Algorithmen und Datenstrukturen

SS 2016

Dominik S. Kaaser

4. Mai 2016

Teil I

Java Collections Framework

Wiederholung
- Abstrakte Datentypen

Java Collections Framework
- Interfaces
- Listen
- Queue, Stack, Deque
- Sets
- Maps

Wiederholung

Abstrakte Datentypen

Definition (Dynamische Menge)

Eine dynamische Menge ist gegeben durch eine oder mehrere **Mengen von Objekten** sowie **Operationen auf diesen Mengen** und den Objekten der Mengen. Dynamische Mengen werden auch **abstrakte Datentypen** (ADT) genannt.

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Wiederholung

Abstrakte Datentypen

Definition (Dynamische Menge)

Eine dynamische Menge ist gegeben durch eine oder mehrere **Mengen von Objekten** sowie **Operationen auf diesen Mengen** und den Objekten der Mengen. Dynamische Mengen werden auch **abstrakte Datentypen** (ADT) genannt.

Definition (Wörterbuch, Warteschlange)

1. Werden auf einer Menge von Objekten die Operationen **Einfügen**, **Entfernen** und **Suchen** betrachtet, so spricht man von einem **Wörterbuch**.
2. Werden auf einer Menge von Objekten die Operationen **Einfügen**, **Entfernen** und **Suchen des Maximums** betrachtet, so spricht man von einer **Warteschlange**.

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Abstrakte Datentypen

Collections Libraries

Problemstellung

- Effizientes Verwalten einer dynamischen Menge von Objekten
- Operationen zum Einfügen, Entfernen, Suchen

- Es handelt sich um ein gelöstes Problem!
  - C++ – Standard Template Library (STL)
  - C# – .NET System.Collections
  - Java – Java Collections Framework
- Fertige Programmbibliotheken sind performanter
- Reduzierter Aufwand bei Design und Entwicklung dank Vereinheitlichungen
  - Zusammenarbeit verschiedener entwickelnder APIs
  - Einfache Verwendung neuer APIs
  - Einfaches Design neuer APIs
  - Wiederverwendung allgemeiner Datenstrukturen

Abstrakte Datentypen

Collections Libraries

Problemstellung

- Effizientes Verwalten einer dynamischen Menge von Objekten
- Operationen zum Einfügen, Entfernen, Suchen

- Es handelt sich um ein gelöstes Problem!
  - C++ – Standard Template Library (STL)
  - C# – .NET System.Collections
  - Java – Java Collections Framework
- Fertige Programmbibliotheken sind performanter
- Reduzierter Aufwand bei Design und Entwicklung dank Vereinheitlichungen
  - Zusammenarbeit verschiedener entwickelter APIs
  - Einfache Verwendung neuer APIs
  - Einfaches Design neuer APIs
  - Wiederholung typischer Datenstrukturen

Abstrakte Datentypen

Collections Libraries

Problemstellung

- Effizientes Verwalten einer dynamischen Menge von Objekten
- Operationen zum Einfügen, Entfernen, Suchen

- Es handelt sich um ein gelöstes Problem!
  - C++ – Standard Template Library (STL)
  - C# – .NET System.Collections
  - Java – Java Collections Framework
- Fertige Programmbibliotheken sind performanter
- Reduzierter Aufwand bei Design und Entwicklung dank Vereinheitlichungen
  - Zusammenarbeit verschiedener Entwickler mit APIs
  - Einfache Verwendung neuer APIs
  - Einfaches Design neuer APIs
  - Wiederverwendung neuer Datenstrukturen

Abstrakte Datentypen

Collections Libraries

Problemstellung

- Effizientes Verwalten einer dynamischen Menge von Objekten
- Operationen zum Einfügen, Entfernen, Suchen

- Es handelt sich um ein gelöstes Problem!
  - C++ – Standard Template Library (STL)
  - C# – .NET System.Collections
  - Java – Java Collections Framework
- Fertige Programmbibliotheken sind performanter
- Reduzierter Aufwand bei Design und Entwicklung dank Vereinheitlichungen
  - Zusammenarbeit verschiedener APIs
  - Einfache Verwendung neuer APIs
  - Einfaches Design neuer APIs
  - Wiederverwendung neuer Datenstrukturen

Abstrakte Datentypen

Collections Libraries

Problemstellung

- Effizientes Verwalten einer dynamischen Menge von Objekten
- Operationen zum Einfügen, Entfernen, Suchen

- Es handelt sich um ein gelöstes Problem!
  - C++ – Standard Template Library (STL)
  - C# – .NET System.Collections
  - Java – Java Collections Framework
- Fertige Programmbibliotheken sind performanter
- Reduzierter Aufwand bei Design und Entwicklung dank Vereinheitlichungen
  - Zusammenarbeit verschiedener Entwickler an APIs
  - Einfache Verwendung neuer APIs
  - Einfaches Design neuer APIs
  - Wiederverwendung neuer Datenstrukturen

Abstrakte Datentypen

Collections Libraries

Problemstellung

- Effizientes Verwalten einer dynamischen Menge von Objekten
- Operationen zum Einfügen, Entfernen, Suchen

- Es handelt sich um ein gelöstes Problem!
  - C++ – Standard Template Library (STL)
  - C# – .NET System.Collections
  - Java – Java Collections Framework
- Fertige Programmbibliotheken sind performanter
- Reduzierter Aufwand bei Design und Entwicklung dank Vereinheitlichungen
  - Zusammenarbeit zwischen verschiedenen APIs
  - Einfache Verwendung neuer APIs
  - Einfaches Design neuer APIs
  - Wiederverwendung von Datenstrukturen

Abstrakte Datentypen

Collections Libraries

Problemstellung

- Effizientes Verwalten einer dynamischen Menge von Objekten
- Operationen zum Einfügen, Entfernen, Suchen

- Es handelt sich um ein gelöstes Problem!
  - C++ – Standard Template Library (STL)
  - C# – .NET System.Collections
  - Java – Java Collections Framework
- Fertige Programmbibliotheken sind performanter
- Reduzierter Aufwand bei Design und Entwicklung dank Vereinheitlichungen
  - Zusammenarbeit zwischen verschiedenen APIs
  - Einfache Verwendung neuer APIs
  - Einfaches Design neuer APIs
  - Kompatibilität eigener Datenstrukturen

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Abstrakte Datentypen

Collections Libraries

Problemstellung

- Effizientes Verwalten einer dynamischen Menge von Objekten
- Operationen zum Einfügen, Entfernen, Suchen

- Es handelt sich um ein gelöstes Problem!
  - C++ – Standard Template Library (STL)
  - C# – .NET System.Collections
  - Java – Java Collections Framework
- Fertige Programmbibliotheken sind performanter
- Reduzierter Aufwand bei Design und Entwicklung dank Vereinheitlichungen
  - Zusammenarbeit zwischen verschiedenen APIs
  - Einfache Verwendung neuer APIs
  - Einfaches Design neuer APIs
  - Kompatibilität eigener Datenstrukturen

Abstrakte Datentypen

Collections Libraries

Problemstellung

- Effizientes Verwalten einer dynamischen Menge von Objekten
- Operationen zum Einfügen, Entfernen, Suchen

- Es handelt sich um ein gelöstes Problem!
  - C++ – Standard Template Library (STL)
  - C# – .NET System.Collections
  - Java – Java Collections Framework
- Fertige Programmbibliotheken sind performanter
- Reduzierter Aufwand bei Design und Entwicklung dank Vereinheitlichungen
  - Zusammenarbeit zwischen verschiedenen APIs
  - Einfache Verwendung neuer APIs
  - Einfaches Design neuer APIs
  - Kompatibilität eigener Datenstrukturen

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Abstrakte Datentypen

Collections Libraries

Problemstellung

- Effizientes Verwalten einer dynamischen Menge von Objekten
- Operationen zum Einfügen, Entfernen, Suchen

- Es handelt sich um ein gelöstes Problem!
  - C++ – Standard Template Library (STL)
  - C# – .NET System.Collections
  - Java – Java Collections Framework
- Fertige Programmbibliotheken sind performanter
- Reduzierter Aufwand bei Design und Entwicklung dank Vereinheitlichungen
  - Zusammenarbeit zwischen verschiedenen APIs
  - Einfache Verwendung neuer APIs
  - Einfaches Design neuer APIs
  - Kompatibilität eigener Datenstrukturen

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Abstrakte Datentypen

Collections Libraries

Problemstellung

- Effizientes Verwalten einer dynamischen Menge von Objekten
- Operationen zum Einfügen, Entfernen, Suchen

- Es handelt sich um ein gelöstes Problem!
  - C++ – Standard Template Library (STL)
  - C# – .NET System.Collections
  - Java – Java Collections Framework
- Fertige Programmbibliotheken sind performanter
- Reduzierter Aufwand bei Design und Entwicklung dank Vereinheitlichungen
  - Zusammenarbeit zwischen verschiedenen APIs
  - Einfache Verwendung neuer APIs
  - Einfaches Design neuer APIs
  - Kompatibilität eigener Datenstrukturen

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Java Collections Framework

Definition (Collection)

Eine Collection, auch Container, ist ein Objekt, das mehrere Elemente gruppiert. Container können Elemente speichern, suchen, modifizieren sowie aggregierte Daten liefern.

- Repräsentation natürlicher Gruppierungen
  - Figuren in einem Spiel
  - Gepackte Arbeitsaufträge
  - Zuweisung von Telefonnummern zu Namen in einem Telefonbuch
- Wir betrachten im Folgenden die Bestandteile des Java Collections Framework.

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Java Collections Framework

Definition (Collection)

Eine Collection, auch Container, ist ein Objekt, das mehrere Elemente gruppiert. Container können Elemente speichern, suchen, modifizieren sowie aggregierte Daten liefern.

- Repräsentation natürlicher Gruppierungen
  - Figuren in einem Spiel
  - Geplante Arbeitsaufträge
  - Zuweisung von Telefonnummern zu Namen in einem Telefonbuch
- Wir betrachten im Folgenden die Bestandteile des Java Collections Framework.

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Java Collections Framework

Definition (Collection)

Eine Collection, auch Container, ist ein Objekt, das mehrere Elemente gruppiert. Container können Elemente speichern, suchen, modifizieren sowie aggregierte Daten liefern.

- Repräsentation natürlicher Gruppierungen
  - Figuren in einem Spiel
  - Geplante Arbeitsaufträge
  - Zuweisung von Telefonnummern zu Namen in einem Telefonbuch
- Wir betrachten im Folgenden die Bestandteile des Java Collections Framework.

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Java Collections Framework

Definition (Collection)

Eine Collection, auch Container, ist ein Objekt, das mehrere Elemente gruppiert. Container können Elemente speichern, suchen, modifizieren sowie aggregierte Daten liefern.

- Repräsentation natürlicher Gruppierungen
  - Figuren in einem Spiel
  - Geplante Arbeitsaufträge
  - Zuweisung von Telefonnummern zu Namen in einem Telefonbuch
- Wir betrachten im Folgenden die Bestandteile des Java Collections Framework.

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Java Collections Framework

Definition (Collection)

Eine Collection, auch Container, ist ein Objekt, das mehrere Elemente gruppiert. Container können Elemente speichern, suchen, modifizieren sowie aggregierte Daten liefern.

- Repräsentation natürlicher Gruppierungen
  - Figuren in einem Spiel
  - Geplante Arbeitsaufträge
  - Zuweisung von Telefonnummern zu Namen in einem Telefonbuch

- Wir betrachten im Folgenden die Bestandteile des Java Collections Framework.

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Java Collections Framework

Definition (Collection)

Eine Collection, auch Container, ist ein Objekt, das mehrere Elemente gruppiert. Container können Elemente speichern, suchen, modifizieren sowie aggregierte Daten liefern.

- Repräsentation natürlicher Gruppierungen
  - Figuren in einem Spiel
  - Geplante Arbeitsaufträge
  - Zuweisung von Telefonnummern zu Namen in einem Telefonbuch

Definition (Collections Framework)

Ein Collections Framework ist eine einheitliche Architektur für die Darstellung und Verwaltung von Containern.

- Wir betrachten im Folgenden die Bestandteile des Java Collections Framework.

Java Collections Framework

Definition (Collection)

Eine Collection, auch Container, ist ein Objekt, das mehrere Elemente gruppiert. Container können Elemente speichern, suchen, modifizieren sowie aggregierte Daten liefern.

- Repräsentation natürlicher Gruppierungen
  - Figuren in einem Spiel
  - Geplante Arbeitsaufträge
  - Zuweisung von Telefonnummern zu Namen in einem Telefonbuch

Definition (Collections Framework)

Ein Collections Framework ist eine einheitliche Architektur für die Darstellung und Verwaltung von Containern.

- Wir betrachten im Folgenden die Bestandteile des Java Collections Framework.

Interfaces und Klassen

Überblick

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `containsAll(Collection<?> c)`
  - `addAll(Collection<? extends E> c)`
  - `removeAll(Collection<?> c)`
  - `retainAll(Collection<?> c)`
  - `clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

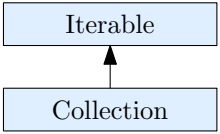
- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interface Collection

- Collections verwalten eine Gruppe von Objekten.
- Die Interfaces sind generisch (`public interface Collection<E>...`).
- Grundlegende Operationen
  - `int size()`
  - `boolean isEmpty()`
  - `boolean contains(Object element)`
  - `boolean add(E element)`
  - `boolean remove(Object element)`
  - `Iterator<E> iterator()`
- Operationen auf Basis gesamter Collections
  - `boolean containsAll(Collection<?> c)`
  - `boolean addAll(Collection<? extends E> c)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `void clear()`
- Alle Methoden liefern true zurück, falls der Container beim Aufruf verändert wurde.
- Zusätzlich existieren Operationen zum Konvertieren von und zu Arrays.

Interfaces und Klassen

Überblick



Iteratoren

- Iteratoren sind kleine Objekte, mit deren Hilfe ein Container durchlaufen wird.

Iteratoren

- Iteratoren sind kleine Objekte, mit deren Hilfe ein Container durchlaufen wird.

```
public interface Iterator<E> {
  boolean hasNext(); // liefert true, falls es noch weitere Elemente gibt
  E next(); // liefert das naechste Element
  void remove(); //entfernt das letzte zuvor zurueckgelieferte Element
}
```

Iteratoren

- Iteratoren sind kleine Objekte, mit deren Hilfe ein Container durchlaufen wird.

```
public interface Iterator<E> {
  boolean hasNext(); // liefert true, falls es noch weitere Elemente gibt
  E next(); // liefert das naechste Element
  void remove(); //entfernt das letzte zuvor zurueckgelieferte Element
}
```

- Klassische Iterator-Schleife

```
public void
test(Collection<String> c) {
  Iterator<String> it = c.iterator();
  while(it.hasNext())
    System.out.println(it.next());
}
```

Iteratoren

- Iteratoren sind kleine Objekte, mit deren Hilfe ein Container durchlaufen wird.

```
public interface Iterator<E> {
  boolean hasNext(); // liefert true, falls es noch weitere Elemente gibt
  E next(); // liefert das naechste Element
  void remove(); //entfernt das letzte zuvor zurueckgelieferte Element
}
```

- Klassische Iterator-Schleife

```
public void
test(Collection<String> c) {
  Iterator<String> it = c.iterator();
  while(it.hasNext())
    System.out.println(it.next());
}
```

- Verwendung in der for-each-Schleife

```
public void
test(Collection<String> c) {
  for(String entry : c)
    System.out.println(entry);
}
```

Konvertierungskonstruktoren

Konstruktoren

Implementierungen enthalten einen Konvertierungskonstruktor.

Konvertierungskonstruktoren

Konstruktoren

Implementierungen enthalten einen Konvertierungskonstruktor.

```
public class CollectionsConstructorTest {
  public void test(Collection<String> someCollection) {

    List<String> list = new ArrayList<String>(someCollection);

  }
}
```

Konvertierungskonstruktoren

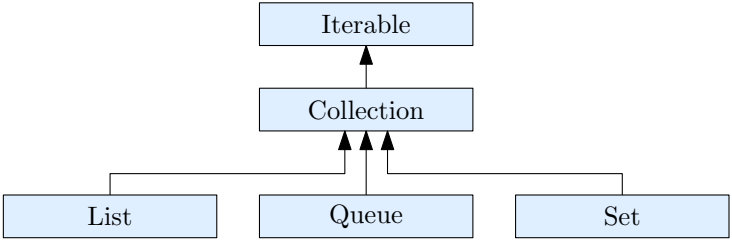
Konstruktoren

Implementierungen enthalten einen Konvertierungskonstruktor.

```
public class CollectionsConstructorTest {
  public void test(Collection<String> someCollection) {

    List<String> list = new ArrayList<>(someCollection); // Java 7

  }
}
```

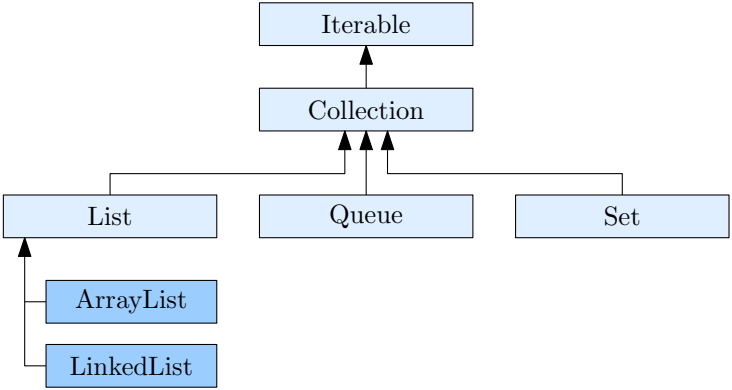
Interfaces und Klassen

Überblick

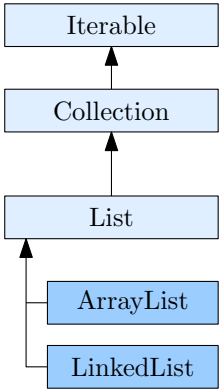


Interfaces und Klassen

Überblick

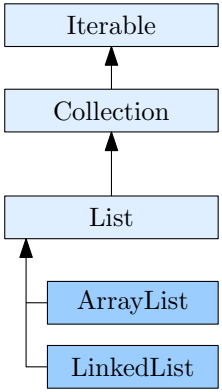


Listen



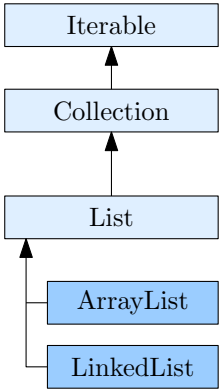
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
- Suchen in der Regel teuer, $O(n)$
- Bidirektionaler Iterator `ListIterator`

Listen



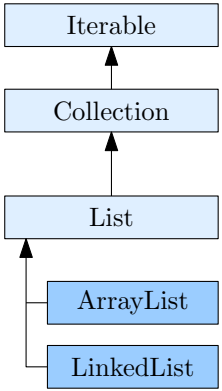
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
- Suchen in der Regel teuer, $O(n)$
- Bidirektionaler Iterator `ListIterator`

Listen



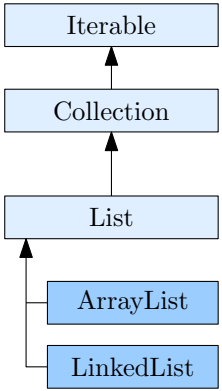
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
- Suchen in der Regel teuer, $O(n)$
- Bidirektionaler Iterator `ListIterator`

Listen



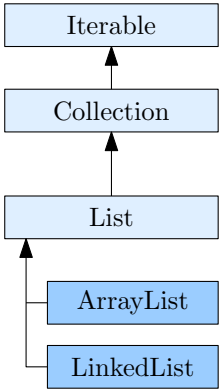
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
  - `void add(int index, E element)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `E remove(int index)`
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
  - `List<E> subList(int fromIndex, int toIndex)`
- Suchen in der Regel teuer, $O(n)$
  - `int indexOf(Object o)`
  - `int lastIndexOf(Object o)`
- Bidirektionaler Iterator `ListIterator`

Listen



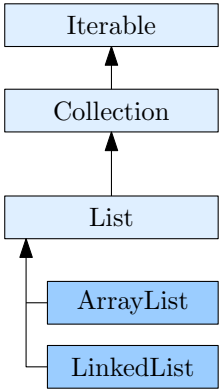
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
  - `void add(int index, E element)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `E remove(int index)`
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
- Suchen in der Regel teuer, $O(n)$
- Bidirektionaler Iterator `ListIterator`

Listen



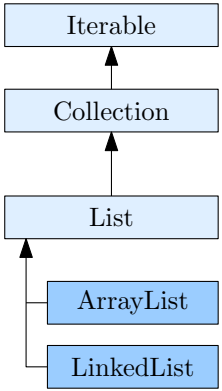
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
  - `void add(int index, E element)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `E remove(int index)`
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
- Suchen in der Regel teuer, $O(n)$
- Bidirektionaler Iterator `ListIterator`

Listen



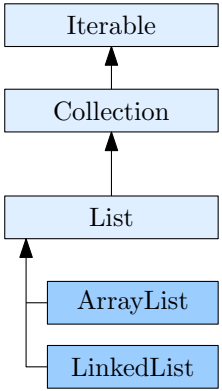
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
  - `void add(int index, E element)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `E remove(int index)`
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
- Suchen in der Regel teuer, $O(n)$
- Bidirektionaler Iterator `ListIterator`

Listen



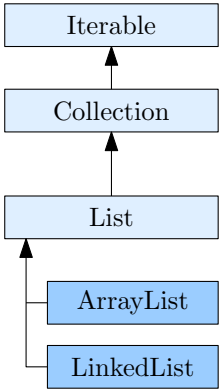
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
  - `void add(int index, E element)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `E remove(int index)`
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
- Suchen in der Regel teuer, $O(n)$
- Bidirektionaler Iterator `ListIterator`

Listen



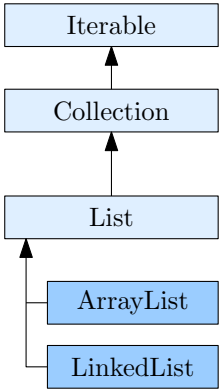
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
  - `void add(int index, E element)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `E remove(int index)`
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
  - `List<E> subList(int fromIndex, int toIndex)`
- Suchen in der Regel teuer, $O(n)$
  - `int indexOf(Object o)`
  - `int lastIndexOf(Object o)`
- Bidirektionaler Iterator `ListIterator`

Listen



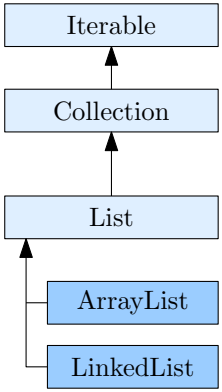
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
  - `void add(int index, E element)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `E remove(int index)`
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
  - `List<E> subList(int fromIndex, int toIndex)`
- Suchen in der Regel teuer, $O(n)$
- Bidirektionaler Iterator `ListIterator`

Listen



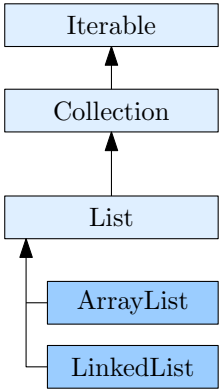
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
  - `void add(int index, E element)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `E remove(int index)`
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
  - `List<E> subList(int fromIndex, int toIndex)`
- Suchen in der Regel teuer, $O(n)$
  - `int indexOf(Object o)`
  - `int lastIndexOf(Object o)`
- Bidirektionaler Iterator `ListIterator`

Listen



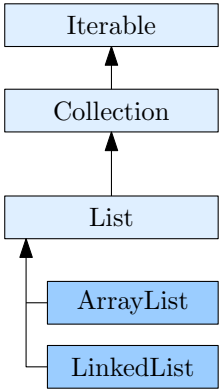
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
  - `void add(int index, E element)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `E remove(int index)`
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
  - `List<E> subList(int fromIndex, int toIndex)`
- Suchen in der Regel teuer, $\mathrm{O}(n)$
  - `int indexOf(Object o)`
  - `int lastIndexOf(Object o)`
- Bidirektionaler Iterator `ListIterator`

Listen



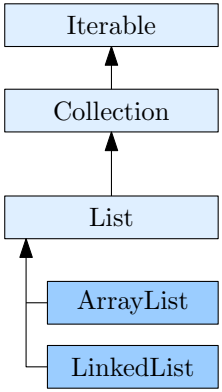
- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
  - `void add(int index, E element)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `E remove(int index)`
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
  - `List<E> subList(int fromIndex, int toIndex)`
- Suchen in der Regel teuer, $\mathrm{O}(n)$
  - `int indexOf(Object o)`
  - `int lastIndexOf(Object o)`
- Bidirektionaler Iterator `ListIterator`

Listen



- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
  - `void add(int index, E element)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `E remove(int index)`
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
  - `List<E> subList(int fromIndex, int toIndex)`
- Suchen in der Regel teuer, $\mathrm{O}(n)$
  - `int indexOf(Object o)`
  - `int lastIndexOf(Object o)`
- Bidirektionaler Iterator `ListIterator`

Listen



- Sequenz von Daten mit fixer Reihenfolge
- Implementierungen `ArrayList` und `LinkedList`
- Kontrolle über die Position der einzufügenden Elemente
- Zugriff mittels Integer-Index möglich (0-indiziert)
  - `void add(int index, E element)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `E remove(int index)`
- Erlauben mehrfach vorkommende Elemente
- Zugriff auf Teilbereiche mit
  - `List<E> subList(int fromIndex, int toIndex)`
- Suchen in der Regel teuer, $\mathrm{O}(n)$
  - `int indexOf(Object o)`
  - `int lastIndexOf(Object o)`
- Bidirektionaler Iterator `ListIterator`

Listen

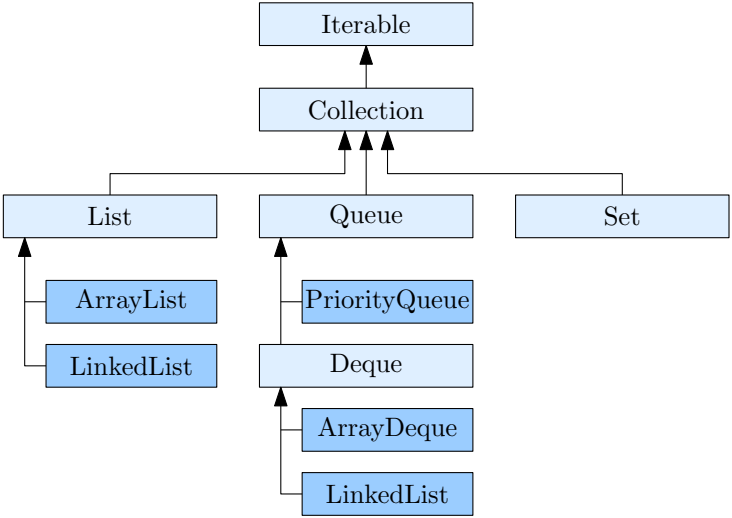
Beispiel

ListDemo.java

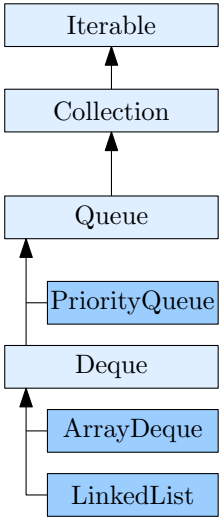
```
import java.util.Scanner;
import java.util.List;
import java.util.LinkedList;

public class ListDemo {
  public static void main(String[] args) {

    List<String> list = new LinkedList<>();
    Scanner s = new Scanner(System.in);
    String line;

    while (!(line = s.nextLine()).equals(""))
      list.add(line);

    System.out.println("\nListe:");
    for (String element : list)
      System.out.println(element);

    System.out.println();

    String search;
    while (!(search = s.nextLine()).equals("")) {
      if (list.contains(search)) {
        int position = list.indexOf(search);
        System.out.println(search + " gefunden, position " + position);
      } else
        System.out.println(search + " nicht gefunden.");
    }

    s.close();
  }
}
```

Interfaces und Klassen

Überblick

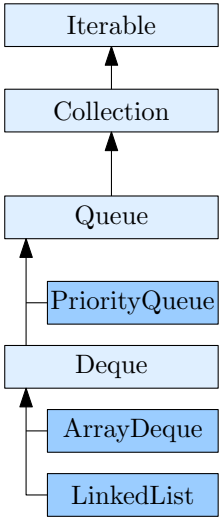


Queue and Deque



- Interfaces für Warteschlangen
  - FIFO-Queue
  - PriorityQueue
- Queues (FIFOs) und Stacks (LIFOs) können als Deques (double-ended queues) implementiert werden.
- Implementierungen `ArrayDeque` und `LinkedList`

Queue and Deque

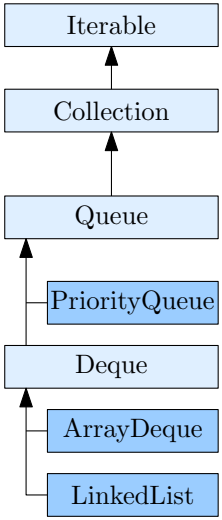


- Interfaces für Warteschlangen
  - FIFO-Queue
  - PriorityQueue
- Beispiel Prioritätswarteschlange

```
public class PriorityQueueDemo {
  public static <E> List<E> sort(Collection<E> c) {

    Queue<E> queue = new PriorityQueue<E>(c);
    List<E> result = new LinkedList<E>();

    while (!queue.isEmpty())
      result.add(queue.remove());

    return result;

  }
}
```

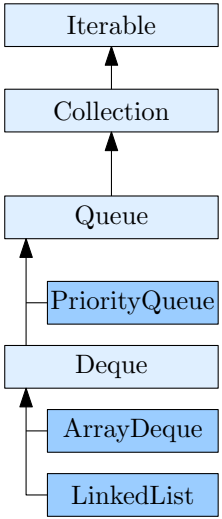
- Queues (FIFOs) und Stacks (LIFOs) können als Deques

Queue and Deque



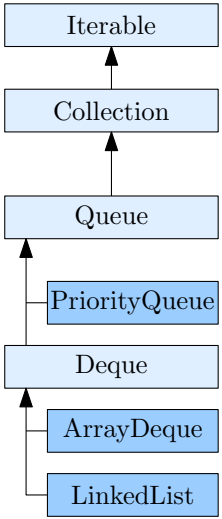
- Interfaces für Warteschlangen
  - FIFO-Queue
  - PriorityQueue
- Queues (FIFOs) und Stacks (LIFOs) können als Deques (double-ended queues) implementiert werden.
- Implementierungen `ArrayDeque` und `LinkedList`

Queue and Deque



- Interfaces für Warteschlangen
  - FIFO-Queue
  - PriorityQueue
- Queues (FIFOs) und Stacks (LIFOs) können als Deques (double-ended queues) implementiert werden.
- Implementierungen `ArrayDeque` und `LinkedList`

Queue and Deque



- Interfaces für Warteschlangen
  - FIFO-Queue
  - PriorityQueue
- Queues (FIFOs) und Stacks (LIFOs) können als Deques (double-ended queues) implementiert werden.
- Implementierungen `ArrayDeque` und `LinkedList`

Operation	Erstes Element	Letztes Element
Insert	`addFirst(e)`, `offerFirst(e)`	`addLast(e)`, `offerLast(e)`
Remove	`removeFirst()`, `pollFirst()`	`removeLast()`, `pollLast()`
Examine	`getFirst()`, `peekFirst()`	`getLast()`, `peekLast()`

HeapSort mit PriorityQueue

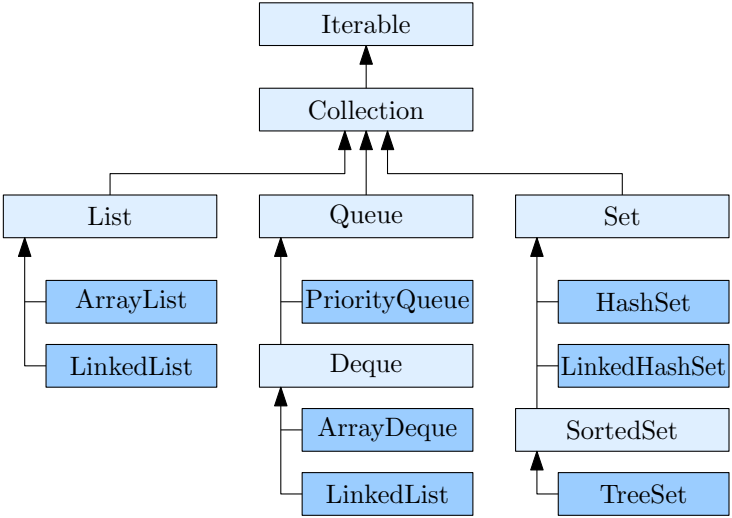
Beispiel

PriorityQueueDemo.java

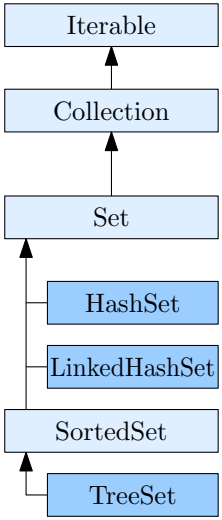
```
import java.util.Arrays;
import java.util.Collection;
import java.util.List;
import java.util.Queue;
import java.util.PriorityQueue;
import java.util.LinkedList;

public class PriorityQueueDemo {
  public static <E> List<E> sort(Collection<E> c) {

    Queue<E> queue = new PriorityQueue<E>(c);
    List<E> result = new LinkedList<E>();

    while (!queue.isEmpty())
      result.add(queue.remove());

    return result;

  }

  public static void main(String[] args) {
    List<String> argsList = Arrays.asList(args);
    List<String> sortedList = sort(argsList);
    for (String element : sortedList)
      System.out.println(element);
  }
}
```

Interfaces und Klassen

Überblick



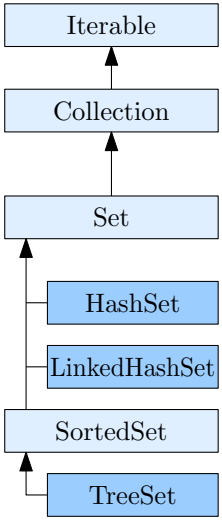
Set



- Sets modellieren mathematische Mengen
- Keine mehrfach vorkommenden Elemente
- Implementierungen:
  - HashSet: unsortiert, basierend auf Hashtabelle
  - LinkedHashSet: wie HashSet, zusätzlich verkettete Liste
  - SortedSet: sortiert, basierend auf Vergleichsoperator
- Spezialfall SortedSet
  - Sortierte Menge
  - Schnitt, Vereinigung und Differenz effizient
  - Binäre Suchbäume
- Beispiele
  - SetDemo
  - SetTiming
  - SortedSetDemo

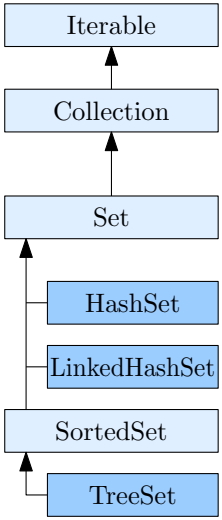
Wiederholung
Abstrakte Datentypen
Java Collections Framework
Interfaces
Listen
Queue, Stack, Deque
Sets
Maps

Set



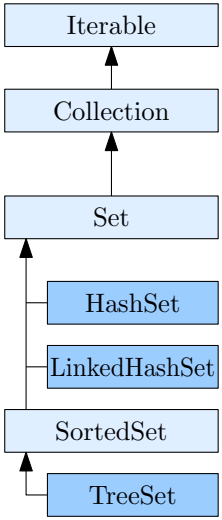
- Sets modellieren mathematische Mengen
- Keine mehrfach vorkommenden Elemente
- Implementierungen:
  - `HashSet` – verwendet Hashtabelle
  - `TreeSet` – verwendet Suchbaum
  - `LinkedHashSet` – verwendet zusätzlich verkettete Liste
- Spezialfall `SortedSet`
  - Sortierte Menge
  - Schnelle Zugriff auf größtes und kleinstes Element
  - Bereichsanfragen
- Beispiele
  - `SetDemo`
  - `SetTiming`
  - `SortedSetDemo`

Set



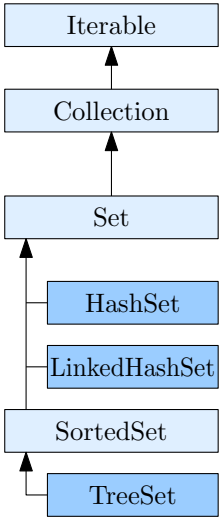
- Sets modellieren mathematische Mengen
- Keine mehrfach vorkommenden Elemente
- Implementierungen:
  - `HashSet` – verwendet Hashtabelle
  - `TreeSet` – verwendet Suchbaum-Struktur
  - `LinkedHashSet` – verwendet zusätzliche verzeigerter Liste
- Spezialfall `SortedSet`
  - Sortierte Menge
  - Schnitt, Vereinigung und Differenz effizient
  - Bereichsanfragen
- Beispiele
  - `SetDemo`
  - `SetTiming`
  - `SortedSetDemo`

Set



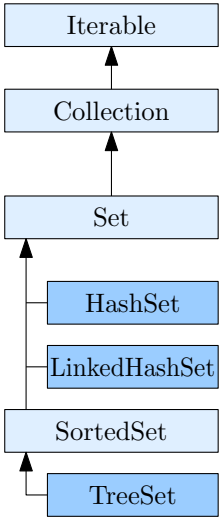
- Sets modellieren mathematische Mengen
- Keine mehrfach vorkommenden Elemente
- Implementierungen:
  - `HashSet` – verwendet Hashtabelle
  - `TreeSet` – verwendet Suchbaum-Struktur
  - `LinkedHashSet` – verwendet zusätzliche verzeigerter Liste
- Spezialfall `SortedSet`
  - Elemente müssen
  - Schnitt, Vereinigt und Differenz sind effizient
  - Beschränkungen
- Beispiele
  - `SetDemo`
  - `SetTiming`
  - `SortedSetDemo`

Set



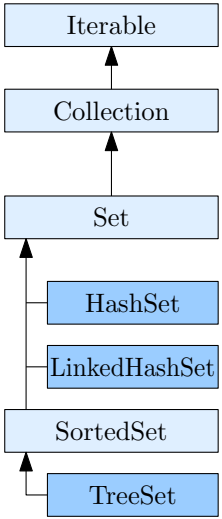
- Sets modellieren mathematische Mengen
- Keine mehrfach vorkommenden Elemente
- Implementierungen:
  - `HashSet` – verwendet Hashtabelle
  - `TreeSet` – verwendet Suchbaum-Struktur
  - `LinkedHashSet` – verwendet zusätzliche verzeigerter Liste
- Spezialfall `SortedSet`
- Beispiele
  - `SetDemo`
  - `SetTiming`
  - `SortedSetDemo`

Set



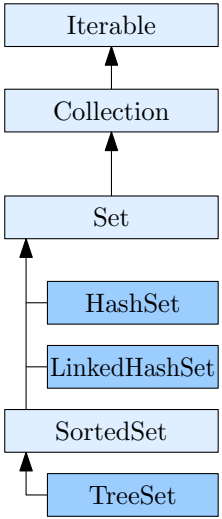
- Sets modellieren mathematische Mengen
- Keine mehrfach vorkommenden Elemente
- Implementierungen:
  - `HashSet` – verwendet Hashtabelle
  - `TreeSet` – verwendet Suchbaum-Struktur
  - `LinkedHashSet` – verwendet zusätzliche verzeigerter Liste
- Spezialfall `SortedSet`
  - Sortierte Menge
  - Schnitt, Vereinigt, Differenz und Minimum, Maximum
  - Bereichsanfragen
- Beispiele
  - `SetDemo`
  - `SetTiming`
  - `SortedSetDemo`

Set



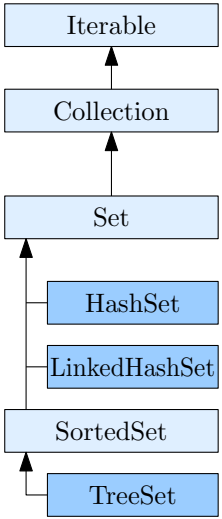
- Sets modellieren mathematische Mengen
- Keine mehrfach vorkommenden Elemente
- Implementierungen:
  - `HashSet` – verwendet Hashtabelle
  - `TreeSet` – verwendet Suchbaum-Struktur
  - `LinkedHashSet` – verwendet zusätzliche verzeigerter Liste
- Spezialfall `SortedSet`
  - Sortierte Menge
  - Erlaubt Zugriff auf größtes und kleinstes Element
  - Bereichsabfragen
- Beispiele
  - `SetDemo`
  - `SetTiming`
  - `SortedSetDemo`

Set



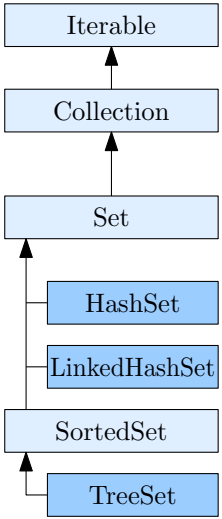
- Sets modellieren mathematische Mengen
- Keine mehrfach vorkommenden Elemente
- Implementierungen:
  - `HashSet` – verwendet Hashtabelle
  - `TreeSet` – verwendet Suchbaum-Struktur
  - `LinkedHashSet` – verwendet zusätzliche verzeigerter Liste
- Spezialfall `SortedSet`
  - Sortierte Menge
  - Erlaubt Zugriff auf größtes und kleinstes Element
  - Bereichsabfragen
- Beispiele
  - `SetDemo`
  - `SetTiming`
  - `SortedSetDemo`

Set



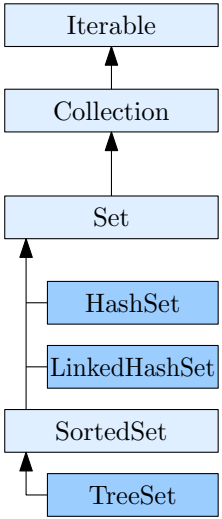
- Sets modellieren mathematische Mengen
- Keine mehrfach vorkommenden Elemente
- Implementierungen:
  - `HashSet` – verwendet Hashtabelle
  - `TreeSet` – verwendet Suchbaum-Struktur
  - `LinkedHashSet` – verwendet zusätzliche verzeigerter Liste
- Spezialfall `SortedSet`
  - Sortierte Menge
  - Erlaubt Zugriff auf größtes und kleinstes Element
  - Bereichsabfragen
- Beispiele
  - `SetDemo`
  - `SetTiming`
  - `SortedSetDemo`

Set



- Sets modellieren mathematische Mengen
- Keine mehrfach vorkommenden Elemente
- Implementierungen:
  - `HashSet` – verwendet Hashtabelle
  - `TreeSet` – verwendet Suchbaum-Struktur
  - `LinkedHashSet` – verwendet zusätzliche verzeigerter Liste
- Spezialfall `SortedSet`
  - Sortierte Menge
  - Erlaubt Zugriff auf größtes und kleinstes Element
  - Bereichsabfragen
- Beispiele
  - `SetDemo`
  - `SetTiming`
  - `SortedSetDemo`

Set



- Sets modellieren mathematische Mengen
- Keine mehrfach vorkommenden Elemente
- Implementierungen:
  - `HashSet` – verwendet Hashtabelle
  - `TreeSet` – verwendet Suchbaum-Struktur
  - `LinkedHashSet` – verwendet zusätzliche verzeigerter Liste
- Spezialfall `SortedSet`
  - Sortierte Menge
  - Erlaubt Zugriff auf größtes und kleinstes Element
  - Bereichsabfragen
- Beispiele
  - `SetDemo`
  - `SetTiming`
  - `SortedSetDemo`

Sets

Beispiel

SetDemo.java

```
import java.util.Scanner;
import java.util.Set;
import java.util.TreeSet;
import java.util.HashSet;
import java.util.LinkedHashSet;
public class SetDemo {
  public static void main(String[] args) {
    Set<String> hashSet = new HashSet<>();
    Set<String> treeSet = new TreeSet<>();
    Set<String> linkedHashSet = new LinkedHashSet<>();
    Scanner s = new Scanner(System.in);
    String line;

    while (!(line = s.nextLine()).equals("")) {
      hashSet.add(line);
      treeSet.add(line);
      linkedHashSet.add(line);
    }
    s.close();

    System.out.print("HashSet:\t");
    for (String element : hashSet)
      System.out.print(element + ", ");
    System.out.println();

    System.out.print("TreeSet:\t");
    for (String element : treeSet)
      System.out.print(element + ", ");
    System.out.println();

    System.out.print("LinkedHashSet:\t");
    for (String element : linkedHashSet)
      System.out.print(element + ", ");
    System.out.println();
  }
}
```

Zeitmessung Implementierungen

Beispiel

SetTiming.java

```
import java.util.Set;
import java.util.TreeSet;
import java.util.HashSet;
import java.util.LinkedHashSet;

public class SetTiming {
  public static void main(String[] args) {
    Set<Integer> hashSet = new HashSet<>();
    Set<Integer> treeSet = new TreeSet<>();
    Set<Integer> linkedHashSet = new LinkedHashSet<>();
    int numberOfInserts = 10000000;
    long startTime;

    startTime = System.currentTimeMillis();
    for (int i = 0; i < numberOfInserts; i++)
      hashSet.add((int) (Math.random() * 1000000.0));
    long timeHashSet = System.currentTimeMillis() - startTime;

    startTime = System.currentTimeMillis();
    for (int i = 0; i < numberOfInserts; i++)
      treeSet.add((int) (Math.random() * 1000000.0));
    long timeTreeSet = System.currentTimeMillis() - startTime;

    startTime = System.currentTimeMillis();
    for (int i = 0; i < numberOfInserts; i++)
      linkedHashSet.add((int) (Math.random() * 1000000.0));
    long timeLinkedHashSet = System.currentTimeMillis() - startTime;

    System.out.println("HashSet: " + timeHashSet + "ms");
    System.out.println("TreeSet: " + timeTreeSet + "ms");
    System.out.println("LinkedHashSet: " + timeLinkedHashSet + "ms");

  }
}
```

Sortierte Mengen

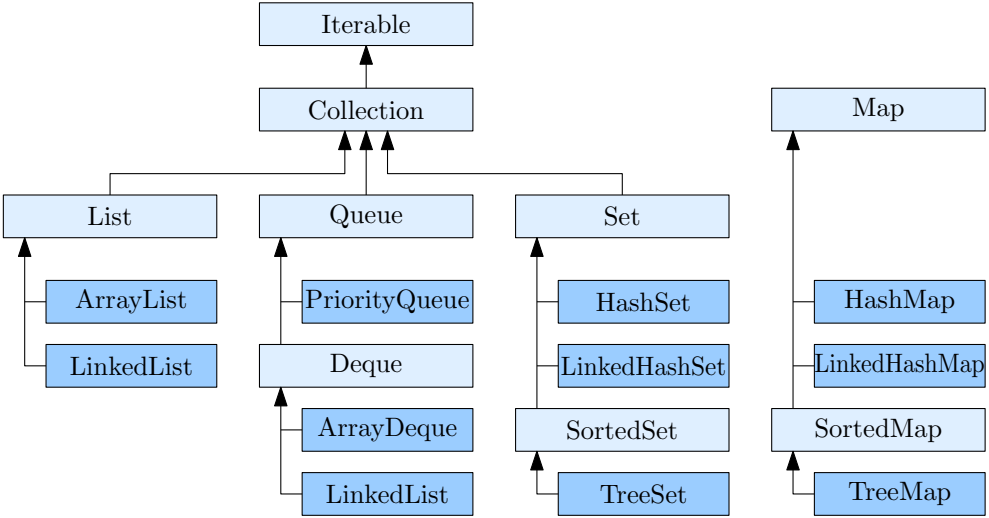
Beispiel

SortedSetDemo.java

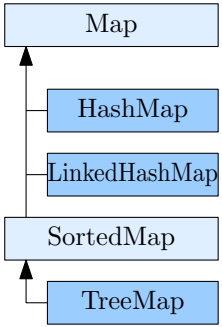
```
import java.util.Scanner;
import java.util.Set;
import java.util.TreeSet;
import java.util.SortedSet;

public class SortedSetDemo {
  public static void main(String[] args) {

    SortedSet<String> set = new TreeSet<>();

    Scanner s = new Scanner(System.in);

    while (s.hasNext())
      set.add(s.next());

    s.close();

    for (char c = 'a'; c <= 'z'; c++) {
      String from = String.valueOf(c);
      String to = String.valueOf((char) (c + 1));

      Set<String> subset = set.subSet(from, to);

      System.out.print(c + " (" + subset.size() + "): ");
      for (String element : subset)
        System.out.print(element + " ");
      System.out.println();
    }

  }
}
```

Interfaces und Klassen

Überblick

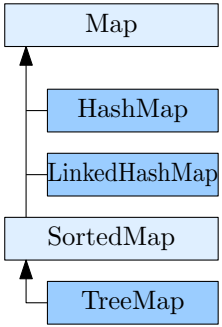


Map I



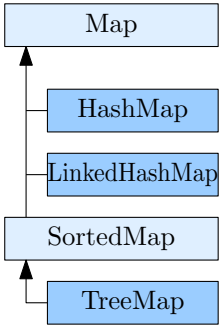
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
- Operationen auf Basis kompletter Maps

Map I



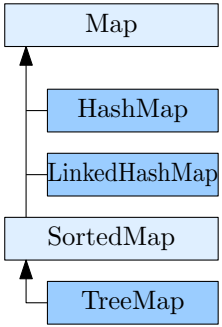
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
- Operationen auf Basis kompletter Maps

Map I



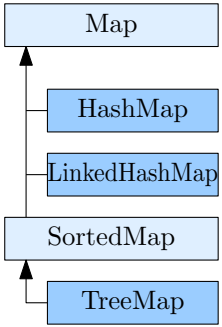
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`
- Operationen auf Basis kompletter Maps
  - `putAll(Map t)`
  - `clear()`

Map I



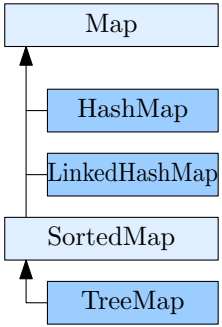
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`
- Operationen auf Basis kompletter Maps

Map I



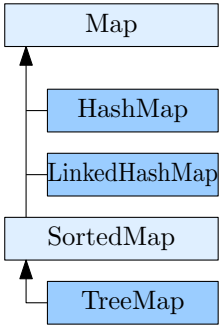
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`
- Operationen auf Basis kompletter Maps

Map I



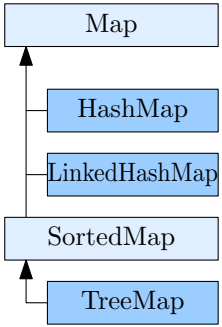
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`
- Operationen auf Basis kompletter Maps

Map I



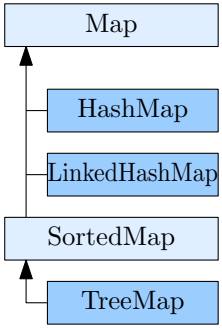
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`
- Operationen auf Basis kompletter Maps

Map I



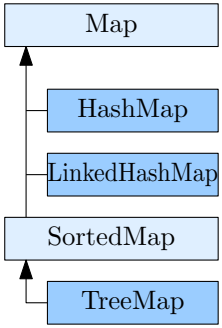
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`
- Operationen auf Basis kompletter Maps

Map I



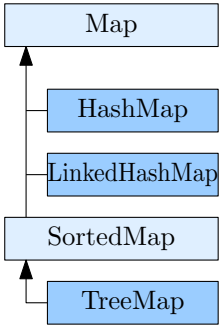
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`
- Operationen auf Basis kompletter Maps

Map I



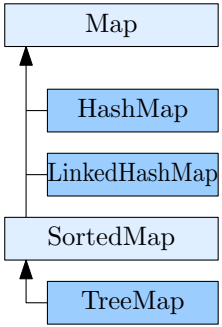
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`
- Operationen auf Basis kompletter Maps

Map I



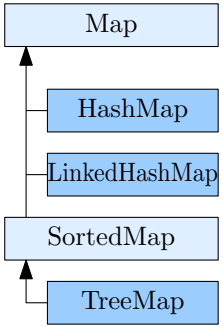
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`
- Operationen auf Basis kompletter Maps
  - `void putAll(Map<? extends K,? extends V> m)`
  - `void clear()`

Map I



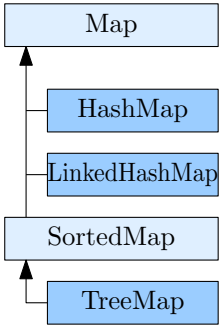
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`
- Operationen auf Basis kompletter Maps
  - `void putAll(Map<? extends K,? extends V> m)`
  - `void clear()`

Map I



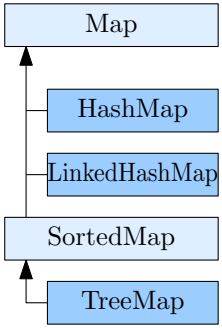
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`
- Operationen auf Basis kompletter Maps
  - `void putAll(Map<? extends K,? extends V> m)`
  - `void clear()`

Map I



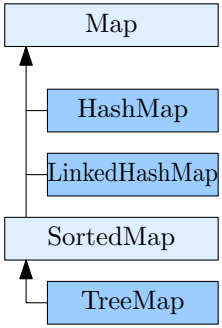
- Klassenhierarchie Parallel zu Collections
- Struktur um Schlüsseln Werte zuzuordnen
- Schlüssel sind eindeutig, vergleichbar mit mathematischen Funktionen
- Grundlegende Operationen
  - `V put(K key, V value)`
  - `V get(Object key)`
  - `V remove(Object key)`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `int size()`
  - `boolean isEmpty()`
- Operationen auf Basis kompletter Maps
  - `void putAll(Map<? extends K,? extends V> m)`
  - `void clear()`

Map II



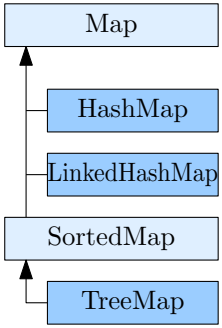
- Zugriff als `Collection`
  - `Set<K> keySet()`
  - `Collection<V> values()`
  - `Set<Map.Entry<K,V>> entrySet()`
- Beispiel
  - `MapDemo`
- MultiMaps nicht im Java Collections Framework enthalten
- Abhilfe: `Map<K, List<V>>`
- Beispiel
  - `PhoneBook`

Map II



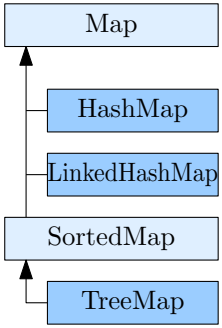
- Zugriff als `Collection`
  - `Set<K> keySet()`
  - `Collection<V> values()`
  - `Set<Map.Entry<K,V>> entrySet()`
- Beispiel
  - `MapDemo`
- MultiMaps nicht im Java Collections Framework enthalten
- Abhilfe: `Map<K, List<V>>`
- Beispiel
  - `PhoneBook`

Map II



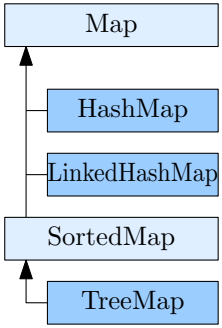
- Zugriff als `Collection`
  - `Set<K> keySet()`
  - `Collection<V> values()`
  - `Set<Map.Entry<K,V>> entrySet()`
- Beispiel
  - `MapDemo`
- MultiMaps nicht im Java Collections Framework enthalten
- Abhilfe: `Map<K, List<V>>`
- Beispiel
  - `PhoneBook`

Map II



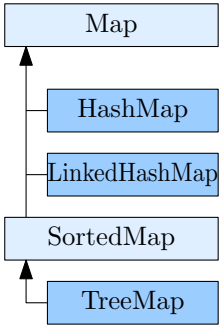
- Zugriff als `Collection`
  - `Set<K> keySet()`
  - `Collection<V> values()`
  - `Set<Map.Entry<K,V>> entrySet()`
- Beispiel
  - `MapDemo`
- MultiMaps nicht im Java Collections Framework enthalten
- Abhilfe: `Map<K, List<V>>`
- Beispiel
  - `PhoneBook`

Map II



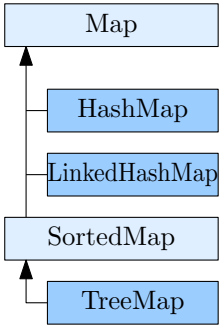
- Zugriff als `Collection`
  - `Set<K> keySet()`
  - `Collection<V> values()`
  - `Set<Map.Entry<K,V>> entrySet()`
- Beispiel
  - `MapDemo`
- MultiMaps nicht im Java Collections Framework enthalten
- Abhilfe: `Map<K, List<V>>`
- Beispiel
  - `PhoneBook`

Map II



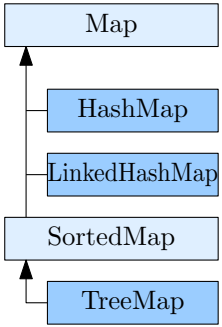
- Zugriff als `Collection`
  - `Set<K> keySet()`
  - `Collection<V> values()`
  - `Set<Map.Entry<K,V>> entrySet()`
- Beispiel
  - `MapDemo`
- MultiMaps nicht im Java Collections Framework enthalten
- Abhilfe: `Map<K, List<V>>`
- Beispiel
  - `PhoneBook`

Map II



- Zugriff als `Collection`
  - `Set<K> keySet()`
  - `Collection<V> values()`
  - `Set<Map.Entry<K,V>> entrySet()`
- Beispiel
  - `MapDemo`
- MultiMaps nicht im Java Collections Framework enthalten
- Abhilfe: `Map<K, List<V>>`
- Beispiel
  - PhoneBook

Map II



- Zugriff als `Collection`
  - `Set<K> keySet()`
  - `Collection<V> values()`
  - `Set<Map.Entry<K,V>> entrySet()`
- Beispiel
  - `MapDemo`
- MultiMaps nicht im Java Collections Framework enthalten
- Abhilfe: `Map<K, List<V>>`
- Beispiel
  - `PhoneBook`

Maps

Beispiel

MapDemo.java

```
import java.util.Map;
import java.util.HashMap;
import java.util.Scanner;

public class MapDemo {
  public static void main(String[] args) {
    Scanner s = new Scanner(System.in);

    Map<String, Integer> map = new HashMap<>();

    while (s.hasNext()) {
      String input = s.next();

      if (!map.containsKey(input))
        map.put(input, 0);

      map.put(input, map.get(input) + 1);
    }

    s.close();

    for (Map.Entry<String, Integer> entry : map.entrySet())
      System.out.println(entry.getKey() + ": " + entry.getValue());
  }
}
```

Telefonbuch

Beispiel

PhoneBook.java

```
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Scanner;
import java.util.Set;
import java.util.SortedMap;
import java.util.TreeMap;

public class PhoneBook {
  private static Scanner s;

  private SortedMap<String, List<String>> data = new TreeMap<>();

  public List<String> search() {
    while (s.hasNext()) {
      String line = s.nextLine();

      if (data.containsKey(line))
        return data.get(line);

      String end = line.substring(0, line.length() - 1)
          + (char) (line.charAt(line.length() - 1) + 1);

      Set<String> keys = data.subMap(line, end).keySet();
      for (String key : keys)
        System.out.println(" - " + key);

    }
    return new LinkedList<String>();
  }

  public void addNumber(String name, String number) {
    if (!data.containsKey(name))
      data.put(name, new LinkedList<String>());
    data.get(name).add(number);
  }

  public void delete(String name) {
```

Telefonbuch

Beispiel

PhoneBook.java

```
    data.remove(name);
  }

  public List<String> get(String name) {
    return data.get(name);
  }

  public void printList(List<String> list) {
    for (String entry : list)
      System.out.println(" # " + entry);
  }

  public void printList(String name) {
    if (data.containsKey(name))
      printList(data.get(name));
  }

  public void printAll() {
    for (Map.Entry<String, List<String>> entry : data.entrySet()) {
      System.out.println(entry.getKey());
      printList(entry.getValue());
    }

  }

  public static void main(String[] args) {
    PhoneBook contacts = new PhoneBook();
    help();
    System.out.print("> ");
    s = new Scanner(System.in);
    while (s.hasNext()) {
      String line = s.nextLine();
      cls(line);
      if (line.startsWith("add")) {
        String[] split = line.split(" ", 3);
        if (split.length == 3)
          contacts.addNumber(split[1], split[2]);
      } else if (line.startsWith("delete")) {
        String[] split = line.split(" ", 2);
```

Telefonbuch

Beispiel

PhoneBook.java

```
        if (split.length == 2)
          contacts.delete(split[1]);
      } else if (line.startsWith("get")) {
        String[] split = line.split(" ", 2);
        if (split.length == 2)
          contacts.printList(split[1]);
      } else if (line.startsWith("search")) {
        contacts.printList(contacts.search());
      } else if (line.startsWith("print")) {
        contacts.printAll();
      } else {
        System.out.println("unknown command");
      }
      System.out.print("> ");
    }
    s.close();
  }

  private static void help() {
    System.out.println("Commands:\n"
        + " - print \tprint the entire phone book\n"
        + " - add   \tadd a number\n" + " - delete\tdelete a contact\n"
        + " - get   \tlist contact's numbers\n"
        + " - search\tsearch for a contact with given suffix\n");
  }

  private static void cls(String command) {
    for (int i = 0; i < 100; i++)
      System.out.println();
    help();
    System.out.println("> " + command);
    System.out.println();
  }

}
```

Fragen?

Wiederholung
- Abstrakte Datentypen

Java Collections Framework
- Interfaces
- Listen
- Queue, Stack, Deque
- Sets
- Maps