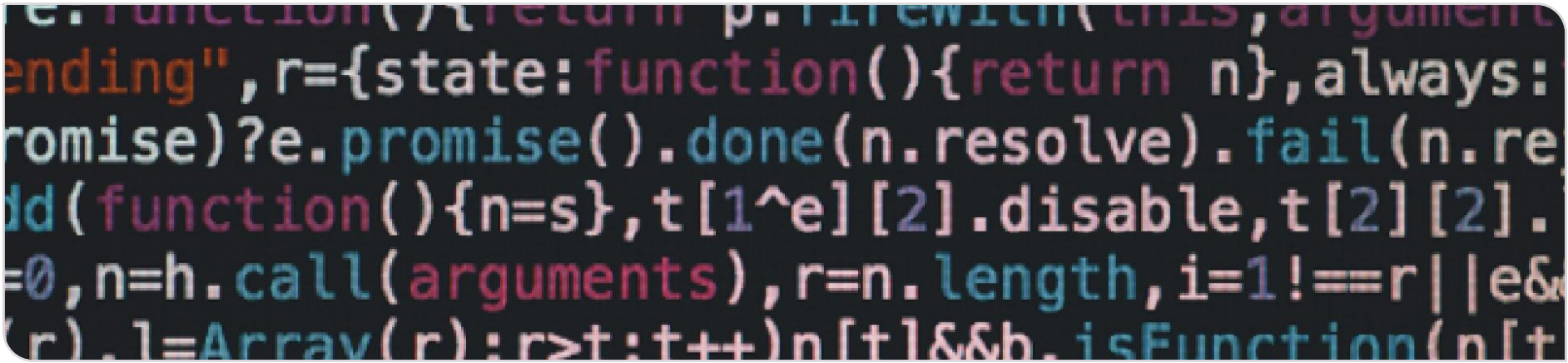
Karlsruher Institut für Technologie

Programmieren Tutorium 25

Rekursion, Java API

Benoit Legien | 11. Januar 2024



Frage im letzten Tutorium

- `ArrayList<Integer> list = new ArrayList<>();`

- Seit Java 7: “Diamond Notation”
  - Typ darf weggelassen werden, solange der Compiler den Typen aus dem Kontext ermitteln kann.

Nachtrag: Wrapper-Klassen

- Automatisches Umwandeln seit Java 5 (“Autoboxing”)

```java
Integer i = 5;
int j = i;

ArrayList<Integer> list = new ArrayList<>();
list.add(46);
```

Inhalt

1. Blatt 3 - Häufige Fehler
2. Wiederholung Interfaces, Generics
3. Rekursion
4. Java API
5. OOP-Designprinzipien
6. Schluss

Blatt 3

Häufige Fehler

Blatt 3 - Häufige Fehler (1/3)

1. String-Formatting
   - keine Abzüge, aber trotzdem good practice

```
round + ". Zug, Spieler " + player + ":";
"%d. Zug, Spieler %d:".formatted(round, player);
```

   - %s - insert a string
   - %d - insert a signed integer (decimal)

   - → Cheatsheet in Ilias?

Blatt 3 - Häufige Fehler (2/3)

1. String-Formatting
2. Klassenkonstanten statt lokale Konstanten

```java
class Application {
      public static void main(String[] args) {

            final int column = 7;
            final int row = 5;

            // …
      }
}
```

```java
class Application {

      public static final int COLUMN = 7;
      public static final int ROW = 5;

      public static void main(String[] args) {
            // …
      }
}
```

Blatt 3 - Häufige Fehler (3/3)

1. String-Formatting
2. Klassenkonstanten statt lokale Konstanten
3. toString()-Methode
   - https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html
   - @Override!
     - JavaDocs können weggelassen werden
   - `toString()` gibt String zurück statt `System.out.println(...)`

Wiederholung

Interfaces & Generics

Wiederholung - Interfaces

Welche Aussagen sind korrekt?

1. Ein Interface ist eine Sammlung von Methodensignaturen ohne Implementierung. ✔
2. Jede Klasse, die ein Interface implementiert, muss für jede Methode eine Implementierung angeben. ✗
3. In Interfaces können auch Konstanten und statische Methoden definiert werden. ✔
4. Es gibt keine Unterschiede zwischen Interfaces und abstrakten Klassen. ✗

Wiederholung - Generics

Welche Aussagen sind korrekt?

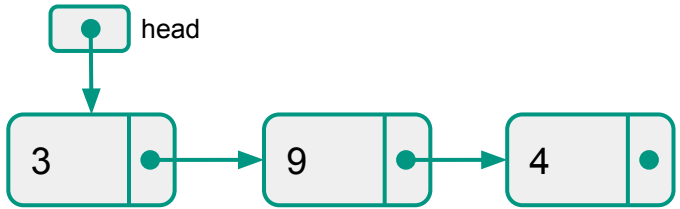
1. Generics werden eingesetzt, um Klassen zu spezifizieren. ✗
2. Typ-Parameter dürfen mit jedem Datentyp instanziiert werden. ✗
3. Typ-Parameter können mittels `extends` eingeschränkt werden ✔

Rekursion

Rekursive Datentypen

- Objekte einer Klasse enthalten Verweise auf Objekte derselben Klasse

- z.B. Liste




```
public class ListElement<T> {
     T value;
     ListElement<T> next;
}
```

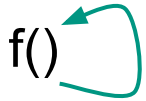
Rekursive Methoden

- Methoden rufen sich (direkt oder indirekt) selber auf

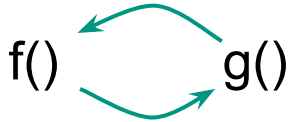
- Warum?
  - Großes, komplexes Problem → aufteilen in kleinere Probleme, die leichter lösbar sind (*Divide and Conquer / Teile und Herrsche*)
  - Anwendung vor allem in der Algorithmik → mehr in Algo1 (2. Semester)

Rekursion - Vokabeln

- Eine Methode f heißt (direkt) **rekursiv**, wenn im Rumpf von f Aufrufe von f vorkommen.



- Eine Methode f heißt **indirekt rekursiv**, wenn im Rumpf von f eine Methode g aufgerufen wird, die ihrerseits direkt oder indirekt auf Aufrufe von f führt.

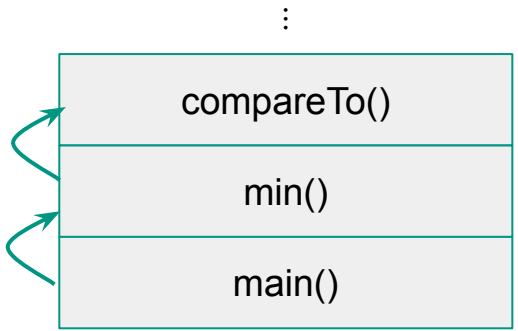


- Eine Methode f heißt **endständig rekursiv**, wenn im Rumpf von f nach dem rekursiven Aufruf von f kein weiterer Code steht.

Eigentlich erst in Propa (5. Semester) wichtig

Rekursion - Technische Sicht (vereinfacht)

- Methoden-Aufrufe liegen im “call stack”
- Jeder Methodenaufruf legt eine neuen “Frame” an



- Zu viele Methoden-Aufrufe → Stack Overflow

Welche Fragen habt ihr noch?

Rekursion - Aufgabe (1/2)

- Schreibt eine äquivalente rekursive Methode zu der folgenden Methode:

```
public static void printTimes(String output, int times) {
     for (int i = 0; i < times; i++) {
          System.out.println(output);
     }
}
```

Rekursion - Aufgabe (1/2)

- Schreibt eine äquivalente rekursive Methode zu der folgenden Methode:

- Lösung:

```java
public static void printTimes(String output, int times) {
     if (times <= 0) {
          return;
     }

     System.out.println(output);
     printTimes(output, times - 1);
}
```

Rekursion - Aufgabe (2/2)

Aufgabe: Schreibt den folgenden Sortieralgorithmus als rekursive Methode:

1. Es wird eine unsortierte Liste von Ganzzahlen als Parameter übergeben.
2. Suche das kleinste Element der unsortierten Liste und entferne es aus der Liste. (Ihr könnt davon ausgehen, dass eine Methode gegeben ist, die das minimale Element berechnet)
3. Füge das entfernte kleinste Element an das Ende einer sortierten Liste.
4. Wiederhole Schritte 2-3 mit dem Rest der unsortierten Liste.
5. Gebe die fertige sortierte Liste als Lösung zurück.

Tipp: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

```
add(E e), remove(Object o), size(), addAll(Collection<? extends E> c)
```

Lösung

```java
public static ArrayList<Integer> sortRecursively(ArrayList<Integer> unsorted) {
     ArrayList<Integer> sorted = new ArrayList<>();

     if (unsorted.isEmpty()) {    // Abbruchbedingung
          return sorted;
     }

     Integer min = min(unsorted); // Berechne das kleinste Element
     unsorted.remove(min);        // Entferne das kleinste Element aus der unsortierten Liste
     sorted.add(min);             // Füge das kleinste Element an das Ende der sortierten Liste

     ArrayList<Integer> sortedRemaining = sortRecursively(unsorted); // Sortiere die Restliste
     sorted.addAll(sortedRemaining);

     return sorted;
}
```

Java API

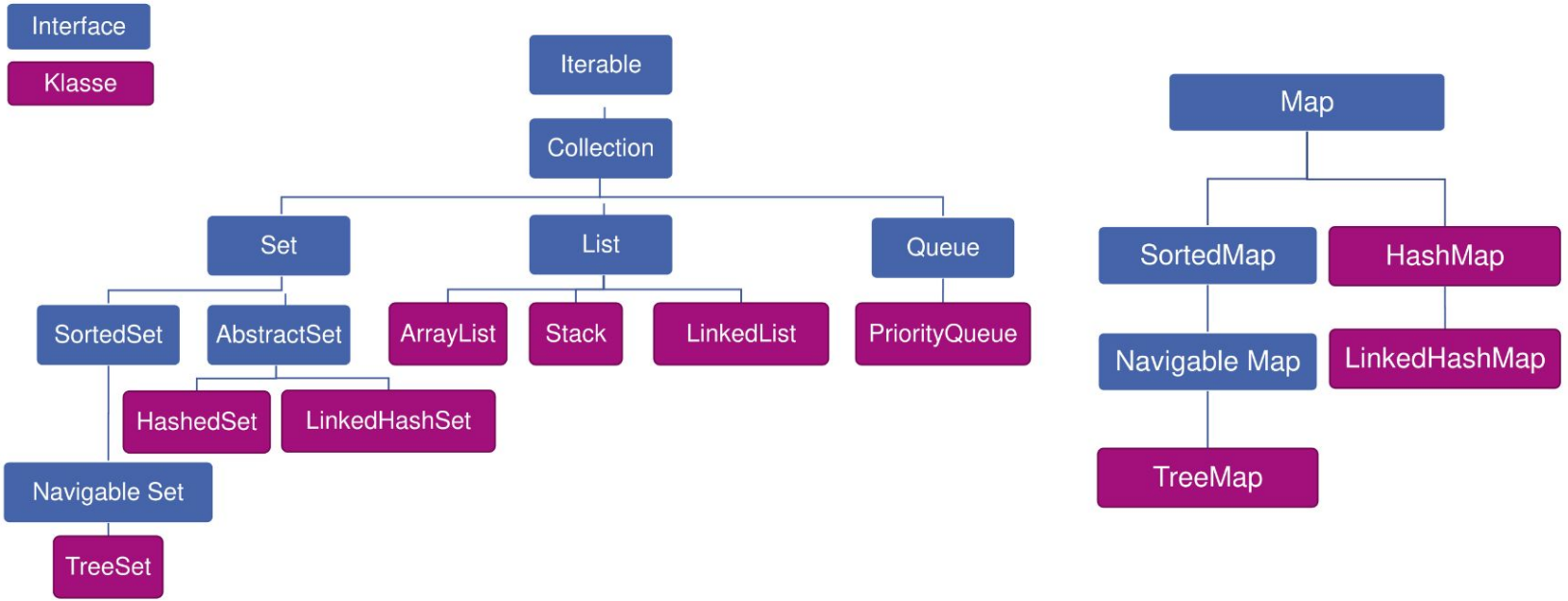
Was ist die Java API?

- Sammlung von häufig benötigten Funktionalitäten
- Ermöglicht effizientere Entwicklung

Java API - Wichtige Packages

- `java.lang` (Object, String, Enum, Math, Iterable, Byte, Float, …)
- `java.util` (Collections, Datenstrukturen, …)
- `java.io` (Input/Output, Streams, Dateien, …)
- `java.net` (Netzwerke, …)
- `java.security`
- `java.swing`, `awt`, `javafx` (GUI, …)

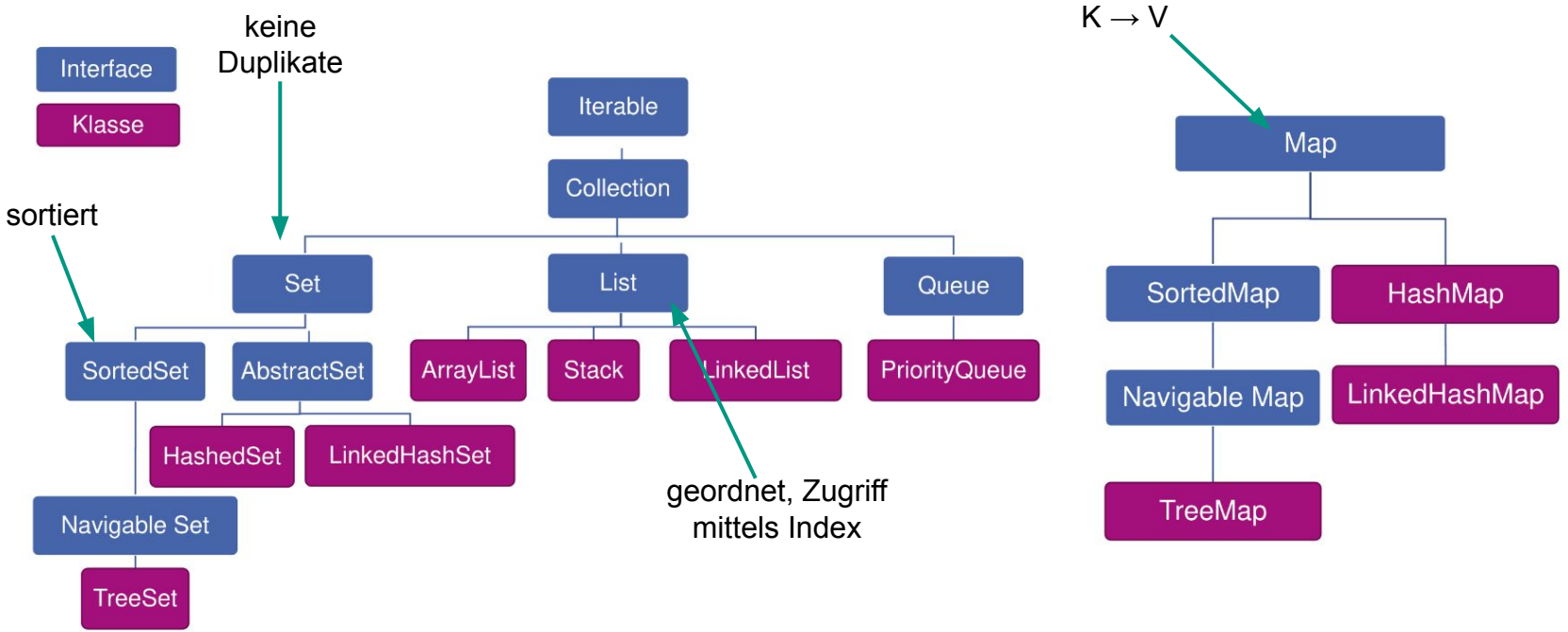
`java.util` - Wichtige Interfaces / Klassen



Collection<E>

- Für Sammlungen von Daten
- Keine Aussage, ob …
  - die Elemente geordnet sind
  - es Duplikate gibt
- Spezifischere Collections
  - `Set`, `List`, `Queue`, …

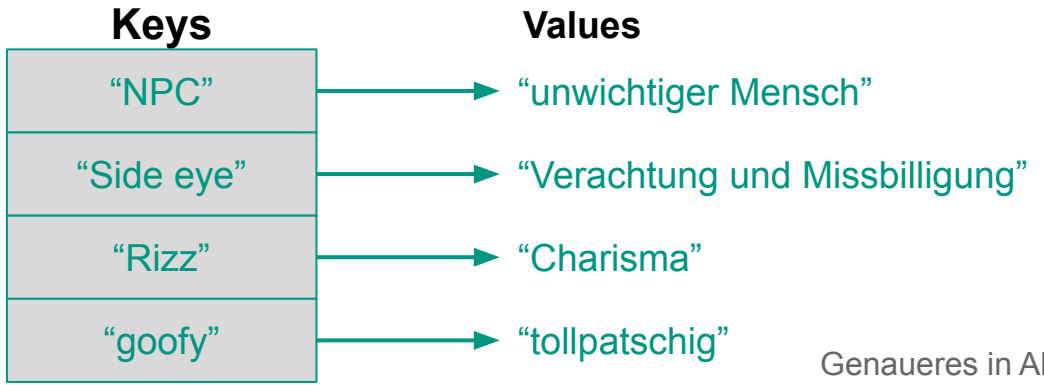
`java.util` - Wichtige Interfaces / Klassen



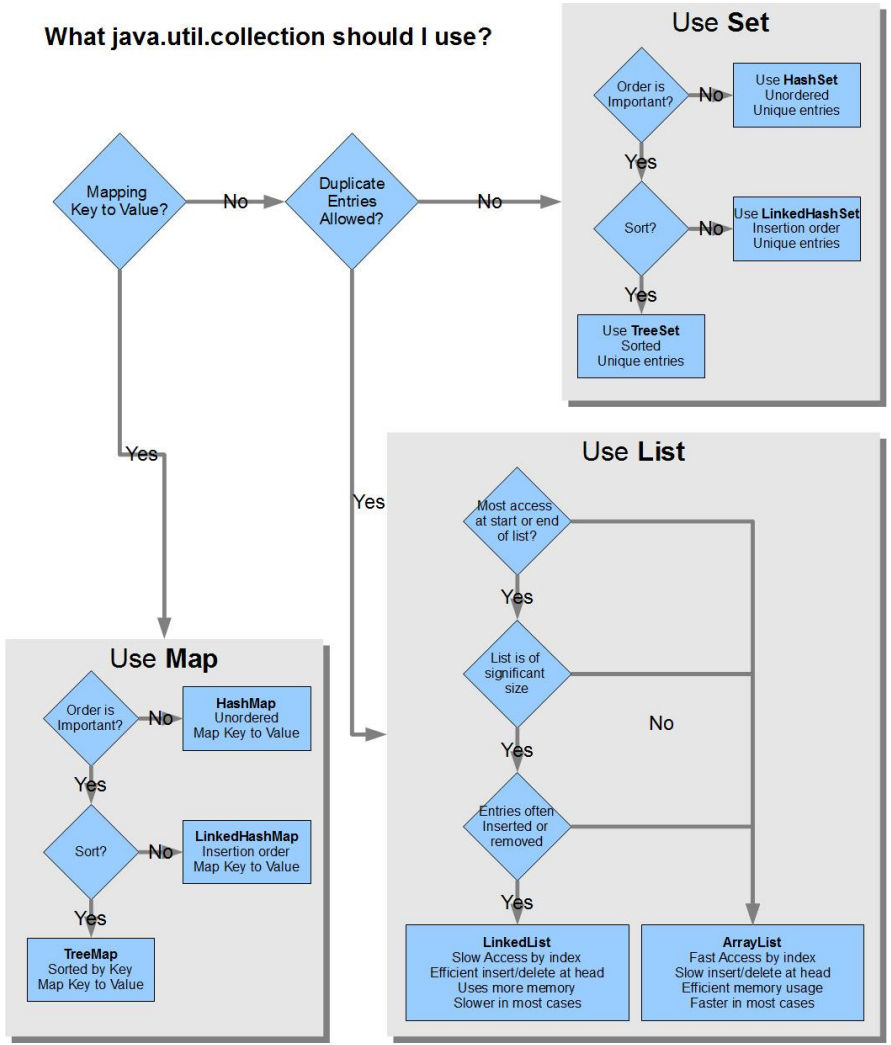
Das Interface `Map<K, V>`

- Modelliert eine mathematische Funktion K → V (linkstotal und rechtseindeutig)
  - Eine Map enthält keinen Schlüssel mehrmals
  - Jeder Schlüssel bildet auf höchstens einen Wert ab

vereinfachte Darstellung:

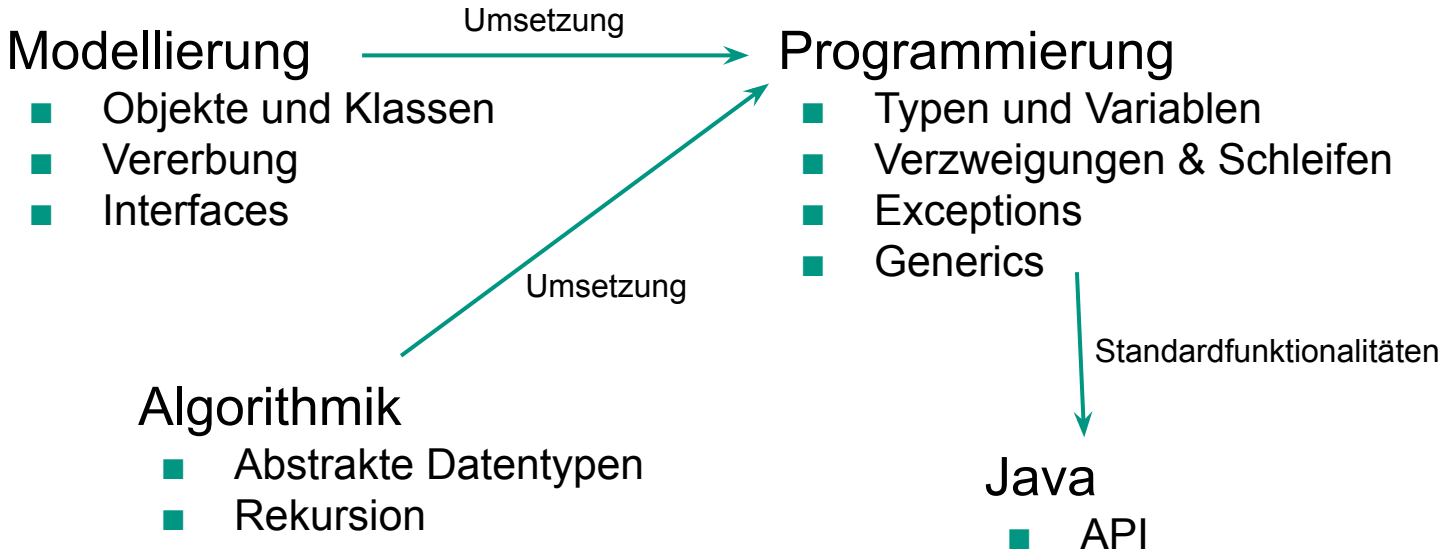


Genaueres in Algo1 (2. Semester)


What java.util.collection should I use?
Mapping Key to Value?
No
Duplicate Entries Allowed?
No
Use Set
Order is Important?
No
Use HashSet
Unordered
Unique entries
Yes
Sort?
No
Use LinkedHashSet
Insertion order
Unique entries
Yes
Use TreeSet
Sorted
Unique entries
Yes
Use Map
Order is Important?
No
HashMap
Unordered
Map Key to Value
Yes
Sort?
No
LinkedHashMap
Insertion order
Map Key to Value
Yes
TreeMap
Sorted by Key
Map Key to Value
Yes
Use List
Most access at start or end of list?
Yes
List is of significant size
No
Yes
Entries often Inserted or removed
Yes
LinkedList
Slow Access by index
Efficient insert/delete at head
Uses more memory
Slower in most cases
ArrayList
Fast Access by index
Slow insert/delete at head
Efficient memory usage
Faster in most cases

Welche Fragen habt ihr noch?

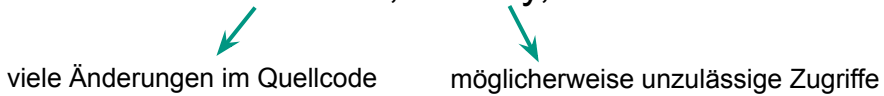
Zusammenfassung bisheriger Vorlesungen



OOP Designprinzipien

Warum Designprinzipien?

- Viele Möglichkeiten, eine Idee umzusetzen
  - Manche sind besser, manche schlechter
    - z.B. Neue Features, Library, …



- In Softwaretechnik (2. Semester): Etablierte Design-Patterns
  - Grundlagen hier schon

OOP Designprinzipien

- Prinzip 1 (Datenkapselung)
  Minimiere die Zugriffsmöglichkeiten auf Klassen und Attribute
- Prinzip 2
  Bevorzuge Komposition gegenüber Vererbung
- Prinzip 3
  Programmiere gegen Schnittstellen und nicht gegen eine Implementierung
- Prinzip 4 (Open-Closed Principle)
  Software-Komponenten sollten offen für Erweiterung, aber geschlossen für Änderung sein
- SOLID-Prinzipien

1. Datenkapselung

- Wann immer möglich: **`private`** mit Gettern/Settern verwenden!

Schlecht

```
public class Monopoly {

      public Player currentPlayer;
      …

}
```

Besser

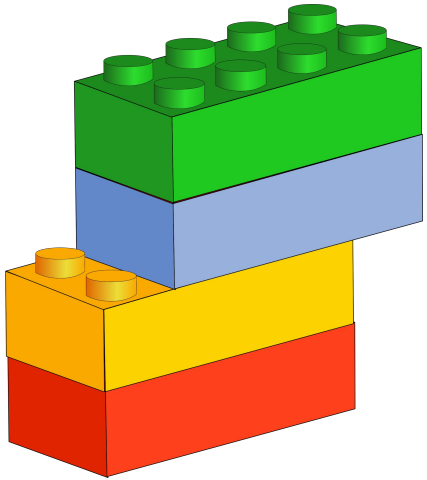
```
public class Monopoly {

      private Player currentPlayer;
      …
      public Player getCurrentPlayer() {...}
      public void setCurrentPlayer() {...}
      // oder nextPlayer() oder so
      …
}
```

Zugriff kann gesteuert werden

2. Komposition > Vererbung

Was ist Komposition?

“Wiederverwendung durch Zusammensetzung bestehender Objekte zu einem neuen mit erweiterter Funktionalität”



2. Komposition > Vererbung

Komposition

```
class Object3D {
      Vector3D position, rotation, scale;
}

class Physics3D {
      Object3D influenceOn;
      Vector3D gravity, velocity;
      public Physics3D(Object3D object) { … }
      ...
}

class Player {
      Object3D position;
      Physics3D physics;
}
```

Besser

Vererbung

```
abstract class Object3D {
      Vector3D position, rotation, scale;
      ...
}

abstract class PhysicsObject extends Object3D {
      Vector3D gravity, velocity;
      ...
}

class Player extends PhysicsObject {
      ...
}
```

Schlecht

2. Komposition > Vererbung

Sehr ausführliches Video:

[The Flaws of Inheritance - CodeAesthetic]

3. Schnittstellen statt Implementierungen

- Beispiel:
  - Erweiterung unseres 3D Games durch Fahrzeuge

zu spezifisch

```
class Car {...}

class Player {
      ...
      public void enterCar(Car car) {...}
}
```

Schlecht

möglichst unspezifisch

```
interface Vehicle {...}
class Car implements Vehicle {...}
class Truck implements Vehicle {...}
...

class Player {
      ...
      public void enterVehicle(Vehicle vehicle) {...}
}
```

Besser

4. Open-Closed-Principle (OCP)

- Offen für Erweiterung, Geschlossen für Änderung
- ⇒ bei Erweiterung neuen Code hinzufügen, nicht alten Code ändern!

```
Employee[] employees;

for (Employee employee : employees) {
      if (employee instanceof Intern) {
            employee.paySalary(0);
      } else if (employee instanceof Manager) {
            employee.paySalary(69420);
      } ...
}
```

Schlecht

Bei Erweiterung muss alter Code verändert werden!

```
Employee[] employees;

for (Employee employee : employees) {
      salary = employee.getRole().getSalary();
      employee.paySalary(salary);
}
```

Besser

SOLID-Prinzipien

1. **S**ingle Responsibility Principle
   - Jede Klasse sollte nur eine Verantwortung („Grund zur Änderung“) haben.
2. **O**pen/Closed Principle ← siehe 4.
   - Klassen sollten Erweiterungen erlauben, ohne dabei ihr Verhalten zu ändern
3. **L**iskov Substitution Principle ← Vererbung
   - Eine Instanz einer abgeleiteten Klasse sollte sich so verhalten, dass jemand, der meint, ein Objekt der Basisklasse vor sich zu haben, nicht durch unerwartetes Verhalten überrascht wird.
4. **I**nterface Segregation Principle
   - Klassen sollten durch Interface nicht gezwungen werden, nicht notwendige Methoden zu implementieren; stattdessen Interface auftrennen
5. **D**ependency Inversion Principle
   - Abstraktes soll nicht von Details abhängen

Weitere Prinzipien (für Schnittstellen)

1. Verstecken von interner Repräsentation → möglichst generelle Superklasse/Schnittstelle verwenden
2. Interfaces gut abstrahieren → nicht ein “Alles-Interface” sondern sinnvoll trennen ← siehe SOLID Interface Segregation Principle
3. Konsistentes Abstraktionsniveau
4. Methodennamen sinnvoll benennen

→ mehr Details in den Folien der Vorlesung

Welche Fragen habt ihr noch?

Aufgaben (1/2)

- Welche Designprinzipien bricht der folgende Code?

```
public class OnlineShop {
      ...
      public void sell(Customer c,
                       Product p, int amount) {
            if (i instanceof DiscountedProduct) {
                  c.bill(i.price * amount * i.discount);
            } else {
                  c.bill(i.price * amount);
            }
            this.ship(c, i, amount);
      }

      public void add(ArrayList<Product> toAdd) {
            this.items.addAll(toAdd);
      }
      ...
}
```

```
public class Product {
      public String name;
      public double price;
      ...
}

public class DiscountedProduct extends Product
{
      public double discount;
      ...
}
```

Aufgaben (1/2) - Lösung

Open/Closed Principle!

Single Responsibility

```
public class OnlineShop {
      ...
      public void sell(Customer c,
                  Product p, int amount) {
            if (p instanceof DiscountedProduct) {
                  c.bill(p.price * amount * p.discount);
            } else {
                  c.bill(p.price * amount);
            }
            this.ship(c, p, amount);
      }

      public void add(ArrayList<Product> toAdd) {
            this.items.addAll(toAdd);
      }
      ...
}
```

Methodenname ungenau

Schnittstellen anstatt Klassen bevorzugen

+

Internen Aufbau verstecken

Datenkapselung!

```
public class Product {
      public String name;
      public double price;
      ...
}

public class DiscountedProduct extends Product
{
      public double discount;
      ...
}
```

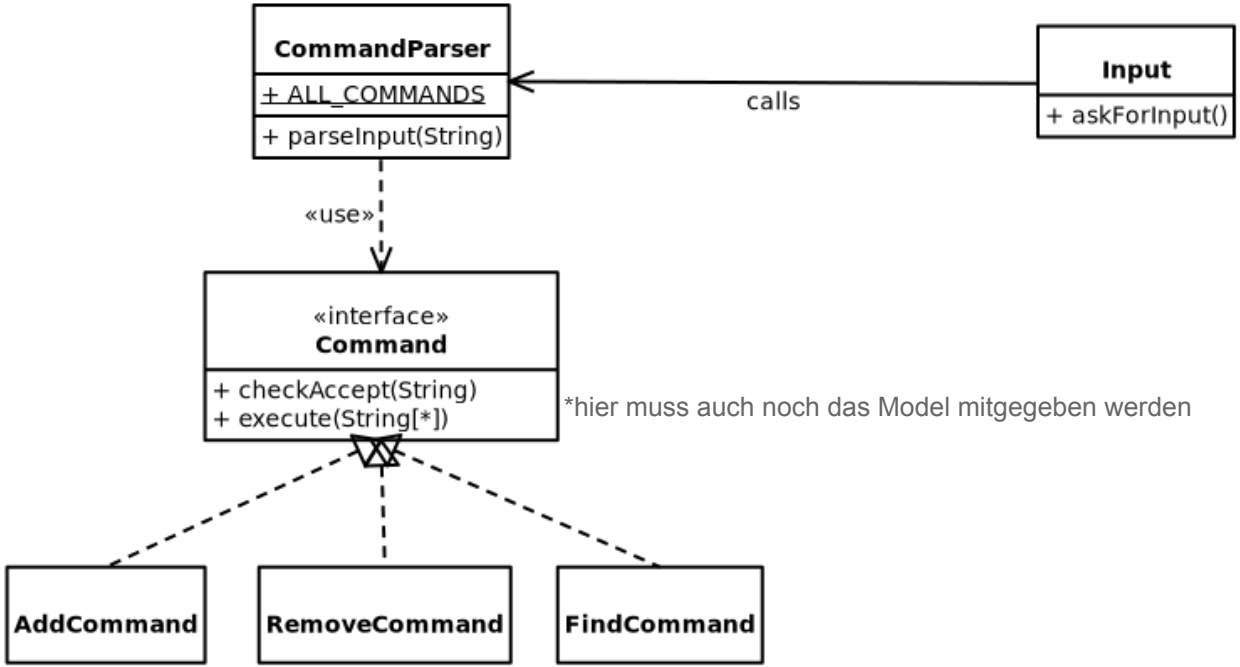
Komposition statt Vererbung bevorzugen

Aufgaben (2/2)

- Aufgabe: Verbessere den folgenden Command-Parser

```
public boolean parseCommand() {
      String input = this.scanner.nextLine();
      String[] args = input.split(" ");

      // Parse input
      if (args[0].equals("add")) {
            this.model.add(args[1]);
            return true;
      } else if (args[0].equals("find")) {
            this.model.find(args[1]);
            return true;
      } else if (args[0].equals("remove")) {
            this.model.remove(args[1]);
            return true;
      }
      return false;
}
```

Aufgaben (2/2) - Lösung(-svorschlag)



Mögliche Erweiterungen:

- CommandResult Klasse für Rückgabewert
- ALL_COMMANDS als HashMap

Schluss

Zusammenfassung

- Rekursion
  - Wiederhole selbes Schema auf Teilmengen, um eine einfache Lösung zu finden
- Java API
  - Ermöglicht viele Basisfunktionalitäten
- OOP-Designprinzipien
  - `private` ist Standard-Modifier
  - Komposition > Vererbung
  - Interfaces sind toll, aber nur wenn sie sinnvoll designed wurden

Nächste Woche voraussichtlich Präsenzübung-Einsicht hier im Tut.

Bis nächste Woche :)