universität innsbruck

Collections

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Motivation

- Beim Programmieren werden sehr oft bestimmte Datenstrukturen benötigt.
  - Datenstrukturen sollten schon vorhanden sein.
  - Datenstrukturen sollten generisch sein.
  - Allgemeine Algorithmen (anwendbar auf unterschiedliche Datenstrukturen) sollten vorhanden sein.
- Lösung?
  - Java Collections-Framework

Grundlagen (1)

- Das Collections-Framework befindet sich im Paket `java.util`.
- Für unterschiedliche Datenstrukturen stehen unterschiedliche Interfaces zur Verfügung.
  - Die Interfaces geben die Schnittstellen für den Zugriff auf die Datenstrukturen an (z.B. Einfügen in Listen).
  - Die konkrete Verwaltung wird durch konkrete Implementierungen dieser Interfaces bestimmt.
- Jedes Interface kann durch unterschiedliche Implementierungen unterstützt werden (z.B. `LinkedList`, `ArrayList`).
- Es gibt Algorithmen, die auf Collection-Objekte (Listen etc.) angewandt werden können (z.B. Sortieren).

Grundlagen (2)

- Im Gegensatz zu Arrays können in Collections keine primitiven Datentypen gespeichert werden, da Collections immer mit Objekten arbeiten.
- Durch Autoboxing und Autounboxing können dennoch primitive Datentypen aufgenommen und abgefragt werden.
- Mit der erweiterten `for`-Schleife (foreach) kann über Collections, die das `Collection`-Interface implementieren, iteriert werden.
- Seit Java 5 sind alle Klassen und Interfaces generisch (im Kapitel Generics mehr dazu). Durch die Angabe von Typparametern kann der Typ der Datenelemente festgelegt werden.

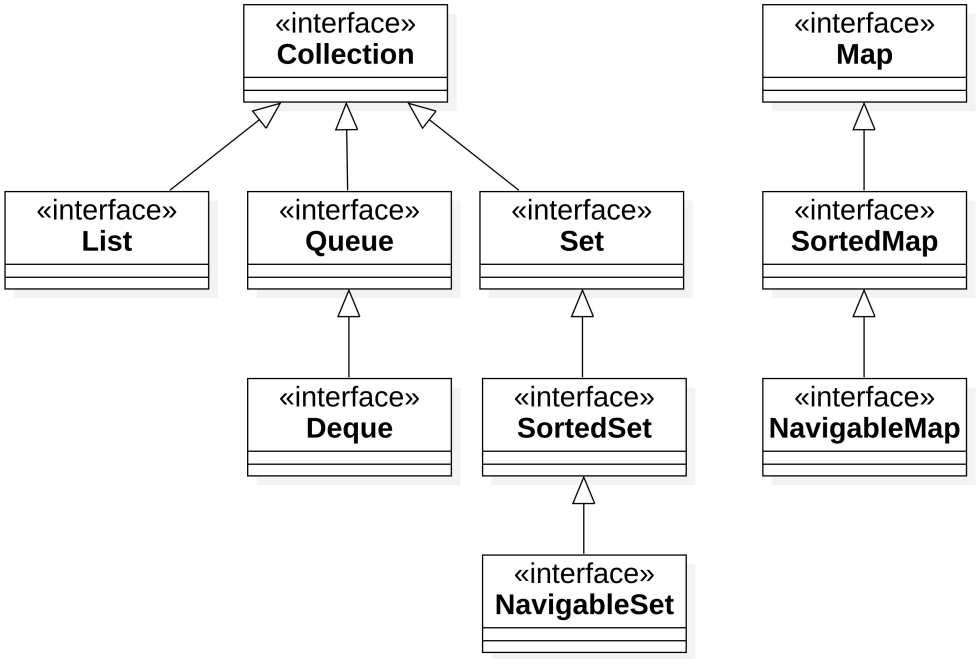
```
Collection<String> collection = new ArrayList<String>();
```

- Seit Java 7 muss bei der Deklaration der Typparameter nicht wiederholt werden.

```
Collection<String> collection = new ArrayList<>();
```

Interfaces (Überblick)

- Interfaces sind die Grundlage (java.util.Collection oder java.util.Map)
  - Basisoperationen (Hinzufügen, Finden, etc.)
  - Mengenoperationen (andere Collection einfügen)
  - Feldoperationen (z.B. andere Ansichten auf key/value)
- Applikationen sollten ihren Code ausschließlich auf Interfaces aufbauen → Austauschbarkeit.
- Interfaces bilden Hierarchien.



Collection-Interface

- Eine Collection ist eine Sammlung von Elementen.
- Es gibt keine direkte Implementierung dieses Interfaces.
- Das Interface bietet keinen indizierten Zugriff.
- Auszug einiger Methoden:
  - `contains(object)` – Ermittelt, ob das Element `object` enthalten ist
  - `isEmpty()` – Ermittelt, ob keine Elemente gespeichert sind
  - `iterator()` – Gibt einen Iterator zurück
  - `size()` – Anzahl der Elemente, die gespeichert sind
  - `stream()` – Gibt einen Stream zurück
- Auszug einiger optionaler Methoden:
  - `add(element)` – Fügt `element` in die Collection ein
  - `addAll(collection)` – Fügt alle Elemente in die Collection ein
  - `clear()` – Entfernt alle Elemente aus der Collection
  - `remove(object)` – Entfernt das Object `object` aus der Collection

List-Interface

- Erweitert das Interface `Collection`.
- Eine geordnete Collection, welche identische Elemente beinhalten kann.
- Elemente können über einen Index angesprochen werden (nicht immer effizient).
- Auszug einiger Methoden:
  - `get(index)` – Zugriff auf das Element an Position `index`
  - `indexOf(object)` – Ermittelt den Index des ersten Vorkommnis
  - `listIterator()` – Gibt einen `ListIterator` zurück
- Auszug einiger optionaler Methoden:
  - `add(element)` – Einfügen von `element` an der letzten Position
  - `add(element, index)` – Einfügen von `element` an Position `index`
  - `remove(index)` – Entfernen des Elements an Position `index`
  - `set(index, element)` – Ersetzt das Element an Position `index`

ArrayList-Klasse

- Arraybasierte Implementierung des `List`-Interfaces.
- Alle optionalen Operationen des `List`-Interfaces werden implementiert.
- Es können `null`-Elemente eingefügt werden.
- Wenn die Anzahl der zu speichernden Elemente steigt, wird immer dafür gesorgt, dass die Kapazität des Arrays ausreicht.
- Die erwartete Maximalgröße kann als Konstuktorparameter übergeben werden. Wird diese nicht angegeben, ist die Kapazität zu beginn 10.
- Die Arraygröße wird bei Löschoperationen nicht angepasst.
- Performance:
  - Indexzugriff auf Elemente ist überall ungefähr gleich schnell.
  - Einfügen und Löschen ist am Listenende schnell und wird mit wachsender Entfernung vom Listenende langsamer.

LinkedList-Klasse

- Implementierung des `List`- und `Deque`-Interfaces basierend auf einer doppelt verketteten Liste.
- Alle optionalen Operationen des List-Interfaces werden implementiert.
- Es können `null`-Elemente eingefügt werden.
- Jeder Knoten speichert eine Referenz auf die Daten, den Vorgänger und den Nachfolger.
- Im Vergleich zur Klasse `ArrayList` ist der Speicherverbrauch pro Element höher. Allerdings wird immer nur soviel Speicher aufgewendet, wie tatsächlich benötigt wird.
- Performance:
  - Indexzugriff auf Elemente ist an den Enden schnell und wird mit der Entfernung von den Enden langsamer.
  - Einfügen und Löschen ohne Indexzugriff ist überall gleich schnell.
  - Ansonsten abhängig vom Indexzugriff.

ArrayList vs. LinkedList

- Zur Veranschaulichung wurde eine Messung der Ausführungszeit für die Operation `add` und `remove` mit jeweils 100.000 Elementen, wobei die Elemente jeweils am Anfang, in der Mitte und am Ende eingefügt bzw. gelöscht wurden, durchgeführt.
- Die konkreten Zahlen sind irrelevant. Die Größenordnung ist von Bedeutung!
- Einfügen:

	Anfang	Mitte	Ende
ArrayList<>()	519,9 ms	230,5 ms	1,9 ms
ArrayList<>(100000)	475,2 ms	228,5 ms	1,1 ms
LinkedList<>()	4,3 ms	4403,1 ms	3,3 ms

- Löschen:

	Anfang	Mitte	Ende
ArrayList<>()	484,3 ms	234,1 ms	1,3 ms
ArrayList<>(100000)	470,5 ms	227,3 ms	0,6 ms
LinkedList<>()	3,3 ms	4368,0 ms	1,9 ms

Beispiel Inventory

- Inventarverwaltung für Produkte
- Interface Inventory

```
public interface Inventory {

    void printInventory();

    void removeProduct(Product product);

    void removeUnavailableProducts();

    void addProduct(Product product);

    void addProducts(Collection<Product> products);

    Collection<Product> getProducts();
}
```

- Implementierung einmal mittels Map, einmal mittels List

src/at/ac/uibk/pm/collection/productinventory/Inventory.java

Beispiel Inventory mit Liste

```
public class ListInventory implements Inventory {
    private List<Product> inventoryList = new ArrayList<>();
    …
    public void addProduct(Product product) {
        inventoryList.add(product);
    }

    public void addProducts(Collection<Product> products) {
        inventoryList.addAll(products);
    }

    public void printInventory() {
        for (Product product : inventoryList) {
            System.out.println(product);
        }
    }

    public void removeProduct(Product product) {
        for (int i = inventoryList.size() - 1; i >= 0; --i) {
            if (inventoryList.get(i).equals(product) {
                inventoryList.remove(i);
            }
        }
    }
  …
}
```

src/at/ac/uibk/pm/collection/productinventory/ListInventory.java

Set-Interface

- Erweitert das Interface `Collection`.
- Ein Set ist eine Collection, die keine duplizierten Elemente enthält.
  - Es kann kein Paar an Elementen e1 und e2 geben, sodass die Elemente e1 und e2 laut der Methode `equals` gleich sind.
- Es werden keine zusätzlichen, abstrakten Methoden zu den aus dem `Collection`-Interface vorhandenen Methoden deklariert.
  - Bei der Methode `add` werden zusätzliche Bedingungen eingeführt, um Duplikatfreiheit zu gewährleisten.

TreeSet-Klasse

- Implementierung des `NavigableSet`-Interfaces basierend auf einem Rot-Schwarz-Baum.
- Die Sortierung der Elemente wird entweder durch das Interface `Comparable` oder mit einem im Konstruktor übergebenen `Comparator` bestimmt.
- Die Sortierung basierend auf dem Interface `Comparable` wird als natürliche Ordnung bezeichnet.
- Bei der natürlichen Ordnung sind keine `null`-Elemente erlaubt.
- Performance:
  - Die Geschwindigkeit von Einfügen, Suchen und Löschen ist proportional zum Logarithmus der Anzahl der Elemente.

HashSet-Klasse

- Implementierung des `Set`-Interfaces basierend auf einer `HashMap`.
- `null`-Elemente sind erlaubt.
- Beim Iterieren über das HashSet darf sich niemals auf eine Reihenfolge verlassen werden.
- Performance:
  - Einfügen, Suchen und Löschen sind immer gleich schnell.
  - Aber: Rehashing kann zu Performance-Problemen führen.

```
Integer[] values = {150, 100, 50};
Set<Integer> set = new HashSet<>(Arrays.asList(values));
System.out.println(set);

Integer[] moreValues = {1, 2, 3};
set.addAll(Arrays.asList(moreValues));
System.out.println(set);
```

```
Ausgabe:
[50, 100, 150]
[1, 50, 2, 3, 100, 150]
```

Map-Interface (1)

- Arrays und Collections speichern einzelne Werte des Elementtyps.
- Arrays werden über Indexwerte adressiert. Der Typ des Index ist `int`.
- Maps sind eine Verallgemeinerung von Arrays mit einem beliebigen Indextyp.
  - Beispiel Telefonbuch
    - Name mit Telefonnummer verknüpft.
    - „Indextyp“ ist der Name (z.B. vom Typ String).
- Ein Map-Eintrag hat einen Schlüssel (key) und einen Wert (value).
- Eine Map ist eine Menge von Schlüssel-Werte Paaren.
  - Schlüssel müssen innerhalb einer Map eindeutig sein.
  - Werte müssen nicht eindeutig sein.
  - Jedem Wert ist genau ein eindeutiger Schlüssel zugeordnet.

Map-Interface (2)

- Das Interface `Collection` wird vom Interface `Map` nicht erweitert.
  - Die Interfaces `Collection` und `Map` sind unabhängige Basistypen.
- Auszug einiger Methoden:
  - `containsKey(key)` – Prüft, ob der Schlüssel `key` vorhanden ist
  - `containsValue(value)` – Prüft, ob der Wert vorhanden ist
  - `isEmpty()` – Prüft, ob keine Schlüssel-Wert Paare vorhanden sind
  - `get(key)` – Ermitteln den assoziierten Wertes für den Schlüssel `key`
- Auszug einiger optionaler Methoden:
  - `put(key, value)` – Einfügen von der Abbildung `key` auf `value`
  - `remove(key)` – Entfernt den Schlüssel `key` und den dazugehörigen Wert

Map-Interface (3)

- Maps und Collections können verknüpft werden.
- Zugriff auf die gespeicherten Schlüssel, Werte und Einträge:

  `Set<K> keySet()`
  - liefert alle Schlüssel einer Map in Form eines Sets.

  `Collection<V> values()`
  - liefert alle Werte der Map in Form einer Collection.

  `Set<Map.Entry<K, V>> entrySet()`
  - Liefert alle Einträge der Map in Form eines Sets.
  - Der Elementtyp ist `Map.Entry<K,V>` für eine Map mit dem Schlüsseltyp `K` und dem Werttyp `V`.
  - `Map.Entry` definiert die Methoden:
    - `K getKey()` – Gibt den Schlüssel für diesen Eintrag zurück.
    - `V getValue()` – Gibt den Wert für diesen Eintrag zurück.
- Maps kennen keine Iteratoren, Collections hingegen schon.
- Mithilfe von `keySet()`, `values()` und `entrySet()` kann somit trotzdem über eine Map iteriert werden.

Map-Interface (4)

- Die drei Methoden `keySet`, `values` und `entrySet` erzeugen keine neue Collection.
- Sie liefern sogenannte Sichten („views“).
  - Da keine Daten kopiert werden, sind diese Methoden auch bei großen Datenmengen sehr effizient.
  - Sichten greifen direkt auf die zugrunde liegende Map zu.
  - Änderungen an den Sichten wirken sich auf die darunterliegende Map aus und umgekehrt.
  - Wird eine Map geändert, dann wirkt sich das auf alle Sichten aus.

Hashbasierte Container (1)

- Die Grundidee bei hashbasierten Containern ist es Elemente entsprechend eines Schlüssels in einer indizierten Tabelle (Hashtabelle) zu speichern.
- Diese Hashtabelle hat eine bestimmte Größe.
- Jeder Eintrag in der Hashtabelle wird als Bucket bezeichnet.
- Der Index wird mittels der Hashfunktion ermittelt.
- Üblicherweise besteht die Hashfunktion aus der Komposition der zwei Funktionen:
  - Hash-Code
  - Kompressionsfunktion

Hashbasierte Container (2)

- In Java wird mit der Methode `hashCode` der Hash-Code des Objekts ermittelt und anschließend auf die Anzahl der Buckets komprimiert.
- Hashbasierte Container benötigen die Methode `equals`, um die Gleichheit von Schlüsselobjekten feststellen zu können.
- Die von der Klasse `Object` geerbte Version dieser Methode ist selten ausreichend, weil sie nur die Referenzen prüft.
- Wenn die Methode `equals` überschrieben wird, muss die Methode `hashCode` auch entsprechend überschrieben werden!

Hashbasierte Container (3)

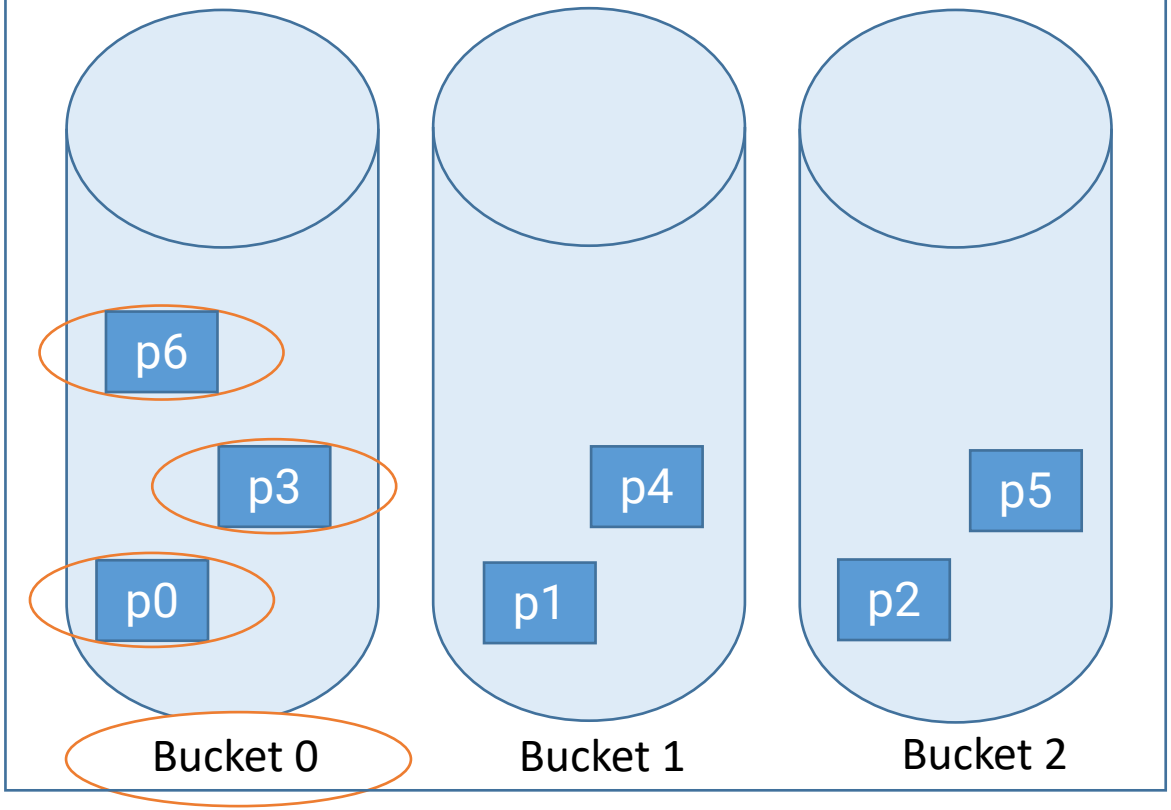
Produkte

- p0 hashCode() = 0
- p1 hashCode() = 1
- p2 hashCode() = 2
- p3 hashCode() = 3
- p4 hashCode() = 4
- p5 hashCode() = 5
- p6 hashCode() = 6

HashMap

Einfügen: `bucket_id = hashFunction(hashCode())`

Suche: `bucket_id = hashFunction(hashCode())` auf gesuchtes Objekt, danach Suche in Bucket mittels `equals()`



Anmerkungen:
(1) Wir definieren die hashCode()-Methode als `int hashCode() { return this.productID; }`.
(2) Als Hashfunktion der Datenstruktur wird `hashCode() % 3` gewählt, da in diesem vereinfachten Beispiel nur drei Buckets vorhanden sind.

Beispiel hashCode (1)

```
public class Product {
    private int productID;
    …
    public int hashCode() {
        return productID;
    }
}
```

```
Map<Product, Integer> products = new HashMap<>();
Product[] productsArray = new Product[20000];
for (int i = 0; i < productsArray.length; ++i) {
    productsArray[i] = new Product(i);
    products.put(productsArray[i], i);
}
long startTime = System.nanoTime();
for (Product product : productsArray) {
    products.get(product);
}
long endTime = System.nanoTime();
System.out.printf("Operation took: %.1f ms%n", (endTime - startTime) / 1e6);
System.out.println("Search result: " + products.get(productsArray[0]));
```

Ausgabe (hashCode version 1)

```
Operation took: 2.0 ms
Search result: 0
```

src/at/ac/uibk/pm/collection/HashMapPerformanceApplication.java

Beispiel hashCode (2)

```java
public class Product {
    private int productID;
    …
    public int hashCode() {
        return 1; // never use!
    }
}
```

```java
Map<Product, Integer> products = new HashMap<>();
Product[] productsArray = new Product[20000];
for (int i = 0; i < productsArray.length; ++i) {
    productsArray[i] = new Product(i);
    products.put(productsArray[i], i);
}
long startTime = System.nanoTime();
for (Product product : productsArray) {
    products.get(product);
}
long endTime = System.nanoTime();
System.out.printf("Operation took: %.1f ms%n", (endTime - startTime) / 1e6);
System.out.println("Search result: " + products.get(productsArray[0]));
```

Ausgabe (hashCode version 2)

```
Operation took: 2924.4 ms
Search result: 0
```

src/at/ac/uibk/pm/collection/HashMapPerformanceApplication.java

Beispiel hashCode (3)

```
public class Product {
    private int productID;
    …
    public int hashCode() {
        return (int) (Math.random() * INTEGER.MAX_VALUE); // never use!
    }
}
```

```
Map<Product, Integer> products = new HashMap<>();
Product[] productsArray = new Product[20000];
for (int i = 0; i < productsArray.length; ++i) {
    productsArray[i] = new Product(i);
    products.put(productsArray[i], i);
}
long startTime = System.nanoTime();
for (Product product : productsArray) {
    products.get(product);
}
long endTime = System.nanoTime();
System.out.printf("Operation took: %.1f ms%n", (endTime - startTime) / 1e6);
System.out.println("Search result: " + products.get(productsArray[0]));
```

Ausgabe (hashCode version 3)

```
Operation took: 2.7 ms
Search result: null
```

src/at/ac/uibk/pm/collection/HashMapPerformanceApplication.java

HashMap-Klasse (1)

- Implementierung des Interfaces Map.
- null-Elemente sind erlaubt.
- Beim Iterieren über die HashMap darf sich niemals auf eine Reihenfolge verlassen werden. Diese basiert auf der Verteilung der Elemente auf die Buckets.
- Der Füllgrad und die initiale Kapazität, sind wichtige Parameter für die Performance. Beide können über den Konstruktor gesetzt werden.
- Je kleiner der Füllgrad, desto geringer ist die Wahrscheinlichkeit für Kollisionen, allerdings steigt der Speicherbedarf.
- Als Default-Füllgrad wird 0,75 verwendet.
- Wenn das Produkt aus Füllgrad und Kapazität größer oder gleich der Anzahl der gespeicherten Elemente ist, wird die Hashtabelle automatisch vergrößert und Rehashing wird durchgeführt.

HashMap-Klasse (2)

- Die Größe von HashMaps kann nicht verringert werden.
- Performace:
  - Einfügen, Suchen und Löschen sind immer gleich schnell.
  - Aber: Rehashing kann zu Performance-Problemen führen

Beispiel Inventory mit Map

```
public class MapInventory implements Inventory {
    private Map<Integer, Product> inventoryMap = new HashMap<Integer, Product>();
    …
    public void addProduct(Product product) {
        inventoryMap.put(product.getProductID(), product);
    }

    public void addProducts(Collection<Product> products) {
        for (Product product : products) {
            inventoryMap.put(product.getProductID(), product);
        }
    }

    public void printInventory() {
        for (Product product : inventoryMap.values()) {
            System.out.println(product);
        }
    }

    public void removeProduct(Product product) {
        inventoryMap.remove(product.getProductID());
    }
    …
}
```

src/at/ac/uibk/pm/collection/productinventory/MapInventory.java

Entscheidungshilfe – Wahl der Datenstruktur

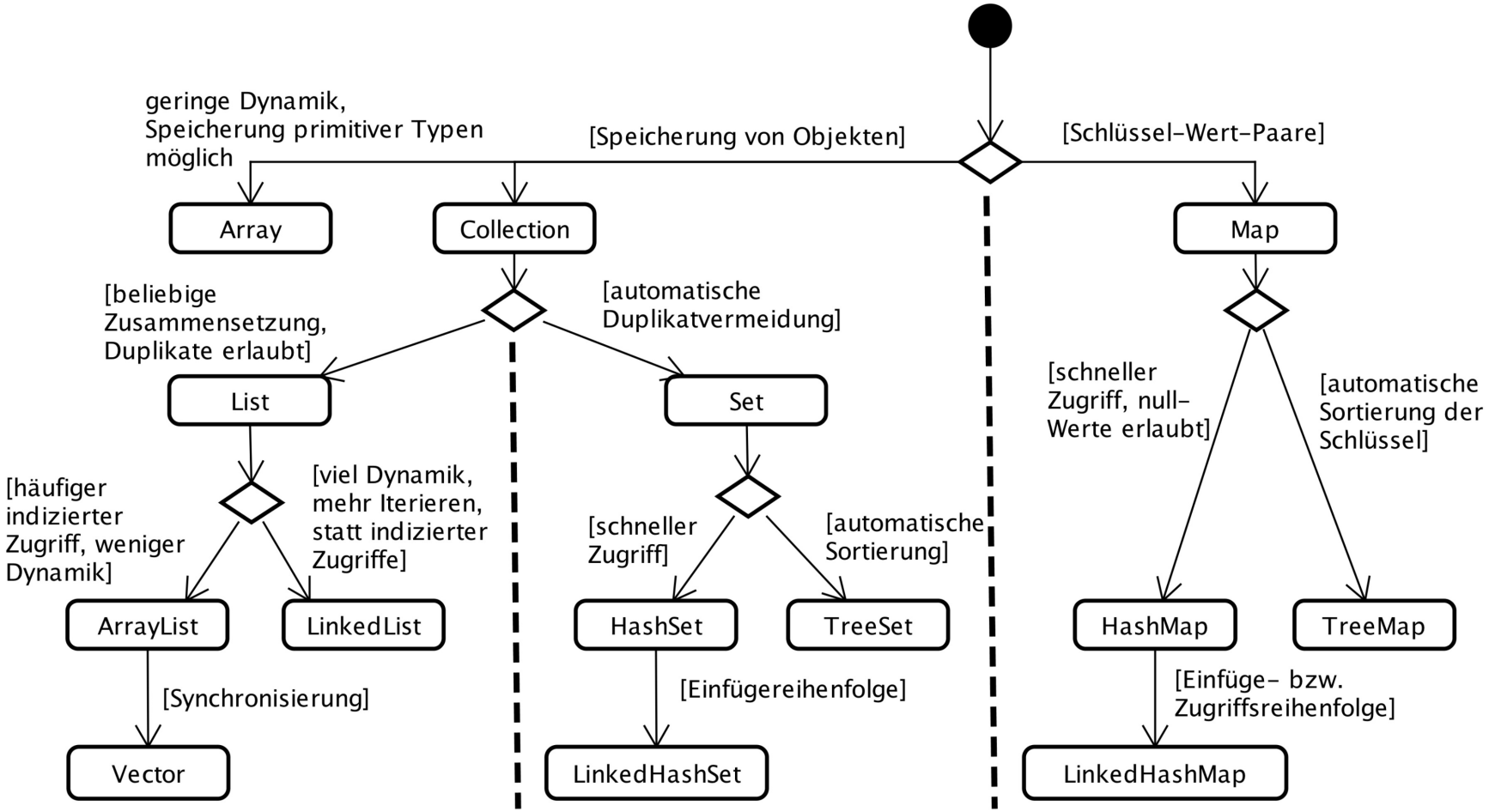


Abbildung entnommen aus „Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung.“, Michael Inden

Collections-Klasse

- Stellt statische Methoden zur Bearbeitung und Erzeugung von Collections zur Verfügung.
- Beispiele:
  - `reverse(list)`
    - „Umdrehen“ der Liste
  - `shuffle(list)`
    - Permutieren der Elemente der Liste
  - `min(collection)`, `max(collection)`
    - Zum Finden des kleinsten bzw. größten Objekts in der Collection.
    - Alle Elemente müssen das Interface `Comparable` implementieren.
  - `sort(list)`
    - Sortieren der Elemente
    - Alle Elemente müssen das Interface `Comparable` implementieren.
  - `unmodifiableList(list)`, `unmodifiableSet(set)`, …
    - Erstellt eine nicht-modifizierbare Sicht

Handling von Collections

```
public List<Product> getProducts() {
    return inventoryList;
}
```

Avoid Leaking References

```
public List<Product> getProducts() {
    return Collections.unmodifiableList(inventoryList);
}
```

- Vorher:
  - Interner Zustand wird nach außen gegeben (Referenz auf Liste wird übergeben)
  - Objekte können abgeändert werden
- Nachher:
  - Neue Liste wird zurückgegeben, die immutable (nicht abänderbar) ist (das Schlüsselwort final würde sich nur auf die Nicht-Abänderbarkeit der Referenz beziehen, nicht auf die Inhalte)
  - Änderung der (internen) Datenstruktur ausgeschlossen
  - Fehlerquelle eliminiert
  - Kapselung besser umgesetzt

Collection-Factory-Methoden (1)

- Die Collection-Factory-Methoden bieten einen Möglichkeit unveränderbare Collections zu erzeugen.
  - Die Collections basieren auf speziellen inneren Klassen, die auf Arrays basieren.
  - Diese Implementierungen zeichnen sich dadurch aus, dass sie weniger Speicherplatz verbrauchen als die entsprechenden Klassen aus dem Collections-Framework.
  - Jede Veränderung (Löschen, Hinzufügen, etc.) führt zu einer `UnsupportedOperationException`.
- Die Interfaces `List`, `Set` und `Map` stellen die Methode `of()` zur Verfügung.
  - Diese Methoden sind explizit für 0 bis 10 Parameter sowie mit einem Vararg-Parameter überladen
- Die Interfaces `Map` stellen die Methode `ofEntries()` zur Verfügung.
- Zum Erstellen von Kopien wird von den Interfaces `List`, `Set` und `Map` die Methode `copyOf()` bereitgestellt.

Collection-Factory-Methoden (2)

- `null`-Werte sind nicht erlaubt und führen zu einer `NullPointerException`.
- Duplizierte Wert bei der Erstellung eines Sets mit den Factory-Methoden führen zu einer `IllegalArgumentException`.
- Duplizierte Keys bei der Erstellung einer Map mit den Factory-Methoden führen zu einer `IllegalArgumentException`.
- Es gibt keine Collection-Factory-Methoden, welche eine Sortierung wie bei `TreeSet` oder `TreeMap` unterstützen.

```
List<String> list = List.of("one", "two", "three");

Set<String> set = Set.of("one", "two", "three");

Map<String, Integer> map = Map.of("one", 1, "two", 2, "three", 3);

Map<String, Integer> map2 = Map.ofEntries(
    Map.entry("one", 1),
    Map.entry("two", 2),
    Map.entry("three", 3));
```

Quellen

- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman: **The Java® Language Specification (***Java SE 17 Edition***)**, Oracle, 2021
- Michael Inden: **Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung**, dpunkt.verlag, 5. Auflage, 2021
- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)
- Simon Harrer, Jörg Lenhard, Linus Dietz: **Java by Comparison: Become a Java Craftsman in 70 Examples**, The Pragmatic Programmers, LLC, 2018