ ) {
    p.next.a[j] = p.a[j+m];
    p.a[j+m] = null;
}
p.next.n = m;
p.n = p.n – m;
```

Wie gesagt, soll die Hälfte aller Komponenten vom alten Array ins neue Array transferiert werden. Damit nicht bei jedem Schleifendurchlauf die Division durch 2 auszuführen ist, wird die Hälfte nur einmal berechnet und in m gespeichert.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// For even array sizes only!

int m = p.n/2;
for ( int j = 0;  j < m; j++ ) {
  p.next.a[j] = p.a[j+m];
  p.a[j+m] = null;
}
p.next.n = m;
p.n = p.n – m;
```

Damit die Reihenfolge der Elemente beim Durchlauf erhalten bleibt, muss die hintere Hälfte in das neue Array kopiert werden – dort natürlich in die *vordere* Hälfte des Arrays.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// For even array sizes only!

int m = p.n/2;
for ( int j = 0;  j < m; j++ ) {
   p.next.a[j] = p.a[j+m];
   p.a[j+m] = null;
}
p.next.n = m;
p.n = p.n – m;
```

Wir hatten schon gesagt, dass diejenigen Arraykomponenten, die momentan *nicht* genutzt werden, null sein sollen. Das wird hier gewährleistet.

Das Nullsetzen von nicht mehr benötigten Arraykomponenten ist nicht zwingend notwendig, denn im Attribut n steht ja drin, welche Arraykomponenten in Gebrauch sind und welche nicht. Es hat aber einen wichtigen Vorteil: Sobald es von außerhalb dieser LinkedList-Klasse ebenfalls keinen Verweis mehr auf ein solches Objekt gibt, ist das Objekt vom Programm aus nicht mehr erreichbar und kann durch den Garbage Collector freigegeben werden.

Alternative LinkedList

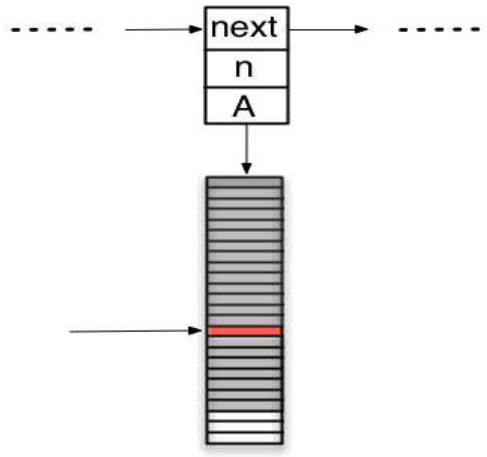
TECHNISCHE UNIVERSITÄT DARMSTADT

```
// For even array sizes only!

    int m = p.n/2;
    for ( int j = 0;  j < m; j++ ) {
        p.next.a[j] = p.a[j+m];
        p.a[j+m] = null;
    }
    p.next.n = m;
    p.n = p.n – m;
```

Schlussendlich müssen wir noch das Attribut n in beiden Listenelementen entsprechend aktualisieren.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT



Beim Entfernen eines Elements ist das Vorgehen analog, nur eben genau umgekehrt. Wenn das zu entfernende Element nicht das einzige ist, also wenn n nicht momentan gleich 1 ist, dann kann man die Verschiebung aller darauffolgenden Arraykomponenten um einen Index genau umkehren.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
p.n−−;
for ( int j = i; j < p.n; j++ )
    p.a[j] = p.a[j+1];
p.a[p.n] = null;
```

Das ist der Quelltext zur Verschiebung der Komponenten beim Entfernen.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
p.n−−;
for ( int j = i; j < p.n; j++ )
    p.a[j] = p.a[j+1];
p.a[p.n] = null;
```

Analog zur Logik beim Einfügen, nur eben umgedreht, geht die Schleife in diesem Fall von der Position, an der das Element entfernt werden soll, bis zum Ende, ...

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
p.n−−;
for ( int j = i; j < p.n; j++ )
   p.a[j] = p.a[j+1];
p.a[p.n] = null;
```

... denn dann wird der Wert an Index j erst überschrieben, nachdem er an Index j – 1 kopiert wurde.

Beachten Sie, dass das zu entfernende Element überhaupt nicht irgendwie behandelt werden muss, denn der erste Durchlauf durch die Schleife überschreibt dieses Element, und damit ist es automatisch entfernt.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT

```
p.n−−;
for ( int j = i; j < p.n; j++ )
    p.a[j] = p.a[j+1];
p.a[p.n] = null;
```

Dieses Element ist im ersten Schleifendurchlauf an Position n – 1 kopiert worden und muss jetzt an Position n entfernt werden.

Alternative LinkedList

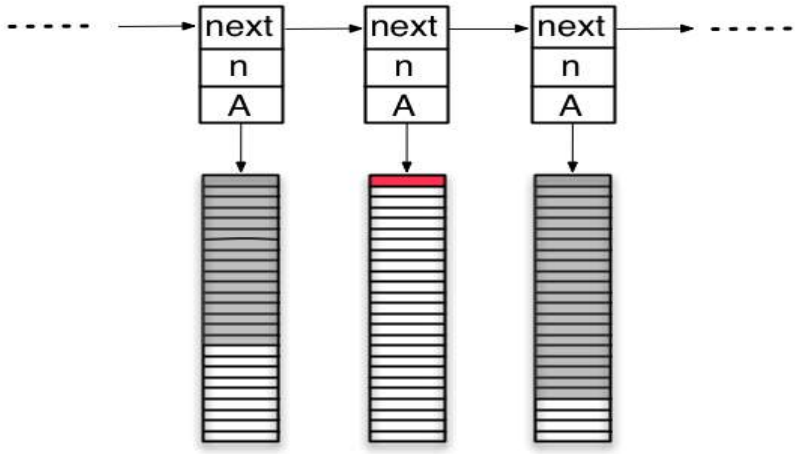
TECHNISCHE UNIVERSITÄT DARMSTADT

```
p.n−−;
for ( int j = i; j < p.n; j++ )
    p.a[j] = p.a[j+1];
p.a[p.n] = null;
```

Natürlich muss dann auch n wieder um eins vermindert werden. Beachten Sie, dass die Indizes in den darauffolgenden Zeilen dadurch stimmen, dass n schon *vorher* dekrementiert wurde.

Alternative LinkedList

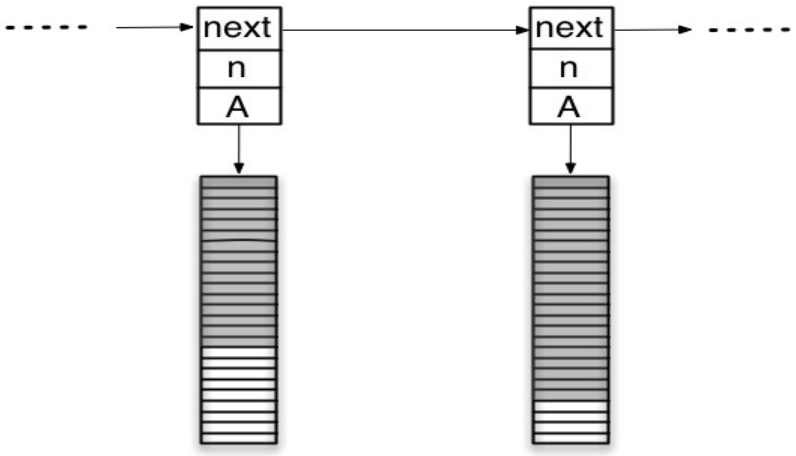
TECHNISCHE UNIVERSITÄT DARMSTADT



Nun noch die Situation, dass das zu entfernende Element das einzige im Array ist.

Alternative LinkedList

TECHNISCHE UNIVERSITÄT DARMSTADT



In diesem Fall wird umstandslos das ganze Array entfernt.

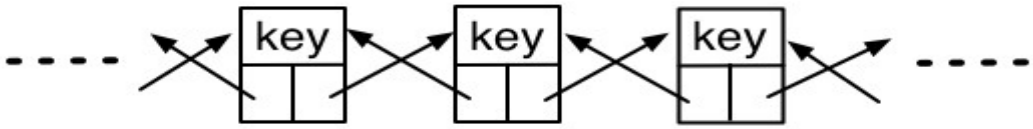
TECHNISCHE UNIVERSITÄT DARMSTADT

Doppelt verkettete Listen

Bis jetzt haben wir Listen nur *vorwärts* verkettet. Jetzt verketten wir sie auch zusätzlich *rückwärts*. Der Vorteil ist, dass man die Liste dann in beiden Richtungen gleichermaßen durchlaufen kann. Der Nachteil ist, dass mehr Speicher benötigt wird und dass Einfügen und Entfernen etwas mehr Laufzeit kosten, wie wir gleich sehen werden.

Doppelt verkettete Listen

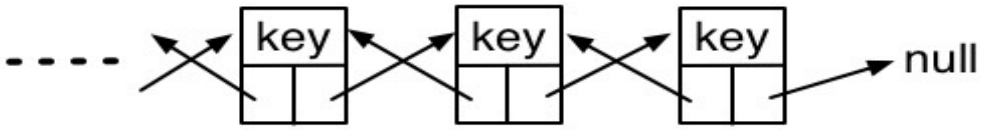
TECHNISCHE UNIVERSITÄT DARMSTADT



Wir sehen uns das nur schematisch an, ohne Java-Code. Für eine doppelte Verkettung braucht es halt *zwei* Verweise auf andere Elemente in jedem Listenelement. Häufig nennt man sie next und backward oder forward und backward.

Doppelt verkettete Listen

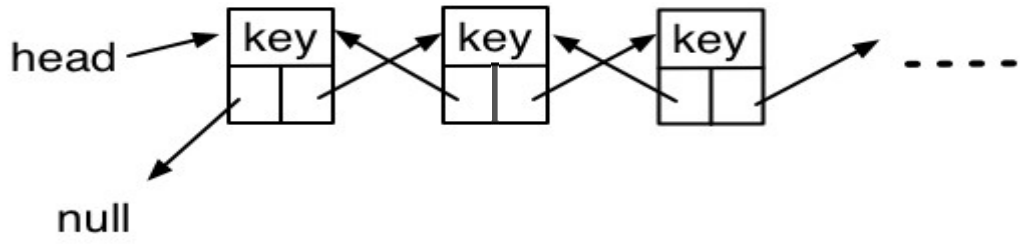
TECHNISCHE UNIVERSITÄT DARMSTADT



Am Ende der Liste sieht alles genauso aus wie bisher.

Doppelt verkettete Listen

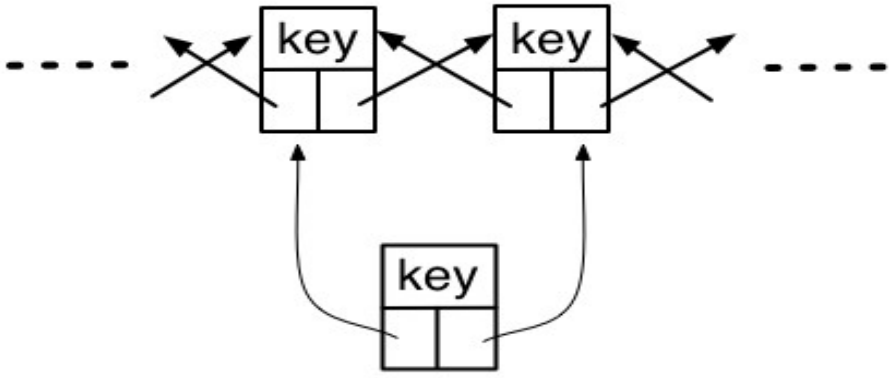
TECHNISCHE UNIVERSITÄT DARMSTADT



Spiegelsymmetrisch dazu ist der Rückwärtsverweis beim ersten Listenelement auf null gesetzt.

Doppelt verkettete Listen

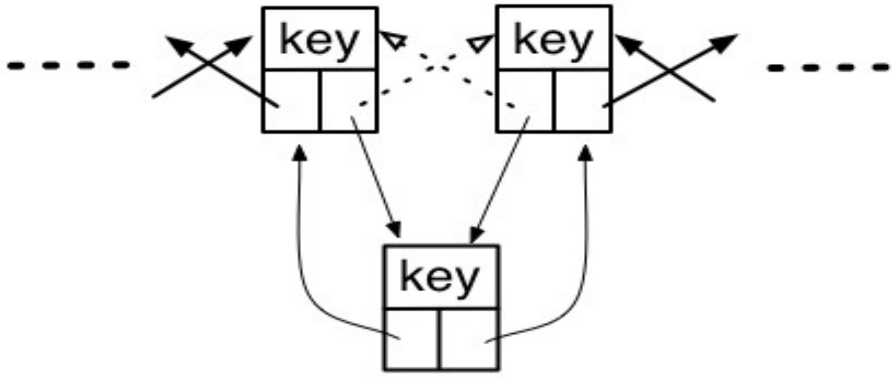
TECHNISCHE UNIVERSITÄT DARMSTADT



Beim Einfügen verdoppeln sich die beiden notwendigen Schritte spiegelbildlich: Zuerst werden die beiden Verweise des neuen Elements auf die beiden Elemente an der Einfügestelle gesetzt, ...

Doppelt verkettete Listen

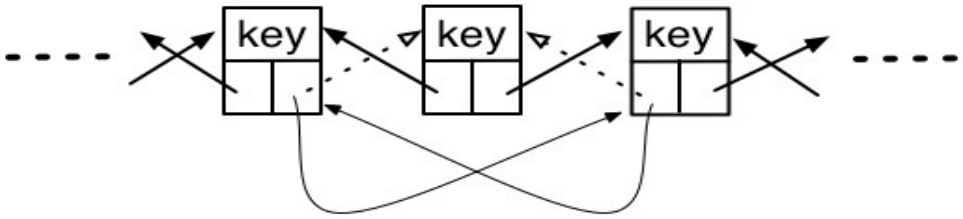
TECHNISCHE UNIVERSITÄT DARMSTADT



... danach werden der Vorwärtsverweise des vorderen und der Rückwärtsverweis des hinteren dieser beiden Elemente auf das neue Element gesetzt.

Doppelt verkettete Listen

TECHNISCHE UNIVERSITÄT DARMSTADT



Und schließlich beim Entfernen eines Elements müssen beide Verweise, die auf dieses Element zeigen, jeweils eins weiter in der Liste gesetzt werden – der Vorwärtsverweis eins weiter nach vorne, der Rückwärtsverweis eins weiter nach hinten.

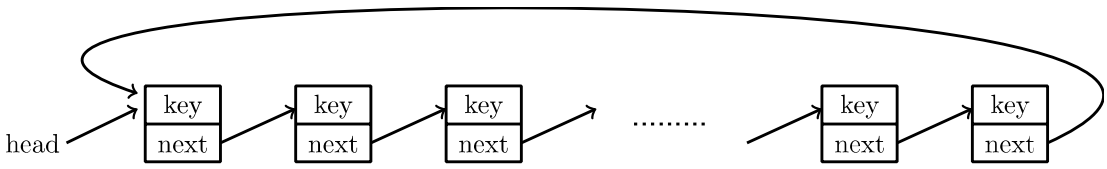
TECHNISCHE UNIVERSITÄT DARMSTADT

Zyklische Listen

Ganz kurz noch eine weitere oftmals hilfreiche Variation von Listen.

Zyklische Listen

TECHNISCHE UNIVERSITÄT DARMSTADT



head = head.next;

Die Variation besteht einfach darin, dass der letzte Verweis nicht auf null, sondern wieder auf das erste Element gesetzt ist. So haben wir keine lineare Liste mehr, sondern eine zyklische.

Solche zyklische Listen werden unter anderem in Rundlaufverfahren, englisch *Round Robin*, eingesetzt, zum Beispiel bei der Vergabe von Zeit auf einem Prozessor: Jedes Listenelement hält Informationen zu einem Prozess, der auf diesem Prozessor laufen soll. Die Prozesse bekommen nacheinander immer eine kleine Zeitscheibe und werden dann wieder suspendiert. Der momentan laufende Prozess ist der, auf den head verweist. Ist dessen Zeitscheibe abgelaufen, geht head einen Schritt weiter in der Liste, und der nächste Prozess wird aktiviert. Und so weiter.

TECHNISCHE UNIVERSITÄT DARMSTADT

Garbage Collector (und damit sich selbst) „austricksen“

Die Möglichkeit zur Listenbildung beinhaltet auch die Gefahr, dass der Speicherplatz durch einen Programmierfehler doch noch durch immer neue Objekte vollständig aufgebraucht wird und irgendwann das Programm mit Fehler abbricht, weil kein Speicherplatz mehr für das nächste einzurichtende Objekt vorhanden ist.

Natürlich wird man so etwas nicht absichtlich machen, sondern aus Versehen, aber das Ergebnis bleibt dasselbe.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X head = new X();
while ( true ) {
    X tmp = new X();
    tmp.next = head;
    head = tmp;
}
```

Dafür reicht schon dieses kleine künstliche Beispiel. Die Klasse X habe ein Attribut next vom Typ X.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X head = new X();
while ( true ) {
    X tmp = new X();
    tmp.next = head;
    head = tmp;
}
```

Wir richten uns ganz normal ein Objekt von Klasse X ein. Die Variable head wird immer auf das erste Objekt in der stetig wachsenden Liste verweisen.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X head = new X();
while ( true ) {
    X tmp = new X();
    tmp.next = head;
    head = tmp;
}
```

Jetzt eine Endlosschleife, in der in jedem Durchlauf ein Objekt mit Operator new eingerichtet wird.

Eigene LinkedList-Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X head = new X();
while ( true ) {
    X tmp = new X();
    tmp.next = head;
    head = tmp;
}
```

Wir hatten schon gesehen, dass diese beiden Anweisungen zusammen das neue Element vorne an die Liste anhängt, auf die head verweist.

Kein Element der Liste darf vom Garbage Collector freigegeben werden, denn sie alle sind vom Programm aus erreichbar. Es ist nur eine Frage der Zeit, bis der gesamte zur Verfügung stehende Speicherplatz aufgebraucht ist.

Damit ist das Thema Collections und somit das ganze Kapitel beendet.