Überblick

Mengen und Multimengen

- Eine *Menge* ist eine (ungeordnete) Zusammenfassung verschiedener Objekte. Insbesondere enthält eine Menge keine Duplikate.
- Eine *Multimenge* ist eine Menge, die Duplikate enthalten kann.
- Mengen und Multimengen sind wichtige Konzepte, u.a. um Assoziationen zwischen Objekten verschiedener Klassen umzusetzen.
- Im folgenden wiederholen wir grundlegende Funktionalitäten von Mengen. Eine Implementierung sollte entsprechende Methoden für diese Funktionalitäten anbieten. Diese gelten natürlich genauso für Multimengen.

Grundlegende Operationen auf Mengen

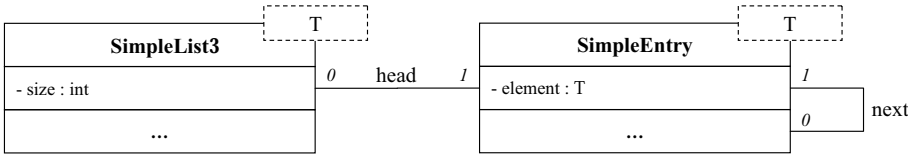
- Leere Menge:
  $\emptyset :\rightarrow Set(T)$
- Elementbeziehung:
  $\in: T \times Set(T) \rightarrow \mathbb{B}$
- Teilmengenbeziehung:
  $\subseteq: Set(T) \times Set(T) \rightarrow \mathbb{B}$
- Vereinigung:
  $\cup : Set(T) \times Set(T) \rightarrow Set(T)$
- Durchschnitt:
  $\cap : Set(T) \times Set(T) \rightarrow Set(T)$
- Differenz:
  $\setminus : Set(T) \times Set(T) \rightarrow Set(T)$
- Kardinalität:
  $|.| : Set(T) \rightarrow \mathbb{N}$

Realisierung von Mengen

- Zur effizienten Realisierung von Mengen spielen folgende Gedanken eine Rolle:
  - Die grundlegenden Operationen sollten effizient sein.
  - Das Einfügen eines neuen Elementes sollte effizient sein. Insbesondere ist darauf zu achten, dass keine Duplikate eingefügt werden.
  - Das Löschen eines Elementes sollte effizient sein.
  - Die Suche nach einem Element sollte effizient sein.
- Für die Realisierung von Multimengen spielen natürlich die selben Gedanken eine Rolle. Allerdings muss nun auf Duplikate keine Rücksicht genommen werden.

Realisierung von Mengen: Beispiel

- Als Beispiel schauen wir uns eine Implementierung einer Menge auf Basis einer typisierten (einfach) verketteten Liste an.
- Zur Erinnerung:



Realisierung von Mengen: Beispiel

```
public class Menge<T>
{
  private SimpleList3<T> elemente;

  public int cardinalitaet()
  {
    return this.elemente.length();
  }

  public T[] getElemente()
  {
    return this.elemente.asArray();
  }

  public void einfuegen(T element)
  {
    if(!this.enthaelt(element))
    {
      this.elemente.add(element);
    }
  }
  ...
}
```

Realisierung von Mengen: Beispiel ($\in$, $\cup$)

```
public boolean enthaelt(T t)
{
  return this.elemente.contains(t);
}


public void vereinigung(Menge<T> andereMenge)
{
  T[] neueElemente = andereMenge.getElemente();
  for(T t : neueElemente)
  {
    this.einfuegen(t);
  }
}
```


Realisierung von Mengen: Beispiel ($\subseteq$)

```
public boolean teilmengeVon(Menge<T> superMenge)
{
  boolean istTeilmenge = true;
  T[] elemente = this.getElemente();
  for(T t : elemente)
  {
    istTeilmenge = istTeilmenge && superMenge.enthaelt(t);
  }
  return istTeilmenge;
}
```

Zusammenfassung

Vergleich der durchschnittlichen Laufzeiten

	einfaches Array	einfache Liste	doppelt verankerte Liste	doppelt verkettete Liste	perfekte Hashverfahren (assoz. Array)	binärer Suchbaum
dynamisch	nein	ja	ja	ja	nein	ja
Einfügen (Schlüssel)	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$ bzw. $O(n)$	$O(\log n)$
Löschen (Schlüssel)	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$
Suchen (Schlüssel)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$

- Eine Standard-Implementierung einer Menge in Java ist z.B. die Klasse `java.util.HashSet`, die auf einem assoziativen Speicher beruht. Dabei wird intern die Methode `hashCode()` für die Objekte der Menge verwendet. Achtung: Die Laufzeit dieser Klasse degeneriert, falls die Hashfunktion (die auf der Methode `hashCode()` basiert), nicht optimal ist!

Überblick

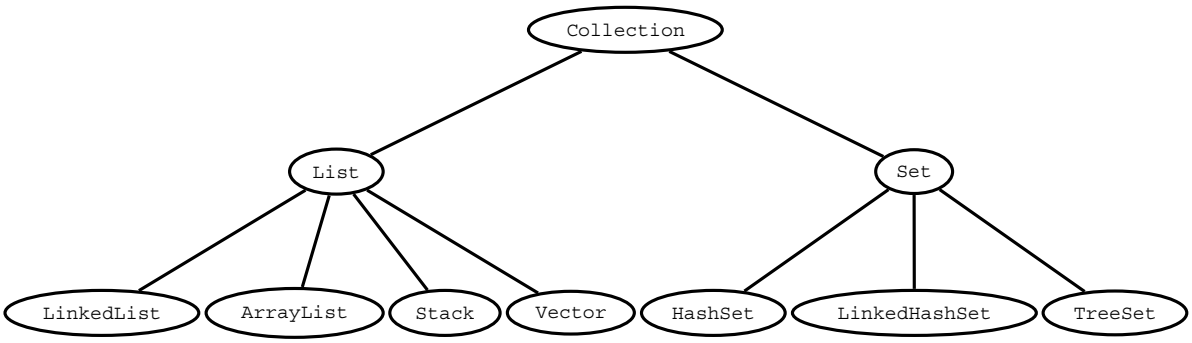
Collections in Java

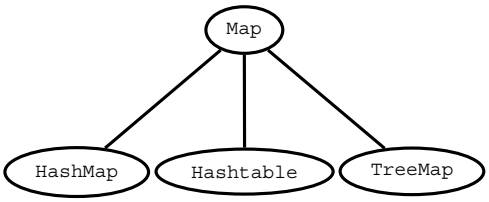
- In Java gibt es eine große Menge an vordefinierten Klassen für mengenartige Datenstrukturen.
- Im folgenden geben wir einen kurzen Überblick über das Collections-Framework von Java.
- Zur Vertiefung empfehlen wir das intensive Studium der Dokumentationen der entsprechenden Klassen.

Das Collections-Framework

- Alle Klassen mengenartiger, assoziativer Datenstrukturen implementieren das Interface `java.util.Map`. Das Interface stellt Basis-Funktionalitäten wie Einfügen (`put`), Löschen (`remove`) und verschiedene Formen der Suche (`get`, `containsKey`, `containsValue`) zur Verfügung.
- Alle Klassen mengenartiger, nicht-assoziativer Datenstrukturen implementieren das Interface `java.util.Collection`. Das Interface stellt Basis-Funktionalitäten wie Einfügen (`add`), Löschen (`remove`) und Suchen (`contains`) zur Verfügung.
- Vom Interface `java.util.Collection` abgeleitete Interfaces sind u.a.:
  - `java.util.Set`: Klassen, die Mengen ohne Duplikate modellieren, implementieren dieses Interface.
  - `java.util.List`: Klassen, die Multimengen modellieren, implementieren dieses Interface.

Hierarchie





Überblick

Zusammenfassung

Sie kennen jetzt

- verschiedene Datenstrukturen, die Mengen von Objekten modellieren:
  - “normale” Arrays,
  - Listen (Array-basierte, einfach verkettete, doppelt verankerte und doppelt verkettete),
  - assoziative Speicher (Hashverfahren),
  - binäre Bäume und Suchbäume

  und deren Vor- und Nachteile,
- das Java Collections-Framework mit den wichtigsten Interfaces, die mengenartige Datenstrukturen implementieren sollten, sowie einige konkrete, vordefinierte Java-Klassen für assoziative und nicht assoziative Mengen und Multimengen.