

Algorithmen und Datenstrukturen

Datenstrukturen: Collections - Refresh

Roland Gisler

Lucerne University of
Applied Sciences and Arts

**HOCHSCHULE
LUZERN**

INFORMATIK



Inhalt

- Was sind Datenstrukturen?
- Eigenschaften von Datenstrukturen
- Java Collection Framework
- Beispiel: **List** und **ArrayList**
- Einstiegsübung: Speicherverwaltung
- Zusammenfassung

Was sind Datenstrukturen?

Was sind Datenstrukturen?

- Sehr häufig haben wir es in der Programmierung mit Mengen von Daten bzw. Objekten zu tun.

Beispiele:

- Messwerte (z.B. **Temperature**)
- Bankkonten (mit Besitzer und Kontostand)
- Formen (wie z.B. **Circle** oder **Rectangle**)
- Personen (z.B. Studierende)



- Datenstrukturen werden verwendet, um **Mengen** von Daten bzw. Objekten **effizient zu speichern** und **verarbeiten** zu können.
- Unter Verarbeiten versteht man Funktionen, wie z.B. alle enthaltenen Daten/Objekte einzeln zu bearbeiten, nach bestimmten Objekten zu suchen, zu zählen, zu filtern oder nach unterschiedlichen Kriterien zu sortieren etc.

Verschiedene Arten von Datenstrukturen

- Es gibt verschiedene Arten von Datenstrukturen. Beispiele:
 - **Array** – indexierte Reihung.
 - **Tree** – hierarchisch geordnete Daten (Baumstruktur).
 - **List** – einfache Reihung.
 - **Map** – Zuordnung zwischen Schlüssel und Wert(-paaren).
 - **Queue** – (Warte-)Schlange, FIFO.
 - **Set** – Sammlung ohne Duplikate.
 - **Stack** – Stapel- oder Kellerspeicher, FILO.
- Java bietet für jede Art wiederum verschiedene Implementationen dieser Datenstrukturen an, um den unterschiedlichen Anforderungen gerecht zu werden → Java Collections Framework.
- Hinweis: Eine universelle Datenstruktur, die **alles** perfekt kann, gibt es (leider) nicht, wir müssen fallspezifisch auswählen!

Eigenschaften von Datenstrukturen

Eigenschaften von Datenstrukturen

- Die verschiedenen Datenstrukturen unterscheiden sich in vielerlei Hinsicht, zum Beispiel:
 - **Grösse**: Dynamisch oder statisch.
 - **Zugriff**: Direktzugriff oder indirekt / sequenziell.
 - **Sortierung**: Sortiert oder unsortiert, mit/ohne Ordnung.
 - **Suche**: Beschleunigte Suche (z.B. binär oder über Hashwert).
 - **Geschwindigkeit**: Grundlegende Operationen wie Suchen, Einfügen, Anhängen, Entfernen, Verschieben von einzelnen Elementen an verschiedenen Positionen.
- Anhand dieser **Eigenschaften** und den teilweise damit eng verbundenen **Algorithmen**, sollten wir für jeden Anwendungsfall gezielt die geeignetste Datenstruktur auswählen!

Java Collection Framework

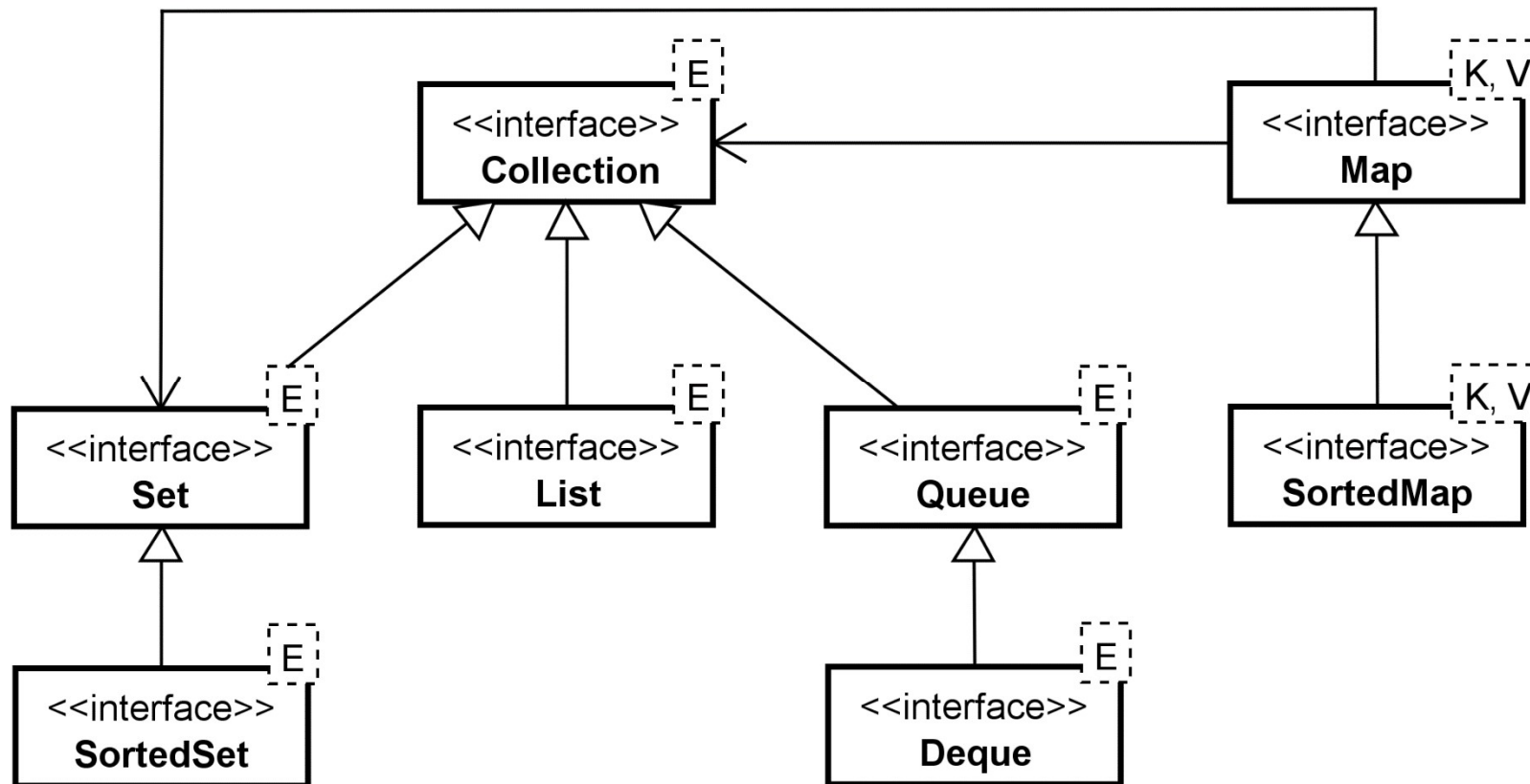
Java Collection Framework - Übersicht

Das Collection Framework besteht aus drei wesentlichen Inhalten:

- **Interfaces:** Abstrakte Datentypen, welche verschiedenartige Datenstrukturarten repräsentieren.
 - `List<E>` abstrahiert Listen, `Map<K, V>` abstrahiert Map etc.
- **Implementationen:** Konkrete, wiederverwendbare Implementierungen der durch die Schnittstellen definierten Datenstrukturen.
 - `LinkedList<E>` und `ArrayList<E>` sind zum Beispiel zwei Implementierungen von `List<E>`.
- **Algorithmen:** Methoden zur Behandlung von Datenstrukturen. Meist polymorph implementiert.
 - `iterator()` ermöglicht sequenziellen Zugriff auf alle Elemente, unabhängig von der konkreten Implementation.
 - `sort(List<E>)` sortiert z.B. beliebige `List`-Implementationen.

Java Collection Framework – Interfaces - Übersicht

- Auszug aus den vorhandenen Schnittstellen (nicht vollständig):



Java Collection Framework - Funktionsbasis

- Damit die polymorphen Algorithmen und Datenstrukturen des Collection Frameworks mit **beliebigen** Elementen funktionieren können, benötigen wir ein paar grundlegende Funktionen:
 - Für die **Suche** von Objekten (oder Verhindern von Duplikaten) müssen wir Objekte als «**gleich**» erkennen können.
 - Für die **Sortierung** müssen wir Objekte vergleichen und als **kleiner** (<), **gleich** (=) oder **grösser** (>) beurteilen können.
- Daraus resultiert, dass
 - die Klasse **Object** (**die** Basisklasse) einige dafür notwendige Methoden definiert: **equals()** und **hashCode()**
 - es zusätzliche Interfaces gibt, die wir ggf. implementieren müssen: **Comparable<T>** oder **Comparator<T>**
- siehe OOP Input → **O10_ObjectEqualsCompare**

Beispiel: List und ArrayList

Beispiel: ArrayList im Einsatz

```
final List<Temperatur> verlauf = new ArrayList<>();
```

```
verlauf.add(new Temperatur(10.2));
```

```
verlauf.add(new Temperatur(15.8));
```

```
verlauf.add(new Temperatur(21.3));
```

```
System.out.println("Anzahl Messwerte: " + verlauf.size());
```

Anzahl Messwerte: 3

```
verlauf.set(1, new Temperatur(18.5));
```

```
final Temperatur t2 = verlauf.get(1);
```

```
System.out.println("Zweiter Messwert: " + t2.getCelsius());
```

Zweiter Messwert: 18.5

Einstiegsübung: Speicherverwaltung (E0)

Speicherverwaltung - Grundlagen

- Wie verwaltet das Betriebssystem eigentlich Speicher?
- Vereinfachte Annahme: Speicher liegt mit einem linearen Adressraum zur Verfügung.
 - Beispiel: Ab Adresse **\$00** bis **\$0F** (also gerade mal 16 Bytes)

\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07	\$08	\$09	\$0A	\$0B	\$0C	\$0D	\$0E	\$0F

- Speicher wird in Blöcken angefordert (**memory allocation**).
 - Beispiele:

```
red = malloc(4); red.write("HSLU");
```

```
blue = malloc(3); blue.write("IST");
```

```
yellow = malloc(5); yellow.write("SUPER");
```

\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07	\$08	\$09	\$0A	\$0B	\$0C	\$0D	\$0E	\$0F
H	S	L	U	I	S	T	S	U	P	E	R				

Speicherverwaltung - Fragmentierung

\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07	\$08	\$09	\$0A	\$0B	\$0C	\$0D	\$0E	\$0F
H	S	L	U	I	S	T	S	U	P	E	R				

- Speicher wird (netterweise) auch wieder freigegeben (free).
 - Beispiel: `free(blue);`

\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07	\$08	\$09	\$0A	\$0B	\$0C	\$0D	\$0E	\$0F
H	S	L	U	I	S	T	S	U	P	E	R				

- Was passiert nun, wenn man versucht einen Block mit `malloc(7)` anzufordern?
- Was passiert, wenn man auch noch `free(yellow)` freigibt?

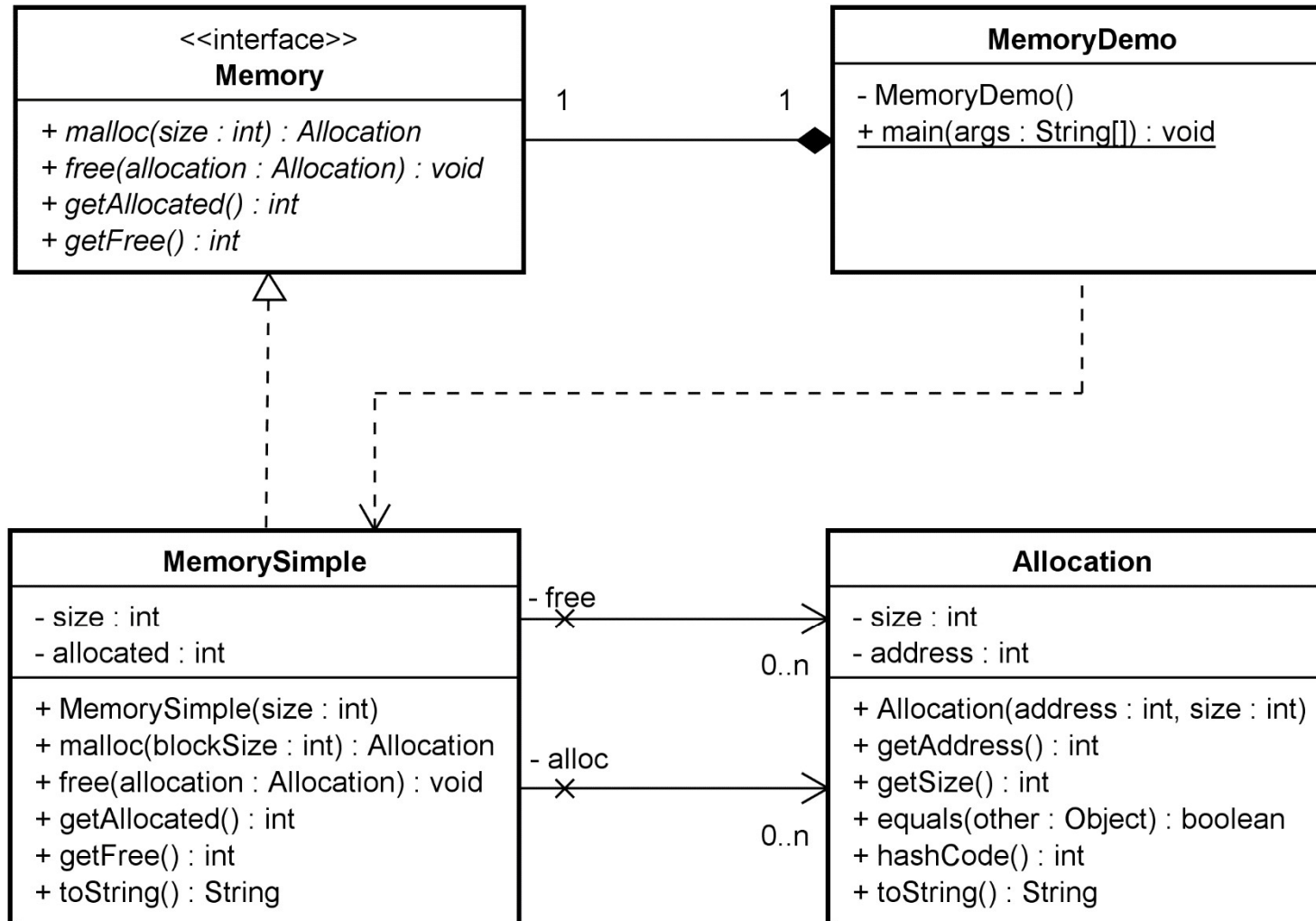
\$00	\$01	\$02	\$03	\$04	\$05	\$06	\$07	\$08	\$09	\$0A	\$0B	\$0C	\$0D	\$0E	\$0F
H	S	L	U	I	S	T	S	U	P	E	R				

Speicherverwaltung - Anforderungen

- Verwaltung soll zuverlässig sein, wertvoller Speicherplatz soll nicht verloren gehen (keine «memory leaks»).
- Allokationen sollen **schnell** sein: Keine lange Suche nach einem in der Grösse passenden Block.
- Freigabe soll ebenfalls **schnell** sein.
- Mit der Zeit entstehen Lücken sowohl durch belegten als auch freien Speicher (Fragmentierung).
- Liegen freie Lücken **nebeneinander**, sollten diese in der Datenstruktur wieder zusammengefasst werden!
- **Welche Datenstruktur(en) mit welchen spezifischen Eigenschaften (Sortierung, Suche etc.) eignet sich dafür am Besten?**

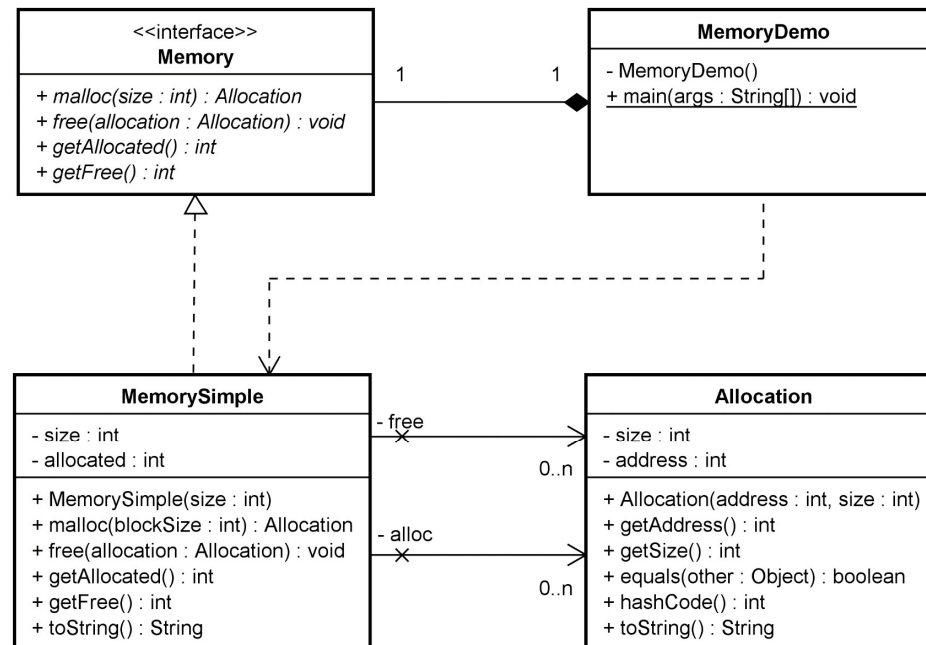
Speicherverwaltung - Modellierung

- Ein einfacher erster Ansatz zur Modellierung als Inspiration:



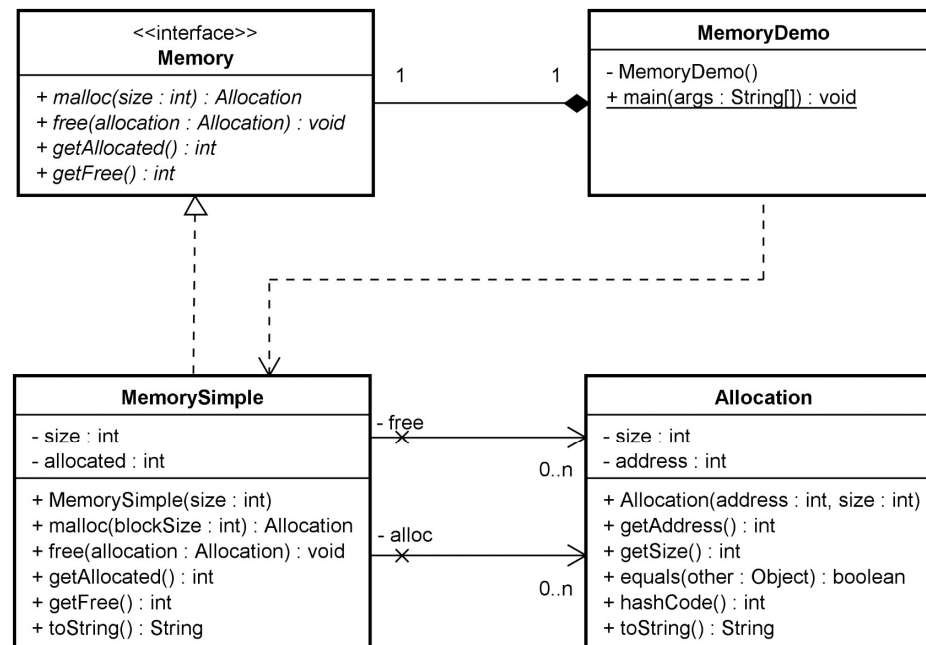
Speicherverwaltung – Modellierung – Diskussion

- Wozu dient uns das Interface **Memory**?
- Warum existieren zwei Beziehungen zwischen **MemorySimple** und **Allocation**?
- Welche Datenstrukturen können wir zur Realisierung dieser Beziehungen ($0..n$) verwenden?
- Warum ist es wichtig, dass **Allocation** die Methoden **equals()** und **hashCode()** korrekt implementiert hat?



Speicherverwaltung – Algorithmen

- Wie implementieren wird die Belegungsstrategie?
 - Nehmen wir den ersten passenden Block?
 - Nehmen wir den noch kleinsten passenden Block?
 - Nehmen wir den grössten noch vorhandenen Block?
- Freigabe von Speicher
 - Wie erkennen und verbinden wir benachbarte, freie Blöcke?
 - Wann führen wir diesen Prozess durch?
- Auswahl und Einsatz von Datenstrukturen ist meist auch stark von den eingesetzten Algorithmen abhängig.



Zusammenfassung

- Datenstrukturen dienen zur Verarbeitung von Mengen von Daten bzw. Objekten.
- Es gibt verschiedene Datenstrukturen mit unterschiedlichen Fähigkeiten und Eigenschaften.
- Das Java Collections Framework enthält Schnittstellen, Algorithmen und konkrete Implementationen von und zu Datenstrukturen.
- **ArrayList** ist ein Beispiel einer häufig eingesetzten, konkreten (Listen-)Datenstruktur in Java.
- Die Auswahl der geeigneten Datenstruktur erfolgt aufgrund der konkreten Anforderungen.
 - Aufgrund der Eigenschaften wie Zugriffsart, Art und Häufigkeit der Operationen, Datenmenge und –struktur.

Fragen?