Collections

Motivation

- Arrays sind eine effiziente, aber nicht besonders flexible Datenstruktur

$\Rightarrow$ Nachträgliche Array-Vergrößerungen nicht vorgesehen

$\Rightarrow$ Viel mehr Flexibilität bieten die Collections-Klassen der JavaSE

Ziel

- Kennenlernen und Anwenden der Klassen `ArrayList`, `HashSet` und `HashMap` basierend auf den Schnittstellen `Collection`, `List`, `Set` und `Map`
- Stream-Schnittstelle zur Verarbeitung von Collection-Elements durch Lambda-Ausdrücke

Beispiel <List>

Variante 1

```
List<Point> polygon = new ArrayList<>();
polygon.add(new Point(1, 1));
polygon.add(new Point(1, 2));
polygon.add(new Point(2, 2));
polygon.add(new Point(2, 1));
polygon.forEach(p -> System.out.format("(%d , %d)\n", p.x, p.y));
```

Variante 2

```
var polygon = List.of(new Point(1, 1), new Point(1, 2), new Point(2, 2), new Point(2, 1));
polygon.forEach(p -> System.out.format("(%d , %d)\n", p.x, p.y));
```

Unterschied zwischen Variante 1 und Variante 2?

Beispiel <Set>

6 aus 49

```
var rnd = new Random();
var lotto = new HashSet<>();
do {
  lotto.add(1 + rnd.nextInt(49));
} while (lotto.size() < 6);
System.out.println(Arrays.toString(lotto.toArray()));
```

Beschreiben Sie den Code! Fehlt etwas?

Beispiel <Map>

```
Map<String, String> dict = new HashMap<>();
dict.put("one", "eins");
dict.put("two", "zwei");
dict.put("three", "drei");
dict.put("four", "vier");
System.out.println(dict.get("three"));
```

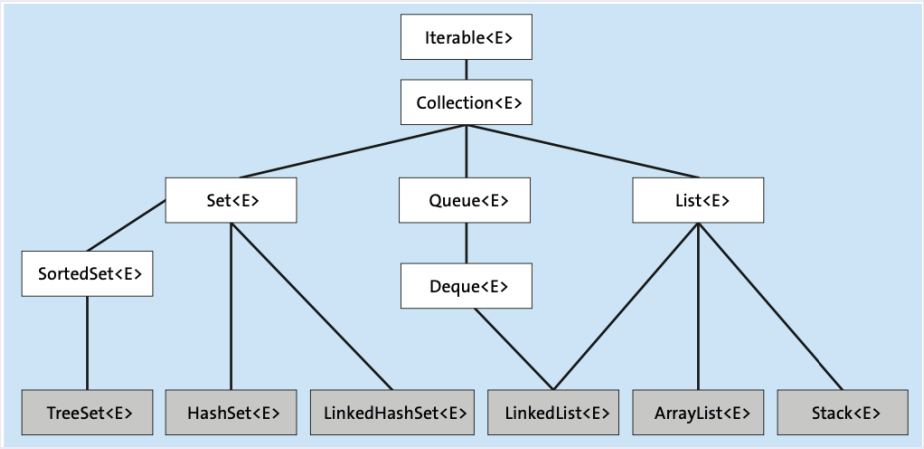
`Map.of` akzeptiert maximal 10 Key-Value-Paare

```
var dict = Map.of("one", "eins", "two", "zwei", "three", "drei", "four", "vier");
```

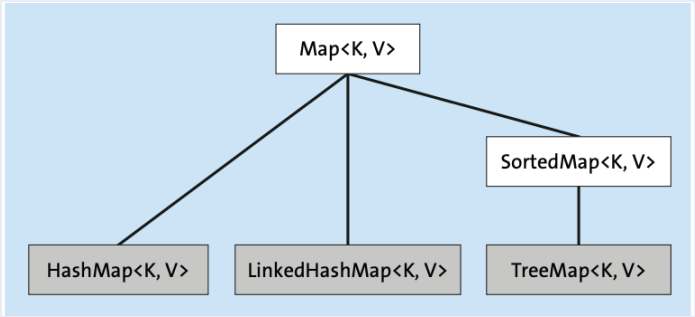
beliebig viele Key-Value-Paare

```
Map<String, String> dict2 = Map.ofEntries(entry("one", "eins"), entry("two", "zwei"),
    entry("three", "drei"), entry("four", "vier"));
```

Übersicht Collection



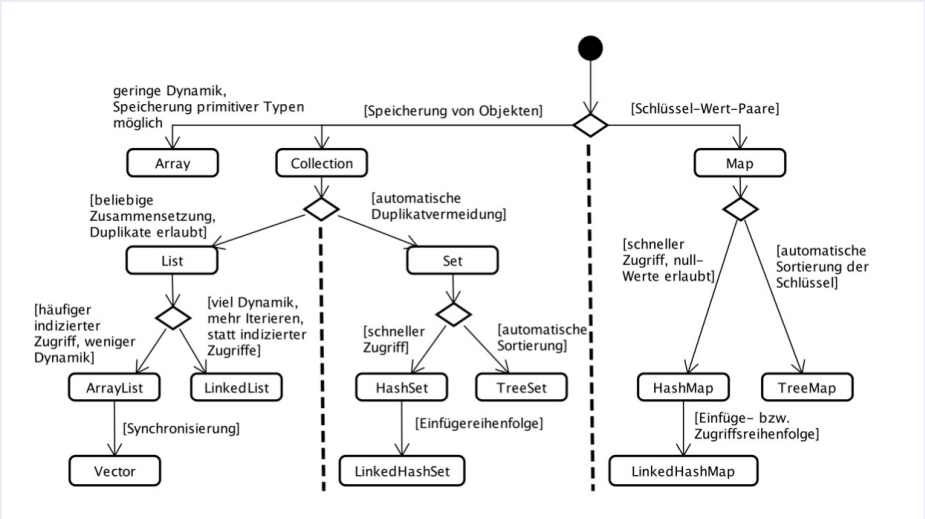
Übersicht Map



Übersicht: Java Collections Framework

- Collection: Oberbegriff für Aggregatdatentypen (Containerdatentypen), in denen andere Elemente enthalten sind
- Operationen: Hinzufügen, Entfernen, Suchen, Durchlaufen
- Hauptinterfaces (in java.util):
  - `Collection` Grundfunktionalität für alle Datentypen außer für Abbildungen. Keine konkrete Implementierung
  - `Set` Aggregation ohne Wiederholung, Reihenfolge unerheblich. Zwei Spezialisierungen: SortedSet und NavigableSet
  - `Queue` Warteschlange, FIFO. Spezialisierung: Deque (an beiden Enden anfügen und entfernen)
  - `List` Aggregation mit Wiederholung und fester Reihenfolge
  - `Map` endliche Abbildung. Spezialisierung: `SortedMap` und `NavigableMap`

Entscheidungshilfe



Regeln, Tipps und Tricks

- Collections können Instanzen beliebiger Klassen aufnehmen, ungeeignet für elementare Datentypen
- Collections bieten mehr Flexibilität als Arrays, sind je nach Einsatzzweck aber langsamer
- for-each-Schleifen eignen sich wunderbar zur Verarbeitung der Elemente einer Collection ()
- Löschen nicht direkt möglich $\Rightarrow$ (Einsatz Iterable-Schnittstelle))
- Die meisten Collections-Klassen sind nicht synchronisiert (wichtig: kommt noch im Thread Kapitel)

Die „Iterable“-Schnittstelle

Die Schnittstelle `Iterable<T>` definiert die Methode `iterator`. Sie gibt ein `Iterator<T>`-Objekt zurück, das die for-each-Schleife auswertet

```
public interface Iterable<T> {
  Iterator<T> interator();
  default void forEach(Consumer<? super T> action){...}
  default Spliterator<T> spliterator() {...}
}
```

Entstehung ...

Vor Java 1.5

```
for (Iterator<Integer> it = s.iterator(); it.hasNext(); ){
  System.out.println(it.next());
}
```

Seit Java 1.5: Jede Klasse, welche die `Iterable<T>`-Schnittstelle implementiert, kann unkompliziert in einer for-each-Schleife durchlaufen werden

```
var s = Set.of(1, 5, 7);
for (Integer i: s)
  System.out.println(i);
```

Ein Zähler mit Iterable

```
public class CounterExample {
  public static void main(String[] arg) {
    int total = 0;
    for (int i : new Counter(3)) {
      total += i;
      System.out.println(total);
    }
    assert total == 10;
  }
}
class Counter implements Iterable<Integer> {
  private int count;
  public Counter(int count) {
    this.count = count;
  }
  public Iterator<Integer> iterator() {
    return new CounterIterator(this.count);
  }
}
```

```
class CounterIterator implements Iterator<Integer> {
  private int count;
  private int i;
  CounterIterator(int count) {
    this.count = count;
    this.i = 0;
  }
  public boolean hasNext() {
    return this.i <= this.count;
  }
  public Integer next() {
    this.i++;
    return this.i;
  }
  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

Das Interface Collection

Die Schnittstelle `Collection<E>` erweitert die gerade beschriebene Schnittstelle `Iterable` um Methoden zur Bearbeitung von Mengen

```
public interface Collection<E> {
  public boolean add (E o) // fuegt ein Element hinzu
  public boolean addAll (Collection<? extends E> c); // fuegt mehrere Elemente auf einmal
      hinzu
  public boolean remove (Object o); // entfernt ein Element
  public void clear();
  public boolean removeAll(Collection<?> c); // entfernt eine ganze Gruppe von Elementen
  public boolean retainAll(Collection<?> c); //  verbleiben nur die Elemente in der
      Collection, die mit Elementen des Parameters c uebereinstimmen
  public boolean contains(Object o);
  public boolean containsAll(Collection<?> c); // ueberprueft, ob alle angegebenen
      Elemente in der Collection enthalten sind
  public boolean isEmpty();
  public int size(); // liefert Anzahl der Elemente zurueck
  public Iterator<E> iterator();
  public Object[] toArray();
  public <T>T[] toArray (T[] a);
}
```

Achtung: a.xxxAll(b) verändert die Menge an

empfohlene Vorgehensweise

```
var a = Set.of(1, 5, 7); // unveraenderlich
var b = Set.of(5, 9, 1); // unveraenderlich
var c = new HashSet<>(a);
c.retainAll(b);
System.out.println(Arrays.toString(c.toArray()));
// Ausgabe: [1, 5]
```

Methode `removeIf`

empfohlene Vorgehensweise

```
var s = new HashSet<Integer>();
for (int I=1; i<10; i++)
  s.add(i)
s.removeIf(i -> ( i % 2 ) !=0 );
s.forEach(i -> System.out.println(i));
```

„Set“-Schnittstelle

- Schnittstelle `Set` ist exakt genauso wie die von `Collection` definiert
- Unterschied: Jede Implementierung von `Set` muss sicherstellen, dass es keine Doppelgänger gibt

$\Rightarrow$ Wenn einem `Set` mit `add` ein Objekt hinzugefügt wird, dass bereits enthalten ist, wird es nicht neuerlich hinzugefügt (`add` liefert dann false)

HashSet

- `HashSet`-Klasse ist die gängigste und für die meisten Anwendungsfälle auch effizienteste Implementierung eines Sets
- `HashSet`-Klasse bietet keine Garantien, dass die Reihenfolge erhalten bleibt, in der die Objekte eingefügt werden.

HashSet-Konstruktor

```
Set<Integer> a = new HashSet<>();

Set<Point> b = new HashSet<>(1000); // hilft Java bei der effizienten Verwaltung der
    Speicherstrukturen

var c =s new HashSet<>(b); // Beim Erzeugen des neuen HashSets werden alle in der
    urspruenglichen Collection enthaltenen Doppelgaenger eliminiert
```

Wie sieht es hier aus?

```
class Rectangle {
  public int w, h;
  public Rectangle(int w, int h) {
    this.w = w;
    this.h = h;
  }
}
public class TestRectangle {
  public static void main(String[] args) {
    var set = new HashSet<Rectangle>();
    set.add(new Rectangle(2, 1));
    set.add(new Rectangle(2, 2));
    set.add(new Rectangle(2, 1));
    for (Rectangle r : set)
    System.out.println(r.w + " " + r.h);
  }
}
```

Wie sieht es hier aus?

Zur Erkennung von Doppelgängern werden die Methoden `hashCode` und `equals` verwendet.

```
@Override
public int hashCode() {
  return Objects.hash(h, w);
}
```

```
@Override
public boolean equals(Object obj) {
  if (this == obj)
  return true;
  if (obj == null)
  return false;
  if (getClass() != obj.getClass())
  return false;
  Rectangle other = (Rectangle) obj;
  return h == other.h && w == other.w;
}
```

Wie sieht es hier aus?

Nachträgliche Änderung eliminiert keine „Doppelgänger“

```
public class TestRectangle {
  public static void main(String[] args) {
    var set = new HashSet<Rectangle>();
    Rectangle r1 = new Rectangle(1, 2);
    Rectangle r2 = new Rectangle(3, 4);
    set.add(r1);
    set.add(r2);
    r2.w = 1;
    r2.h = 2;
    for (Rectangle r : set)
      System.out.println(r.w + " " + r.h);
  }
}
```

LinkedHashSet

- Beim `LinkedHashSet` bleibt die Reihenfolge der Elemente beim Einfügen erhalten
- Das wiederholte Einfügen eines bereits vorhandenen Elements ändert dessen Position in der Menge nicht
- von jedem Element verweisen Links zum vorigen und zum nächsten Element

TreeSet

- Beim `TreeSet` ist die Reihenfolge der Elemente durch deren Sortierordnung vorgegeben und wird durch eine Baumstruktur verwaltet
- Die Klasse der zu speichernden Objekte implementiert entweder `Comparable` oder ein `Comparator` muss erstellt werden
- `TreeSet` schlechtere Performance als `LinkedHashSet`
- `TreeSet` hat automatische Sortierung

Was ist die Ausgabe?

```
// Vergleich von Koordinatenpunkten je nach Abstand zu (0,0)
Comparator<Point> comp = (p1, p2) -> {
  double l1 = Math.sqrt(p1.x * p1.x + p1.y * p1.y);
  double l2 = Math.sqrt(p2.x * p2.x + p2.y * p2.y);
  if (l1 < l2)
    return -1;
  else if (l1 > l2)
    return 1;
  else
    return 0;
};
var set = new TreeSet<Point>(comp);
set.add(new Point(2, 1));
set.add(new Point(2, 2));
set.add(new Point(1, 2));
set.add(new Point(1, 1));
for (Point p : set)
  System.out.println(p.x + " " + p.y);
```

TreeSet hat einiges zusätzliche Methoden

- `first` und `last` liefern das erste und das letzte Element des Sets
- `descendingIterator` für Ausgabe in umgekehrter Reihenfolge

```
for (Iterator<Point> pit = set.descendingIterator(); pit.hasNext();) {
  Point p = pit.next();
  System.out.println(p.x + " " + p.y);
}
```

„List“-Schnittstelle

- `List`-Schnittstelle basiert wie `Set` auf der `Collection`-Schnittstelle
- Unterschiede zwischen `List` und `Set` bestehen darin, dass in Listen die Reihenfolge der Elemente immer erhalten bleibt (unabhängig von der Implementierung) und dass Doppelgänger erlaubt sind

Methoden der List-Schnittstelle

```
public interface List<E> extends Collection<E> {
  // Daten einfuegen und entfernen
  boolean add (E element);
  void add (int index, E element);
  boolean addAll(int index, Collection<? extends E> c);
  E set(int index, E element);

  // Daten lesen
  E get(int index);
  int indexOf(Object o);
  int lastIndexOf(Object o);
  List<E> subList(int from; int to);
  E remove(int index);
  ...
}
```

... im Detail

- `lst.add(obj)` fügt ein neues Element am Ende der Liste hinzu
- `lst.add(n, obj)` verschiebt das Element, das sich bisher an der Position n befindet, sowie alle weiteren Elemente um eine Position nach rechts
- Gezielt Elemente mit `get(n)` lesen
- `ListIterator` kann zur manuellen Bildung von Schleifen verwendet werden. Im Unterschied zum gewöhnlichen Iterator der `Collection`- Schnittstelle kann der `ListIterator` mit der Methode `previous` auch auf das vorige Element zugreifen
- ...

Methoden der List-Schnittstelle

```
var lst = new ArrayList<String>();
lst.add("Listen");
lst.add("mit");
lst.add("Lambda-Ausdruecken");
lst.add("veraendern");
lst.replaceAll(s-> s.toLowerCase());
lst.forEach(s -> System.out.println(s));
```

ArrayList

- `ArrayList`-Klasse ist die populärste und für viele Anwendungen effizienteste List-Implementierung.
- Flexibel in der Größe, aber dem Konstruktor kann auch die Anzahl der zu speichernden Elemente übergeben werden

```
var lst = new ArrayList<String>();
lst.add("Das");
lst.add("ist");
lst.add("ein");
lst.add("Satz.");
lst.add(3, "langer");
for (String s : lst) // Ausgabe: Das ist ein langer Satz.
  System.out.print(s + " ");
for (int i = lst.size() - 1; i >= 0; i--) // Ausgabe: Satz. langer ein ist Das
  System.out.print(lst.get(i) + " ");
```

LinkedList

- `LinkedList`-Klasse ist eine Alternative zur `ArrayList`-Klasse
- Links auf Vorgänger bzw. Nachfolger
- `LinkedList` implementiert `Queue/Deque`

$\Rightarrow$ Nutzung der Klasse als FIFO- und LIFO-Speicher

Problem: Die Methode <T>T[] toArray (T[] a)

Verwendung

- Kopieren in Array von Supertyp von E (geht immer)
- Kopieren in Array von Subtyp von E (Spezialisieren der Collection)

```
Collection<String> cs = ...;
String[] sa = cs.toArray (new String[0]);
```

Die Methode <T>T[] toArray (T[] a)

Spezialisierung

```
List<Object> lo = new Arrays.asList("zero", "one");
String[] sa = lo.toArray (new String[0]);
```

Fehlschlag

```
List<Object> lo = new Arrays.asList("zero", "one", 42);
String[] sa = lo.toArray (new String[0]); //Laufzeitfehler
```

Sinnvolle T-Typen

- Subtypen von E
- Sypertypen von E

$\Rightarrow$ Das lässt sich in Java nur eingeschränkt ausdrücken (Stichwort: Wildcards)

„Probleme“ bei Collections

Ein Designfehler in Java

- In Java gilt: Falls A Subklasse von B, dann auch A[] Subtyp von B[]
- Diese Regel heißt „kovariantes Subtyping von Arrays“

$\Rightarrow$ Sie ist nicht korrekt und erzwingt spezielle Tests zur Laufzeit!

Kompiliert das Programm? Was passiert?

```
public class Test2 {
  public static void main(String[] args) {
    new ArrayCheck().test();
  }
}
class ArrayCheck {
  static class B {
  }
  static class A extends B {
    void m() {
    }
  }
  void test() {
    A[] a = new A[1];
    a[0] = new A();
    a[0].m();
    muddle(a);
    a[0].m();
  }
  void muddle(B[] b) {
    b[0] = new B();
  }
}
```

Bad Smell: Vergleich von Arrays

```
public static void main(final Steing[] args)
{
  final String[][] array1 = {{"0.0", "0.1"}, {"1.0", "1.1"}};
  final String[][] array2 = {{"0.0", "0.1"}, {"1.0", "1.1"}};

  final boolean arrayEquals = Arrays.equals(array1, array2);
  final boolean deepEquals = Arrays.deepEquals(array1, array2);

  System.out.println("equals = " + arrayEquals); // false
  System.out.println("equals = " + deepEquals); // true
}
```

Bad Smell: Rückgabe von Arrays und die Gefahr von Inkonsistenzen

```
public class BadSmell2 {
  private static final String[] CAPITAL_CITIES = new String[] { "Berlin", "London", "Paris
      ", "Wien" };
  public static final String[] getCities() {
    return CAPITAL_CITIES;
  }
  public static final Iterator<String> getCityIterator() {
    return Arrays.asList(CAPITAL_CITIES).iterator();
  }
  public static void main(final String[] args) {
    System.out.println("CITIES " + Arrays.toString(CAPITAL_CITIES));
    final String[] cities = getCities();
    cities[1] = "London has changed!";
    System.out.println("CITIES " + Arrays.toString(cities));
    System.out.println("CITIES " + Arrays.toString(CAPITAL_CITIES));
  }
}
```

Map-Schnittstelle

- Maps dienen zur Speicherung von Schlüssel-Wert-Paaren
- `Map` ist nicht von `Collection` abgeleitet
- Wichtigsten Methoden sind `put` und `get`
- `Map` ist nicht von `Iterable` abgeleitet $\Rightarrow$ kein driekter Nutzen von `for-each`

Beispiel für „Schleifen“ über Maps

```
public class MapTest {
  public static void main(String[] args) {
    var books = new HashMap<String, Book>();
    books.put("978-3-8362-3775-8", new Book("Linux -- Das umfassende Handbuch", 2015));
    books.put("978-3-8362-3778-9", new Book("Linux-Kommandoreferenz", 2016));
    books.put("978-3-8362-4220-2", new Book("Raspberry Pi -- Das umfassende Handbuch",
        2016));
    // Schleife ueber alle Keys
    for (String isbn : books.keySet()) {
      Book b = books.get(isbn);
      System.out.format("ISBN: %s\n", isbn);
      System.out.format("  %s (%d)\n", b.title, b.published);
    }
```

Beispiel für „Schleifen“ über Maps

```
    // Schleife ueber Values
    for (Book b : books.values()) {
      System.out.format("%s (%d)\n", b.title, b.published);
    }
    // Schleife ueber Values + Keys gemeinsam
    // e hat den Typ Map.Entry<String, Book>
    for (var e : books.entrySet()) {
      System.out.format("ISBN: %s\n", e.getKey());
      Book b = e.getValue();
      System.out.format("  %s (%d)\n", b.title, b.published);
    }
    // Schleife mit forEach
    books.forEach((isbn, b) -> {
      System.out.format("ISBN: %s\n", isbn);
      System.out.format("  %s (%d)\n", b.title, b.published);
    });
  }
}
```