Programmierung: Java 2

Jürgen Hermes – IDH – SoSe 2025

Java Collections Framework – Themen

- Konzepte: Iterator, Iterable, Generics
- Lists – Konzepte und Funktionalität
  - Implementation: ArrayList
  - **Implementation: LinkedList**
- **Sets – Konzepte und Funktionalität**
  - **Exkurs: Objekt-Gleichheit**
- **Maps – Konzepte und Funktionalität**
  - **Implementation: HashMap**
  - Exkurs: Rekursion
  - Implementation: TreeMap

Über Listen

- Grundidee: Endliche Anzahl von geordneten Objekten.
- Typ der Ordnung initial meist Reihenfolge der Einspeisung in die Liste.
- Zugriff über indexierte Elemente (wie bei Arrays, Startindex 0)
- Zentrale Methoden des JCF-Interfaces java.util.List (teilweise geerbt):
  - Einfügen oder Löschen: add, addAll, set, remove, removeAll, retainAll, clear ...
  - Auffinden oder geliefert bekommen: contains, indexOf, get, subList, lastIndexOf ...
  - Weiteres: iterator, size, sort, clear, isEmpty, toArray
- Implementationen: `ArrayList` vs. `LinkedList`

ArrayList vs. LinkedList

	ArrayList	**LinkedList**
Speicherstruktur	dynamisches Array, in einem Block gespeichert	(doppelt) verkettete Knoten, verteilt über Speicher
Zugriff	direkter Indexzugriff (schnell)	lineares Durchlaufen (langsam)
Manipulationen	Kopien aller Daten v. a. bei Einfügen / Löschen in der Mitte (langsam)	Lediglich Änderungen einzelner Zeiger nötig (schnell)
Speicherbedarf	nur enthaltene Elemente (kompakter)	zusätzliche Zeiger pro Element nötig
Anwendungsfall	häufige Lesezugriffe, seltene nachträgliche Änderungen	seltene Lesezugriffe, häufige nachträgliche Änderungen

Über Sets

- Grundidee: Repräsentation von Mengen
  - Jedes Element wird nur einmal gespeichert (benötigt Identitätsüberprüfung!)
  - Elemente sind ungeordnet (implementierungsabhängig)
  - Mengenoperationen wie Schnitt, Vereinigung, Differenz
- Sets im JCF
  - Methoden vom Collection-Interface `add()`, `remove()`, `contains()`, `size()`, `isEmpty()`
  - Methoden für Mengenoperationen vom Set-Interface `retainAll()`, `addAll()`, `removeAll()`
  - Implementation: `HashSet` (basierend auf Hash-Tabelle) vs. `TreeSet` (basierend auf sortiertem binärem Suchbaum)

Grundlagen: Objekt-Gleichheit

- Bisher: Verwendung von ==
  - `true`, wenn die Objektadresse (Speicherreferenz) gleich ist
  - `false` in allen anderen Fällen
- Neu: Methode `boolean equals (Object other)`, von der Klasse Object an alle Java-Klassen vererbt. Default: Wie ==, kann abweichend implementiert werden.
  - reflexiv: `a.equals(a) == true`
  - symmetrisch: `a.equals(b) == b.equals(a)`
  - transitiv: `a.equals(b) & b.equals(c) -> a.equals(c)`
  - consistent: (keine Veränderung über die Laufzeit)
- Auch neu: Methode int hashCode(). Für schnelle Vergleiche. Sollte für Objekte, für die equals `true` zurückgibt, den gleichen int-Wert liefern.

Über Maps

- Grundidee: Mapping von Objekten auf Schlüssel (wie bei Array int-Positionen auf Objekte gemappt werden, aber sehr viel flexibler)
- Interface `Map<K,V>`: Ungeordnetes Mapping von eindeutigen K(eys) auf (nicht unbedingt eindeutige) V(alues)
- Zentrale Methoden:
  - `V put(K key, V value)` – Fügt das Key-Value-Paar der Map hinzu, falls es zum K schon ein V gab, wird dieses zurückgeliefert (und das neue stattdessen eingefügt).
  - `boolean containsKey(Object key)` – `true`, wenn K vorhanden ist, sonst `false`
  - `V get(Object key)` – Liefert den V zum K zurück, sofern einer vorhanden (sonst `null`)
  - `Set<K> keySet()` – Liefert ein Set über alle K (z.B. zum Iterieren)
  - `Collection<V> values()` – Liefert eine Collection über alle V

Implementation: HashMap

- Basiert auf hashCode und equals für die Keys
- Intern durch ein Array spezifischer Länge verwaltet (sogen. buckets)
- LinkedList über K/V-Nodes auf den Positionen des Arrays
- Eintragen und Retrieval:
  - Identifizierung des richtigen buckets durch hashCode
  - Identifizierung des richtigen Listenobjekts durch equals
- Transformation großer Hashcodes zu kleinen Arraypositionen z.B. über Modulo-Operator %.
- Für Performanz-Management der Anzahl der Buckets notwendig.