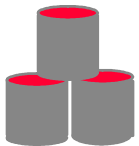
Grundlagen der Informatik
Generische Klassen

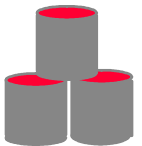
Generische Klassen, das Java-Collection-Framework und mehr

Prof. Dr. Bernhard Schiefer



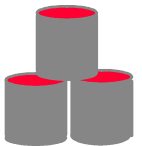
Generische Programmierung

- Beobachtung: In vielen Problemstellungen hängt der grundsätzliche Aufbau der Lösung nicht vom jeweiligen Datentyp ab
- Beispiele:
  - ⇨ Algorithmen: wie z.B. Sortieralgorithmus zum Sortieren von Elementen
    - Das Sortieren von Zahlen, Buchstaben oder Texten funktioniert im Grunde gleich
  - ⇨ Datenstrukturen: Container (z. B. Liste) zur Aufbewahrung von Elementen
    - Die Verwaltung einer Liste von Texten erfolgt im Grunde so wie bei Zahlen
- Gesucht:
  Programmiertechnik, bei der die Algorithmen und Datenstrukturen nicht für spezielle Datentypen geschrieben werden müssen



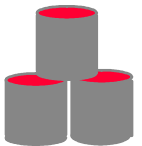
Grundlagen generischer Programmierung

- Basis:
  Die zu verwendeten Datentypen werden als Parameter angegeben
  - ⇨ Übliche Bezeichnung: **Typvariablen, Typparameter**
- Manchmal ist es notwendig/sinnvoll bestimmte Anforderungen an die Parameter zu stellen.
  - ⇨ Z.B. muss mindestens vom Typ X sein
- Wird heute von vielen modernen Programmiersprachen wie Java, C++, C#, VisualBasic und .NET unterstützt



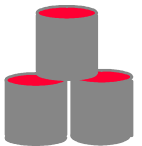
Generische Programmierung: Vorteile

- generische Wiederverwendung von Software-Komponenten
- Algorithmen und Datenstrukturen werden soweit wie möglich von den Datentypen, mit denen sie arbeiten, getrennt
- strenge Typsicherstellung während der Übersetzungszeit
  - ⇨ Container: nur gleichartige Objekte können eingefügt werden
  - ⇨ Algorithmen: Typen von Parametern und Rückgabewerten können garantiert werden
- Quellcode kann lesbarer und Programme sicherer gemacht werden
- abhängig von der Umsetzung: Geschwindigkeitssvorteile



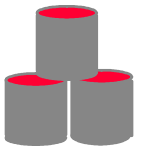
Generische Programmierung: Umsetzung

- Code Duplizierung:
  - ⇨ Der Typparameter wird zur Compilierzeit direkt durch den konkreten Typ ersetzt. Jede Instanziierung ist eine speziell angepasste Kopie des Algorithmus oder der Datenstruktur.
    - z. B. C++-Templates, -Schablonen
- Code Umschreibung:
  - ⇨ Für alle Instanziierungen wird der gleiche Code des Algorithmus oder der Datenstruktur benutzt.
  - ⇨ Die Aufrufschnittstellen zur Benutzung werden durch implizites Umschreiben entsprechend angepasst.
    - z. B. Java
- Mischformen:
  - ⇨ z. B. .NET: Referenztypen unterliegen Umschreibung, elementare Typen unterliegen Duplizierung



Generische Klassen in Java

- Generische Klassen verfügen über Typparameter
  - ⇨ Konventionsgemäß großgeschriebene kurze Bezeichner
  - ⇨ Typparameter sind nur für nicht-elementaren Datentypen vorgesehen, die elementaren Datentypen müssen eingepackt werden.
- Beispiel Definition:
  - ⇨ public class Vector<T> extends AbstractList<T> {
    add (T elem);
    …
    }
- Beispiel Nutzung:
  - ⇨ Vector<Girokonto> konten = **new** Vector<Girokonto>();
    konten.add(**new** Girokonto(1000, 0));



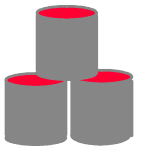
Generische Klassen in Java - Parametereinschränkung

- Verschiedene Einschränkungen zulässiger Parameter für die Ersetzung von Typparametern können angegeben werden:
  - ⇨ Beispiele: *T* extends *Class* **oder** *T* super *Class*
  - ⇨ Diese Angabe erlaubt auch Klassen, die ein Interface implementieren
    - Es wird also nicht zwischen extends und implements unterschieden
- Beispiel Definition:
  - ⇨
```
public class Kontenliste<T extends Konto>  {
   add (T elem);
   …
}
```
- Bei Variablen und Argumenten kann auch das Wildcard Argument ? genutzt werden. Es lässt beliebige Typen zu. Beispiel:
  - ⇨
```
Kontenliste<?>  kl;
```

Das Java Collection Framework

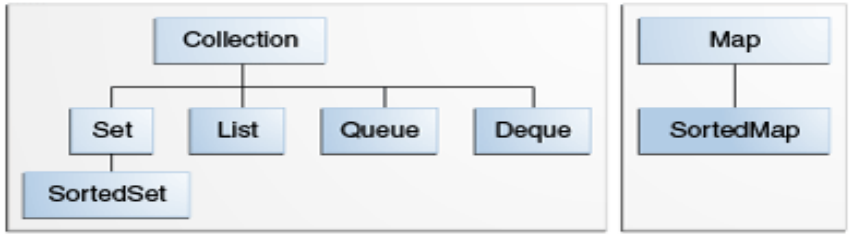
- Java bringt zahleiche nützliche Standardklassen im Paket *java.util* mit
- Dazu gehören generische Klassen im Paket *java.util* , die das Interface *Collection* implementieren.
  - ⇨ Alle Klassen, die Collection implementieren verfügen u.a. über einen Iterator und können mit der for-Schleife einfach durchlaufen werden.
  - ⇨ Klassen: Vector, LinkedList, TreeSet, …
- Beispiel:
  - ⇨ Vector<Konto> konten = …

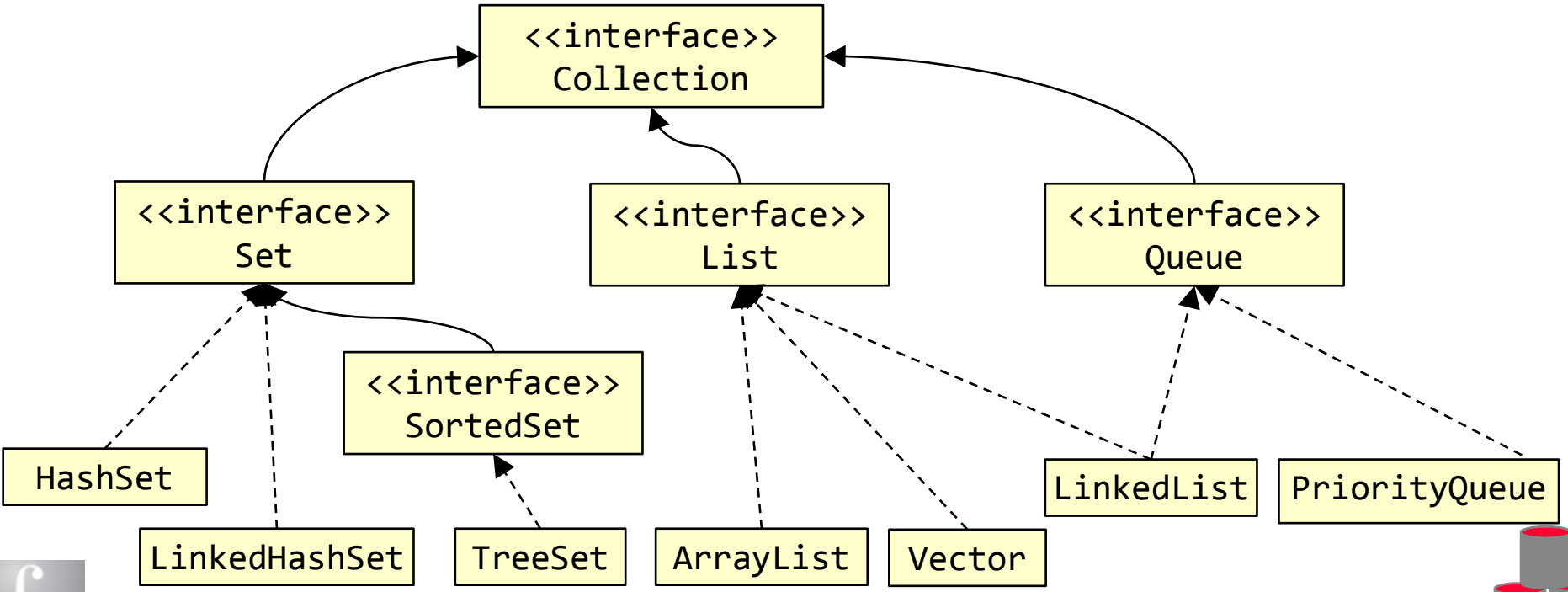
```
for (Konto k : konten) {
  k. auszahlen(5.0);
}
```



Hierarchie der Interfaces im Collection Framework

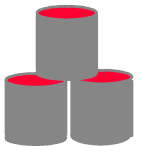
- Aus http://docs.oracle.com/javase/tutorial/collections :





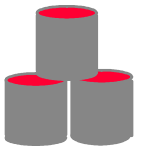
Collections<E> allgemein

- Wesentliche Eigenschaften: Elemente hinzufügbar und iterierbar.
- Wichtige Operationen: (weitere siehe online Javadoc)
  - ⇨ add(E e)
  - ⇨ addAll(Collection<? extends E> c)
  - ⇨ clear()
  - ⇨ contains(Object o)
  - ⇨ remove(Object o) (Achtung bei Duplikaten: Entfernt nur ein Element!)
  - ⇨ size()
  - ⇨ isEmpty()
- Durch die Implementierung des Iterable-Interfaces verwendbar in for-Schleifen:
  - ⇨ Iterator<E> iterator()



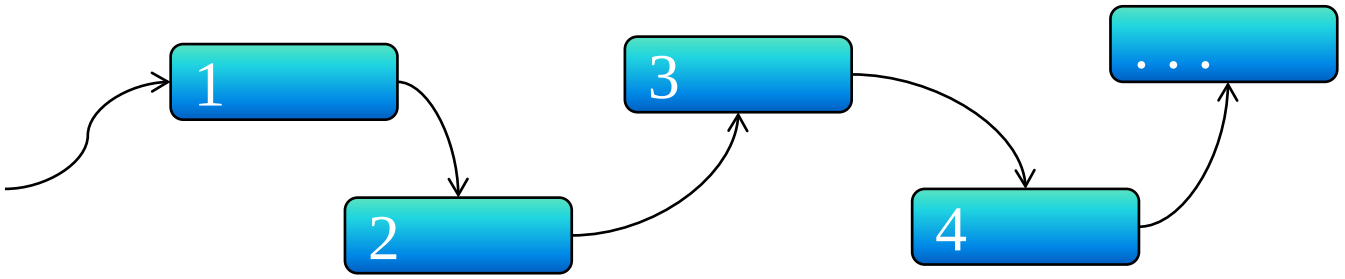
List<E> – Listen

- Wichtige Operationen:
  - ⇨ Alles was von Collection (bzw. Iterable) kommt
  - ⇨ void add(int index, E element)
  - ⇨ boolean addAll(int index, Collection<? extends E> c)
  - ⇨ E get(int index)
  - ⇨ int indexOf(Object o)
  - ⇨ E remove(int index)
- Implementierungen
  - ⇨ ArrayList<E>, Vector<E>
    - wenn bevorzugt über Index zugegriffen wird
    - Hinzufügen nur am Ende
  - ⇨ LinkedList<E>
    - wenn beliebig hinzugefügt, gelöscht und sequentiell durchlaufen wird



Implementierungsalternativen

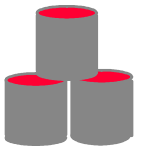
- Variante: LinkedList



- Variante: ArrayList, Vector

Set<E> – Mengen

- Wichtige Operationen:
  - ⇨ Alles was von Collection (bzw. Iterable) kommt
  - ⇨ Keine weiteren Operationen
  - ⇨ Besonderheit: Es können keine Duplikate eingefügt werden
- Implementierungen:
  - ⇨ HashSet<E>
  - ⇨ TreeSet<E>
  - ⇨ EnumSet<E>
  - ⇨ …

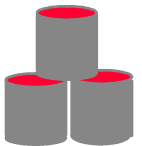


Das Interface Map<K,V>

- Wesentliche Eigenschaften: Zuordnung von Schlüsseln (K) zu Werten (V)
- Wichtige Operationen:
  - ⇨ put(K key, V value)
  - ⇨ V remove(K key)
  - ⇨ V get(K key)
- Implementierungen:
  - ⇨ HashMap, Hashtable, LinkedHashMap, …
- Beispiel:
  - ⇨ Map<Integer, Kunde> telefonbuch= new HashMap<Integer, Kunde>();

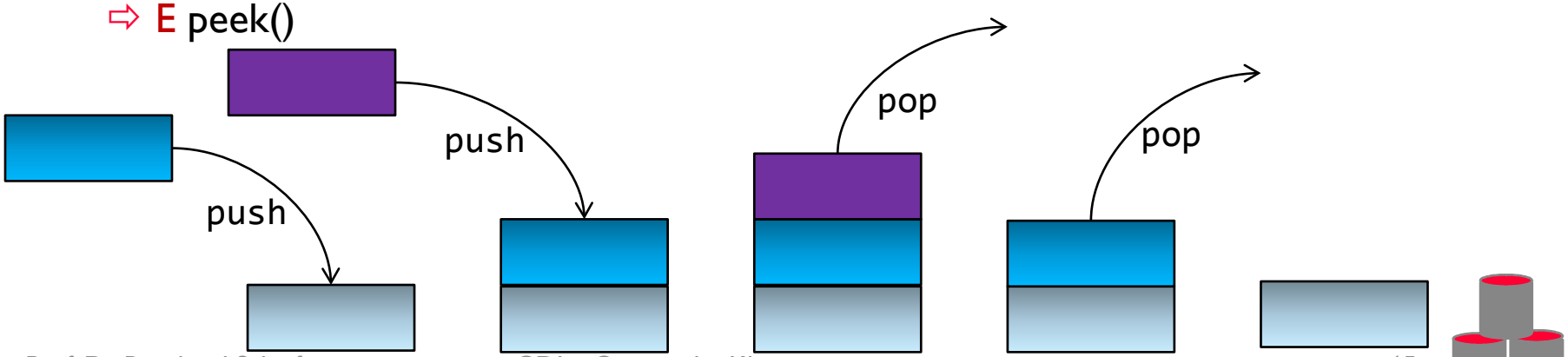
```
telefonbuch.put (4711, m1);   // speichern
```

  - ⇨ Kunde m = telefonbuch.get(4711); // wiederfinden



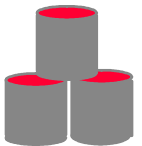
Die Klasse Stack<E>

- Ein Stack (Stapel) ist dann nützlich, wenn eine Struktur benötigt wird, bei der die Elemente in der umgekehrten Reihenfolge, in der sie gespeichert wurden, ausgelesen werden sollen
  - ⇨ Prinzip: Last-In-First-Out (LIFO)
  - ⇨ Beispiele?
- Wichtige Operationen:
  - ⇨ E push(E item)
  - ⇨ E pop()
  - ⇨ E peek()



Das Interface Queue<E>

- Eine Queue sollte immer dann genutzt werden, wenn das zuerst eingefügte Objekt auch als erstes wieder entfernt werden soll
  - ⇨ Prinzip: First-In-First-Out (FIFO)
- Typische Anwendungen:
  - ⇨ Warteschlangen, etc.
- Wichtige Operationen:
  - ⇨ boolean add(E e)
  - ⇨ E remove()
  - ⇨ E peek()



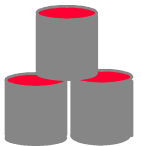
Wandel zwischen Collections und Arrays

- Hilfsklassen stehen für die Konvertierung zur Verfügung
  - ⇨ Arrays.asList ( )
  - ⇨ Collections. toArray( )

- Beispiele:
  - ⇨ String[] werte = {"Berta", "Clara", "Anton" };
    List<String> sl = Arrays.asList(werte);
  - ⇨ String[] werte = sl.toArray(new String[sl.size()]);

Sortieren

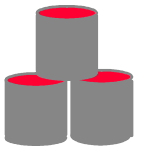
- Häufige Anforderung aus der Praxis:
  - ⇨ Geordnete Ausgabe von Werten (z.B. nach Name sortiert)
  - ⇨ Durchsuchen einer Liste nach verschiedenen Kriterien
  - ⇨ Finden des größtes (kleinstes) Elementes
- Lösung aus dem Collection-Framework:
  - ⇨ void Collections.sort(List<T> list)
  - ⇨ T Collections.max(Collection<T> coll)
  - ⇨ T Collections.min(Collection<T> coll)
- Beispiel:
  - ⇨ String[] werte = { "Berta", "Clara", "Anton" };
    List<String> sl = Arrays.asList(werte);
  - ⇨ Collections.sort(sl);



Problem: Sortierung gemäß Codierung

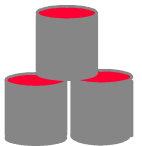
- Die Sortierung von Strings basiert in Java auf der Reihenfolge der Unicode-Werte – das liefert nicht immer das erwartete Ergebnis
  - ⇨ Warum?

- Nebenbei: Ähnliche Probleme auch bei
  - ⇨ Sortierung in anderen ASCII-basierten Programmiersprachen
  - ⇨ sortierter Ausgabe aus Datenbanken, etc.



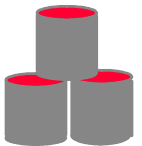
Exkurs: Sortieren deutscher Texte (1)

- Sortieren mit Sonderzeichen (Umlaute, Accents, …) ist immer problematisch.
  - ⇨ aber auch die gewünschte Behandlung von Worten mit gemischter Groß-/Kleinschreibung ist zu bedenken
- Java: Standardsortierung pro Sprache/Locale:
  - ⇨ java.text.Collator deutsch = Collator.getInstance(Locale.GERMAN);
  - ⇨ Collections.sort(sl, deutsch); // Standardsortierung
- Aber: Selbst innerhalb der deutschen Sprache gibt es unterschiedliche Regeln:
  - ⇨ DIN 5007 Variante 1 (z.B. in Lexika, Duden, …)
    - *ä* und *a* sind gleich, *ö* und *o* sind gleich, *ü* und *u* sind gleich
  - ⇨ DIN 5007 Variante 2 (für Namenslisten, Telefonbüchern, …)
    - *ä* und *ae* sind gleich, *ö* und *oe* sind gleich, *ü* und *ue* sind gleich



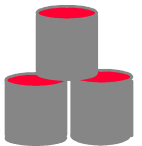
Exkurs: Sortieren deutscher Texte (2)

- Weitere Abweichungen in anderen deutschsprachigen Ländern
  - ⇨ Z.B. Österreich: *ä* folgt auf *a* (kommt also nach *az*), etc.
- Unerwartete Sortierungen auch in anderen europ. Sprachen
  - ⇨ z.B. Schwedisch: å folgt auf z, ä folgt auf å, ö folgt auf ä, ü und y sind gleich
- Java-Lösung
  - ⇨ Klasse: java.text.RuleBasedCollator (extends Collator)
  - ⇨ Diese Klasse erlaubt die Angabe beliebiger Reihenfolgeregeln
  - ⇨ Details siehe JavaDoc der Klasse



Vergleichsoperator

- Sortierung von Instanzen einer Klasse erfordert eine Ordnungsrelation
- Eine natürliche Ordnungsrelation wird in Java definiert durch die Methode *compareTo* (z.B. für String)
- Ergebnis von a.compareTo (b) liefert:
  - ⇨ < 0 falls a < b
  - ⇨ = 0 falls a == b
  - ⇨ > 0 falls a > b
- In der Praxis werden auch Ordnungen zwischen komplexeren Objekten benötigt.
  - ⇨ Lösung?

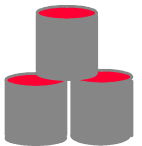


Das Interface Comparable<T>

- Nicht für alle Objekte ist eine Ordnungsrelation sinnvoll/möglich.
  - ⇨ Die Klasse Objekt enthält die Methode nicht
  - ⇨ Nicht jedes Objekt verfügt über *compareTo*
- Realisierung über ein generisches Interface:
  - ⇨ java.lang.Comparable<T>
- Das Interface enthält nur eine Methode
  - ⇨ int compareTo(T x)
- Beispiel: Sortierung nach Kontonummer

```
public class Konto implements Comparable<Konto> {

  @Override
  public int compareTo (Konto k) {
    return getKontonr() - k.getKontonr();
  }
}
```

Was tun bei unterschiedlichen Kriterien

- Häufig werden unterschiedliche Sortierungen für Listen benötigt
  - ⇨ Z.B. nach Name, oder Nummer, oder Anzahl Elemente, oder …

- Lösung: Es existiert eine zweite Variante von Collections.sort
  - ⇨ sort(List<T> list, Comparator<? super T> c)

Das Interface Comparator<T>

- Es existiert eine zweite Variante von Collections.sort:
  - ⇨ sort(List<T> list, Comparator<? super T> c)
- Methoden
  - ⇨ int compare(T o1, T o2)
  - ⇨ boolean equals(Object obj)
- Damit lassen sich beliebig viele Vergleichsmethoden für eine Klasse implementieren
- Beispiel: Sortierung nach Kontostand

```
public class SaldoComparator implements Comparator<Konto> {

  @Override
  public int compare(Konto k1, Konto k2) {
    return k1.getSaldo() – k2.getSaldo();
  }
}
```