Teil VII

Algorithmen und Datenstrukturen – Java Collections

Arrays und Klassen

Arrays

- Homogene Sammlung von Elementen **gleichen** Typs
- Vorteile
  - effizient
  - einfach
- Nachteil: unflexibel
  - Größe muss beim Anlegen feststehen
  - Größe kann nicht verändert werden
  - Einfügen zwischen zwei Elementen nicht möglich
  - Einfügen und Löschen am Anfang nicht möglich

Arrays und Klassen

Arrays

- Homogene Sammlung von Elementen **gleichen** Typs
- Vorteile
  - effizient
  - einfach
- Nachteil: unflexibel
  - Größe muss beim Anlegen feststehen
  - Größe kann nicht verändert werden
  - Einfügen zwischen zwei Elementen nicht möglich
  - Einfügen und Löschen am Anfang nicht möglich

Klassen

- Heterogene Sammlung von Elementen **unterschiedlichen** Typs (Attribute)
- Zusätzlich Methoden zur Veränderung der Werte dieser Attribute
- Vorteil: sehr mächtig zur Strukturierung, insbesondere durch die Kombination von Datenstrukturen und Algorithmen
- In anderen Programmiersprachen auch ‘record’ oder ‘struct” genannt (ohne Funktionen oder Methoden)

Arrays und Klassen

Arrays

- Homogene Sammlung von Elementen **gleichen** Typs
- Vorteile
  - effizient
  - einfach
- Nachteil: unflexibel
  - Größe muss beim Anlegen feststehen
  - Größe kann nicht verändert werden
  - Einfügen zwischen zwei Elementen nicht möglich
  - Einfügen und Löschen am Anfang nicht möglich

Klassen

- Heterogene Sammlung von Elementen **unterschiedlichen** Typs (Attribute)
- Zusätzlich Methoden zur Veränderung der Werte dieser Attribute
- Vorteil: sehr mächtig zur Strukturierung, insbesondere durch die Kombination von Datenstrukturen und Algorithmen
- In anderen Programmiersprachen auch ‘record’ oder ‘struct” genannt (ohne Funktionen oder Methoden)

Gesucht: Alternativen zur Datenstruktur Array

Collections

Collections:

- Algorithmen und Datenstrukturen
  - zur effizienten und effektiven Verwaltung
  - von homogenen Elementen
  - gleichen Typs

Collections

Collections:

- Algorithmen und Datenstrukturen
  - zur effizienten und effektiven Verwaltung
  - von homogenen Elementen
  - gleichen Typs
- Basis: Klassen
  - Datenstrukturen $\rightarrow$ Attribute
  - Algorithmen $\rightarrow$ Methoden

Collections

Collections:

- Algorithmen und Datenstrukturen
  - zur effizienten und effektiven Verwaltung
  - von homogenen Elementen
  - gleichen Typs
- Basis: Klassen
  - Datenstrukturen $\rightarrow$ Attribute
  - Algorithmen $\rightarrow$ Methoden
- Interfaces
  - Abstrakte Methoden ohne Implementierung

Collections

Collections:

- Algorithmen und Datenstrukturen
  - zur effizienten und effektiven Verwaltung
  - von homogenen Elementen
  - gleichen Typs
- Basis: Klassen
  - Datenstrukturen $\rightarrow$ Attribute
  - Algorithmen $\rightarrow$ Methoden
- Interfaces
  - Abstrakte Methoden ohne Implementierung

In anderen Programmierparadigmen

- Datenstrukturen und Algorithmen können voneinander getrennt sein
- Kombination nur konzeptionell: keine Kapselung des Zugriffs der Algorithmen auf die Datenstrukturen

Collections: Allgemeine Funktionalität

Allgemeine Funktionalität in Sammlungen von Elementen gleichen Typs T

- java.util.Collection<T> (interface)

`void clear()`	Lösche alle Elemente aus der Collection
`int size()`	Anzahl der Elemente der Collection
`boolean isEmpty()`	*true* gdw. Collection ist leer

Collections: Allgemeine Funktionalität

Allgemeine Funktionalität in Sammlungen von Elementen gleichen Typs T

- java.util.Collection<T> (interface)

`void clear()`	Lösche alle Elemente aus der Collection
`int size()`	Anzahl der Elemente der Collection
`boolean isEmpty()`	*true* gdw. Collection ist leer

- java.lang.Iterable<T> (interface)

`Iterator<T> iterator()`	erzeugt Iterator

Collections: Allgemeine Funktionalität

Allgemeine Funktionalität in Sammlungen von Elementen gleichen Typs T

- java.util.Collection<T> (interface)

`void clear()`	Lösche alle Elemente aus der Collection
`int size()`	Anzahl der Elemente der Collection
`boolean isEmpty()`	*true* gdw. Collection ist leer

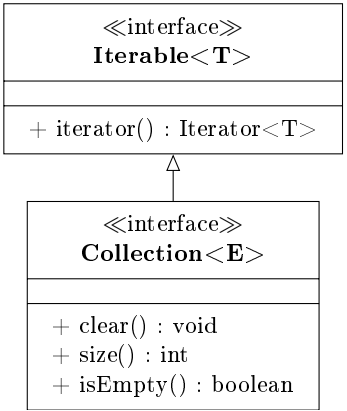
- java.lang.Iterable<T> (interface)

`Iterator<T> iterator()`	erzeugt Iterator

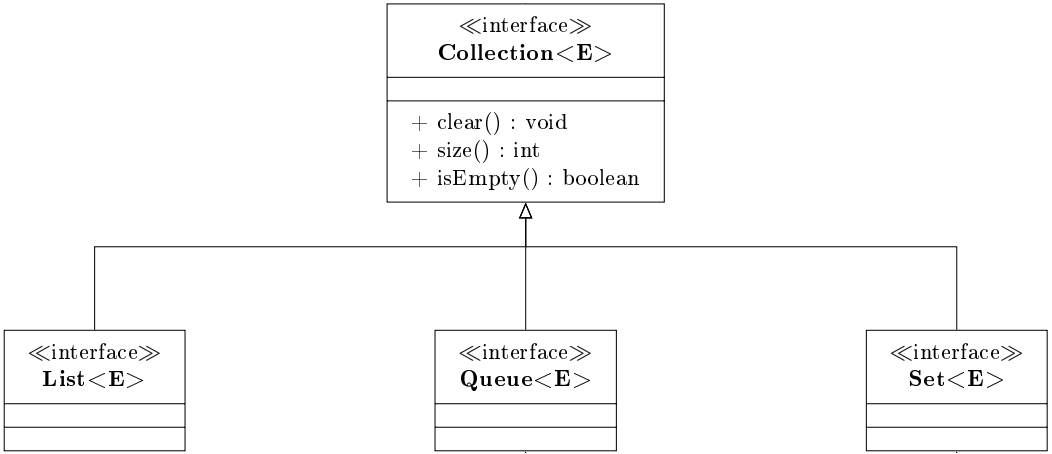
- java.util.Iterator<E> (interface)

`boolean hasNext()`	gibt es ein weiteres Element?
`E next()`	gibt das nächste Element zurück, `null`, falls `hasNext` returns `false`

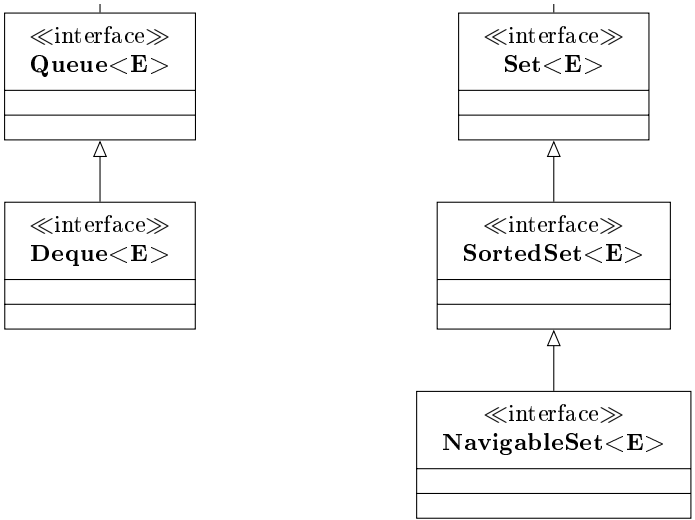
Collections: Allgemeine Funktionalität



Collections: Allgemeine Funktionalität



Collections: Allgemeine Funktionalität



Collections: Interface $\leftrightarrow$ Klasse

Interface wird als **Datentyp** verwendet:

- Typ eines Attributes
- Typ einer Variablen
- Typ eines Parameters
- Typ des Rückgabewertes

Collections: Interface $\leftrightarrow$ Klasse

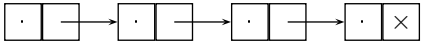
Interface wird als **Datentyp** verwendet:

- Typ eines Attributes
- Typ einer Variablen
- Typ eines Parameters
- Typ des Rückgabewertes

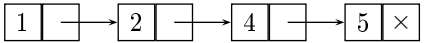
Klasse (Implementierung des Interfaces) wird zur **Instantiierung** verwendet

Listen: konzeptionell

Abstrakte Liste:

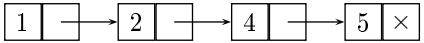


Liste mit Zahlen:

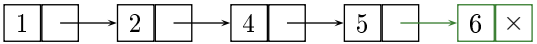


Listen: konzeptionell

Liste mit Zahlen:

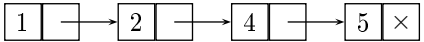


Anfügen am Ende:

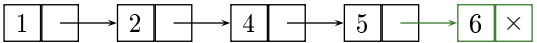


Listen: konzeptionell

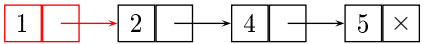
Liste mit Zahlen:



Anfügen am Ende:

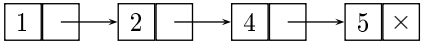


Erstes Element entfernen:

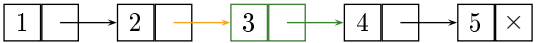


Listen: konzeptionell

Liste mit Zahlen:

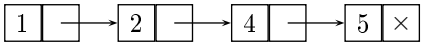


Einfügen in der Mitte (oder am Anfang):

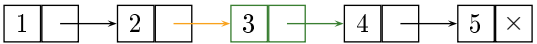


Listen: konzeptionell

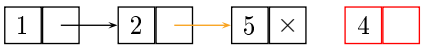
Liste mit Zahlen:



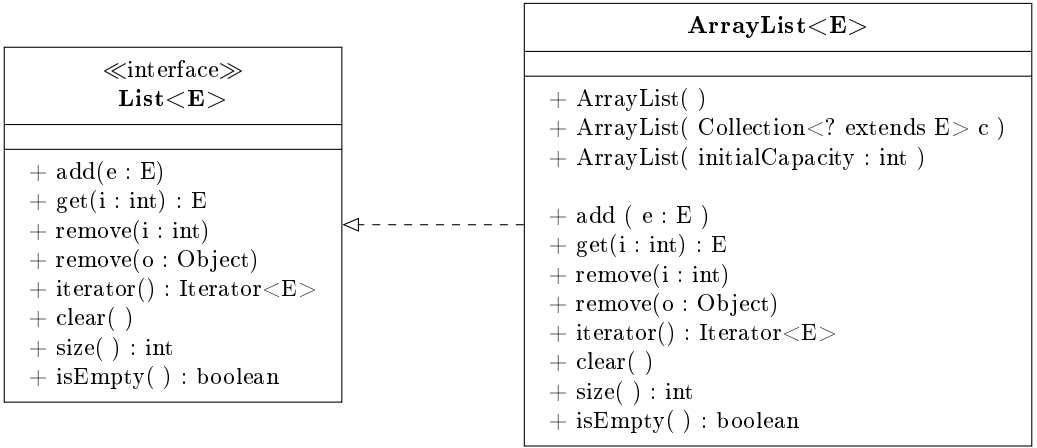
Einfügen in der Mitte (oder am Anfang):



Löschen in der Mitte (oder am Ende):



Listen: List – ArrayList



Listen Implementierung: ArrayList

java.util.ArrayList als Implementierung einer Liste

Eigenschaften:

- wahlfreier Zugriff (wie Array)
- wächst dynamisch (Liste)
- Verwendung: Erzeugung von Instanzen vom Typ `List`

Listen Implementierung: ArrayList

java.util.ArrayList als Implementierung einer Liste

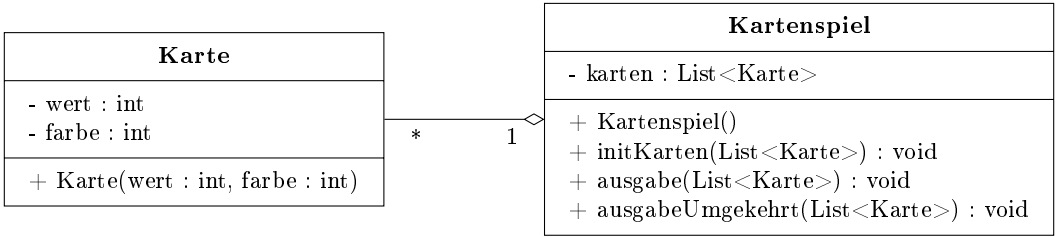
Eigenschaften:

- wahlfreier Zugriff (wie Array)
- wächst dynamisch (Liste)
- Verwendung: Erzeugung von Instanzen vom Typ `List`

Eigenschaften

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Einfügen und Löschen zwischen zwei Elementen ist **einfach**
- Einfügen und Löschen am Anfang und am Ende ist **einfach**

Listen: Beispiel Kartenspiel



Listen: Beispiel Kartenspiel

```
1   public Kartenspiel() {
2     // Erzeuge Liste von Karten
3     karten = new ArrayList<Karte>();
4
5     // Initialisiere Liste von Karten
6     initKarten(karten);
7
8     // Ausgabe der Karten:
9     // Reihenfolge wie in der Liste
10    ausgabe(karten);
11
12    // Ausgabe der Karten:
13    // Reihenfolge umgekehrt wie in der
          Liste
14    ausgabeUmgekehrt(karten);
15  }
```

Listen: Beispiel Kartenspiel

```java
public Kartenspiel() {
  // Erzeuge Liste von Karten
  karten = new ArrayList<Karte>();

  // Initialisiere Liste von Karten
  initKarten(karten);

  // Ausgabe der Karten:
  // Reihenfolge wie in der Liste
  ausgabe(karten);

  // Ausgabe der Karten:
  // Reihenfolge umgekehrt wie in der
     Liste
  ausgabeUmgekehrt(karten);
}
```

```java
public void initKarten(
  List<Karte> karten
) {
  for (
    int farbe = 0;
    farbe <= 3;
    ++farbe
  ) {
    for (
      int wert = 2;
      wert <= 14;
      ++wert
    ) {
      karten.add(new Karte(wert, farbe));
    }
  }
}
```

Listen: Beispiel Kartenspiel

```java
public Kartenspiel() {
  // Erzeuge Liste von Karten
  karten = new ArrayList<Karte>();

  // Initialisiere Liste von Karten
  initKarten(karten);

  // Ausgabe der Karten:
  // Reihenfolge wie in der Liste
  ausgabe(karten);

  // Ausgabe der Karten:
  // Reihenfolge umgekehrt wie in der
     Liste
  ausgabeUmgekehrt(karten);
}
```

```java
public void ausgabe(
  List<Karte> karten
) {
  for (Karte karte : karten) {
    System.out.println(karte.toString());
  }
}
```

Listen: Beispiel Kartenspiel

```
1   public Kartenspiel() {
2     // Erzeuge Liste von Karten
3     karten = new ArrayList<Karte>();
4
5     // Initialisiere Liste von Karten
6     initKarten(karten);
7
8     // Ausgabe der Karten:
9     // Reihenfolge wie in der Liste
10    ausgabe(karten);
11
12    // Ausgabe der Karten:
13    // Reihenfolge umgekehrt wie in der
          Liste
14    ausgabeUmgekehrt(karten);
15  }
```

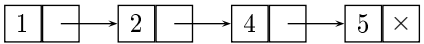
```
1   public void ausgabe(
2     List<Karte> karten
3   ) {
4     for (Karte karte : karten) {
5       System.out.println(karte.toString());
6     }
7   }
```

```
1   public void ausgabeUmgekehrt(
2     List<Karte> karten
3   ) {
4     for (
5       int z = karten.size()-1;
6       z >= 0;
7       --z
8     ) {
9       System.out.println(karten.get(z).
          toString());
10    }
11  }
```

Queue: konzeptionell

Datenstruktur **Queue**: Prinzip FIFO (first-in, first-out)

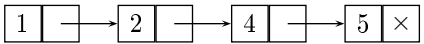
Queue mit Zahlen:



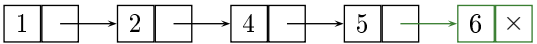
Queue: konzeptionell

Datenstruktur **Queue**: Prinzip FIFO (first-in, first-out)

Queue mit Zahlen:



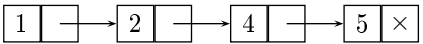
Anfügen am Ende:



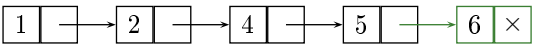
Queue: konzeptionell

Datenstruktur **Queue**: Prinzip FIFO (first-in, first-out)

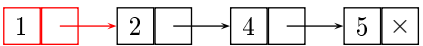
Queue mit Zahlen:



Anfügen am Ende:



Erstes Element entfernen:



Queue: Interface

Datenstruktur **Queue**: Prinzip FIFO (first-in, first-out)
java.util.Queue<E> (Interface)

`add(E e)` fügt e am Ende hinzu

`E remove()`
`E poll()`
} entferne erstes Element

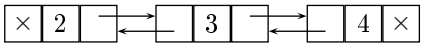
`E element()`
`E peek()`
} gib erstes Element zurück

`remove, element : throw Exception`
`poll, peek : return null`
} falls queue leer

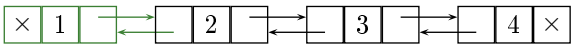
Deque

Datenstruktur **Deque**: zusätzlich Einfügen am Anfang und Entfernen am Ende

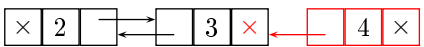
Deque mit Zahlen:



Anfügen am Anfang:



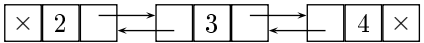
Letztes Element entfernen:



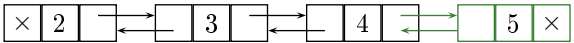
Stack

Datenstruktur **Stack** (Interface: Deque): Prinzip LIFO (last-in, first-out)

Stack mit Zahlen:



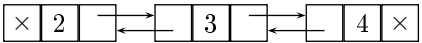
Anfügen am Ende:



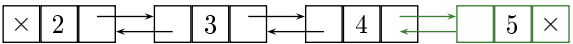
Stack

Datenstruktur **Stack** (Interface: Deque): Prinzip LIFO (last-in, first-out)

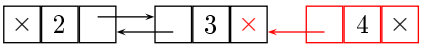
Stack mit Zahlen:



Anfügen am Ende:



Letztes Element entfernen:



Deque

Datenstruktur **Deque**: zusätzlich Einfügen am Anfang und Entfernen am Ende

`extends Queue`
beidseitige Queue:

```
{ add | remove | poll | get | peek } { First | Last }
```

Deque

Datenstruktur **Deque**: zusätzlich Einfügen am Anfang und Entfernen am Ende

`extends Queue`
beidseitige Queue:

```
{ add | remove | poll | get | peek } { First | Last }
```

Stack:

`push(E e)`	fügt e am Anfang hinzu
`E pop()`	entferne erstes Element
`E peek()`	gib erstes Element zurock

Queue, Deque, Stack

Eigenschaften

Queue:

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Einfügen am Ende ist **einfach**
- Abfragen und Entfernen am Anfang ist **einfach**

Queue, Deque, Stack

Eigenschaften

Queue:

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Einfügen am Ende ist **einfach**
- Abfragen und Entfernen am Anfang ist **einfach**

Deque:

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Einfügen am Anfang und am Ende ist **einfach**
- Abfragen und Entfernen am Anfang und am Ende ist **einfach**

Queue, Deque, Stack

Eigenschaften

Queue:

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Einfügen am Ende ist **einfach**
- Abfragen und Entfernen am Anfang ist **einfach**

Deque:

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Einfügen am Anfang und am Ende ist **einfach**
- Abfragen und Entfernen am Anfang und am Ende ist **einfach**

Stack:

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Auf den Stapel (Stack) legen ist **einfach**
- Vom Stapel (Stack) nehmen ist **einfach**

Implementierungen: ArrayDeque, LinkedList

java.util.ArrayDeque

als Implementierung von

- Queue
- Deque

Implementierungen: ArrayDeque, LinkedList

java.util.ArrayDeque

als Implementierung von

- Queue
- Deque

java.util.LinkedList

als Implementierung von

- List
- Queue
- Deque

Mengen

Datenstruktur **Menge**
(`java.util.Set`):

- Sammlung von Elementen gleichen Typs
- Keine Duplikate
- Keine Ordnung
- Keine Teilmengen

Mengen

Datenstruktur **Menge**
(`java.util.Set`):

- Sammlung von Elementen gleichen Typs
- Keine Duplikate
- Keine Ordnung
- Keine Teilmengen

Datenstruktur **sortierte Menge**
(`java.util.SortedSet`):

- Sammlung von Elementen gleichen Typs
- Keine Duplikate
- Ordnung (Sortierung)
  - Kleinstes Element (`first()`)
  - Größtes Element (`last()`)
- Teilmenge (`subset(E from, E to)`); Beispiel:
  - Eingabe: $S = \{1,2,3,4,5\}$ (kein Java-Code)
  - Aufruf: `S.subset(2, 4)`
  - Ergebnis: $S = \{2,3\}$ (kein Java-Code)

Mengen Implementierung: HashSet, TreeSet

Eigenschaften

Set

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Einfügen von Elementen ist **einfach**
- Test, ob ein Element in der Menge ist, ist **einfach**

Mengen Implementierung: HashSet, TreeSet

Eigenschaften

Set

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Einfügen von Elementen ist **einfach**
- Test, ob ein Element in der Menge ist, ist **einfach**

SortedSet

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Einfügen von Elementen ist **einfach**
- Test, ob ein Element in der Menge ist, ist **einfach**
- Sortierung ist **einfach** (implizit)
- Erstes und letztes Element bezüglich der Sortierung ist **einfach** (implizit)
- Bildung von Teilmengen ist **einfach**

Mengen Implementierung: HashSet, TreeSet

java.util.HashSet

als Implementierung von

- Set

java.util.TreeSet

als Implementierung von

- Set
- SortedSet

Abbildungen

Datenstruktur **Abbildung**
(`java.util.Map`):

- Schlüssel – Werte Paare:
  jedem Schlüssel wird eindeutig ein Wert zugeordnet
- Keine Duplikate der Schlüssel
- Duplikate der Werte möglich
- Keine Ordnung
- Keine Teilmengen der Abbildung

Abbildungen

Datenstruktur **Abbildung** (`java.util.Map`):

- Schlüssel – Werte Paare: jedem Schlüssel wird eindeutig ein Wert zugeordnet
- Keine Duplikate der Schlüssel
- Duplikate der Werte möglich
- Keine Ordnung
- Keine Teilmengen der Abbildung

Schlüssel – Werte Paare: `java.util.Map.Entry<K,V>`

- `K getKey()`
- `V getValue()`

Abbildungen

Datenstruktur **Abbildung** (`java.util.Map`):

- Schlüssel – Werte Paare: jedem Schlüssel wird eindeutig ein Wert zugeordnet
- Keine Duplikate der Schlüssel
- Duplikate der Werte möglich
- Keine Ordnung
- Keine Teilmengen der Abbildung

Datenstruktur **sortierte Abbildung** (`java.util.SortedMap`):

- Schlüssel – Werte Paare: jedem Schlüssel wird eindeutig ein Wert zugeordnet
- Keine Duplikate der Schlüssel
- Duplikate der Werte möglich
- Ordnung (Sortierung) auf den **Schlüsseln**
  - Kleinstes Element (`firstKey()`)
  - Größtes Element (`lastKey()`)
- Teilmenge (`subMap(K from, K to)`)

Mengen Implementierung: HashMap, TreeMap

Eigenschaften

Map

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Einfügen von Elementen ist **einfach**
- Test, ob ein Schlüssel in der Abbildung ist, ist **einfach**

Mengen Implementierung: HashMap, TreeMap

Eigenschaften

Map

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Einfügen von Elementen ist **einfach**
- Test, ob ein Schlüssel in der Abbildung ist, ist **einfach**

SortedMap

- Größe muss beim Anlegen **nicht feststehen**
- Größe kann **verändert** werden
- Einfügen von Elementen ist **einfach**
- Test, ob ein Schlüssel in der Abbildung ist, ist **einfach**
- Sortierung nach Schlüsseln ist **einfach** (implizit)
- Erster und letzter Schlüssel bezüglich der Sortierung ist **einfach** (implizit)
- Bildung von Teilmengen ist **einfach**

Abbildungen Implementierung: HashMap, TreeMap

java.util.HashMap

als Implementierung von

- Map

Abbildungen Implementierung: HashMap, TreeMap

java.util.HashMap

als Implementierung von

- Map

java.util.TreeMap

als Implementierung von

- Map
- SortedMap