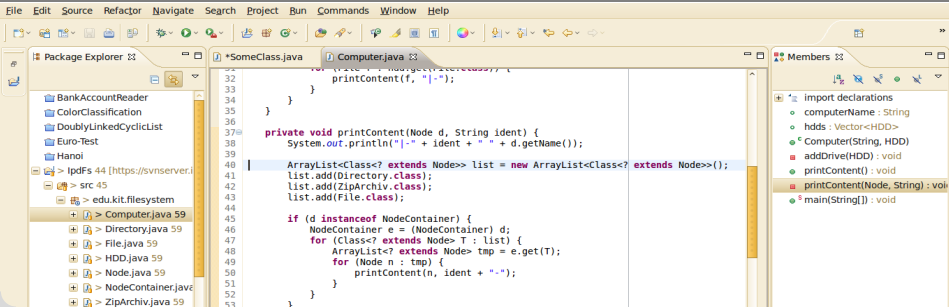


Programmieren-Tutorium Nr. 10 bei Martin Thoma

Überladung, Bindung, List, Dictionarys

Martin Thoma | 12. Februar 2013

FAKULTÄT FÜR INFORMATIK



The screenshot shows an IDE with the following components:

- Package Explorer:** Lists packages like `edu.kit.filesystem` and files like `Computer.java`, `Directory.java`, `File.java`, `HDD.java`, `Node.java`, `NodeContainer.java`, and `ZipArchiv.java`.
- Main Editor:** Displays the code for `Computer.java`. The code includes a `printContent` method and a `main` method. A line of code is highlighted: `ArrayList<Class? extends Node> list = new ArrayList<Class? extends Node>();`.
- Members:** Shows the class members for `Computer`, including `computerName`, `hdds`, `addDrive`, `printContent`, and `main`.

```
32         printContent(f, "|-");
33     }
34 }
35
36
37 private void printContent(Node d, String ident) {
38     System.out.println(f"- " + ident + " " + d.getName());
39 }
40
41 ArrayList<Class? extends Node> list = new ArrayList<Class? extends Node>();
42 list.add(Directory.class);
43 list.add(ZipArchiv.class);
44 list.add(File.class);
45
46 if (d instanceof NodeContainer) {
47     NodeContainer e = (NodeContainer) d;
48     for (Class<? extends Node> T : list) {
49         ArrayList<? extends Node> tmp = e.get(T);
50         for (Node n : tmp) {
51             printContent(n, ident + "-");
52         }
53     }
54 }
```

- 1 Einleitung
- 2 Überladung
- 3 Bindung
- 4 Nachbesprechung
- 5 Java API: List
- 6 Dictionary
- 7 Swing
- 8 Abspann

Main.java

```
1 public class Main {
2     public static void main(String[] args) {
3         CrazyDataStructure a = new ListLike();
4         CrazyDataStructure b = new ArrayListLike();
5     }
6 }

1 public interface CrazyDataStructure {
2     public void add(int a);
3 }

1 public class ListLike implements CrazyDataStructure {
2
3     @Override
4     public void add(int a) {
5
6     }
7 }

1 public class ArrayListLike extends ListLike {
2
3 }
```

- Gibt es einen Compiler-Fehler?
- Gibt es einen Laufzeit-Fehler?
- Alles klappt

Antwort

Alles klappt. Da ArrayListLike von ListLike erbt, erbt die Klasse natürlich auch die Methoden. Insbesondere erbt sie die Methoden, die CrazyDataStructure erwartet. Also implementiert auch ArrayListLike das Interface ListLike, obwohl es nicht explizit dort steht.

Eine Musterlösung zu Blatt 5 ist [hier](#).

Allgemein

- ❶ `void myFunction(int a);` und `void myFunction();`
- ❷ `int myFunction();` und `double myFunction();`
 - Geht in Java nicht
 - Funktioniert aber in Perl und Haskell (Quelle)
- ❸ `void myFunction();` und `int myFunction(int a);`
- ❹ `void myFunction(String a);` und `void myFunction(int a);`
- ❺ `void myFunction(String a);` und `void myFunction(int a, int b);`
- ❻ `void myFunction(String a, int b);` und `void myFunction(int b, String a);`

Allgemein

- ❶ `void myFunction(int a);` und `void myFunction();`
- ❷ `int myFunction();` und `double myFunction();`
 - Geht in Java nicht
 - Funktioniert aber in Perl und Haskell ([Quelle](#))
- ❸ `void myFunction();` und `int myFunction(int a);`
- ❹ `void myFunction(String a);` und `void myFunction(int a);`
- ❺ `void myFunction(String a);` und `void myFunction(int a, int b);`
- ❻ `void myFunction(String a, int b);` und
`void myFunction(int b, String a);`

Allgemein

- ❶ `void myFunction(int a);` und `void myFunction();`
- ❷ `int myFunction();` und `double myFunction();`
 - Geht in Java nicht
 - Funktioniert aber in Perl und Haskell ([Quelle](#))
- ❸ `void myFunction();` und `int myFunction(int a);`
- ❹ `void myFunction(String a);` und `void myFunction(int a);`
- ❺ `void myFunction(String a);` und `void myFunction(int a, int b);`
- ❻ `void myFunction(String a, int b);` und `void myFunction(int b, String a);`

Allgemein

- ❶ `void myFunction(int a);` und `void myFunction();`
- ❷ `int myFunction();` und `double myFunction();`
 - Geht in Java nicht
 - Funktioniert aber in Perl und Haskell ([Quelle](#))
- ❸ `void myFunction();` und `int myFunction(int a);`
- ❹ `void myFunction(String a);` und `void myFunction(int a);`
- ❺ `void myFunction(String a);` und `void myFunction(int a, int b);`
- ❻ `void myFunction(String a, int b);` und
`void myFunction(int b, String a);`

Allgemein

- ❶ `void myFunction(int a);` und `void myFunction();`
- ❷ `int myFunction();` und `double myFunction();`
 - Geht in Java nicht
 - Funktioniert aber in Perl und Haskell ([Quelle](#))
- ❸ `void myFunction();` und `int myFunction(int a);`
- ❹ `void myFunction(String a);` und `void myFunction(int a);`
- ❺ `void myFunction(String a);` und `void myFunction(int a, int b);`
- ❻ `void myFunction(String a, int b);` und
`void myFunction(int b, String a);`

Allgemein

- ❶ `void myFunction(int a);` und `void myFunction();`
- ❷ `int myFunction();` und `double myFunction();`
 - Geht in Java nicht
 - Funktioniert aber in Perl und Haskell ([Quelle](#))
- ❸ `void myFunction();` und `int myFunction(int a);`
- ❹ `void myFunction(String a);` und `void myFunction(int a);`
- ❺ `void myFunction(String a);` und `void myFunction(int a, int b);`
- ❻ `void myFunction(String a, int b);` und `void myFunction(int b, String a);`

Allgemein

- ❶ `void myFunction(int a);` und `void myFunction();`
- ❷ `int myFunction();` und `double myFunction();`
 - Geht in Java nicht
 - Funktioniert aber in Perl und Haskell ([Quelle](#))
- ❸ `void myFunction();` und `int myFunction(int a);`
- ❹ `void myFunction(String a);` und `void myFunction(int a);`
- ❺ `void myFunction(String a);` und `void myFunction(int a, int b);`
- ❻ `void myFunction(String a, int b);` und `void myFunction(int b, String a);`

Allgemein

- ❶ `void myFunction(int a);` und `void myFunction();`
- ❷ `int myFunction();` und `double myFunction();`
 - Geht in Java nicht
 - Funktioniert aber in Perl und Haskell ([Quelle](#))
- ❸ `void myFunction();` und `int myFunction(int a);`
- ❹ `void myFunction(String a);` und `void myFunction(int a);`
- ❺ `void myFunction(String a);` und `void myFunction(int a, int b);`
- ❻ `void myFunction(String a, int b);` und
`void myFunction(int b, String a);`

So was geht nicht:

- `void myFunction(int a, int b);` und
`void myFunction(int b, int a);`

- `void myFunction(String a, int b);` und
`void myFunction(String b, int a);`

⇒ Für den Compiler ist der Bezeichner „a“ bzw. „b“ in der Signatur egal

Dort sind nur der Rückgabewert, der Name der Methode und die Parameter-Typen und deren Reihenfolge wichtig
In Java nicht mal der Rückgabewert

So was geht nicht:

- `void myFunction(int a, int b);` und
`void myFunction(int b, int a);`
- `void myFunction(String a, int b);` und
`void myFunction(String b, int a);`

⇒ Für den Compiler ist der Bezeichner „a“ bzw. „b“ in der Signatur egal

Dort sind nur der Rückgabewert, der Name der Methode und die Parameter-Typen und deren Reihenfolge wichtig
In Java nicht mal der Rückgabewert

So was geht nicht:

- `void myFunction(int a, int b);` und
`void myFunction(int b, int a);`
- `void myFunction(String a, int b);` und
`void myFunction(String b, int a);`

⇒ Für den Compiler ist der Bezeichner „a“ bzw. „b“ in der Signatur egal

Dort sind nur der Rückgabewert, der Name der Methode und die Parameter-Typen und deren Reihenfolge wichtig

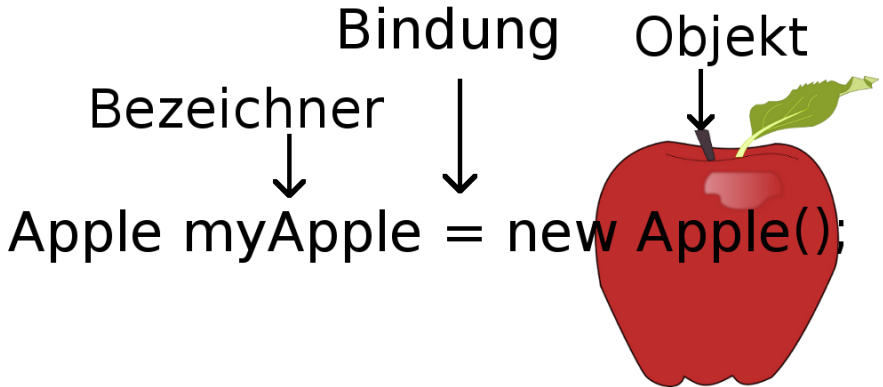
In Java nicht mal der Rückgabewert

Java Language Specification, 8.4.2

Two methods have the same signature if they have the same name and argument types.

Beispiele:

- `Math.signum(double)`, `Math.signum(float)`
- `PrintStream: println();, println(boolean);, println(char); ...`
- `Interface List: Object[] toArray();, <T> T[] toArray(T[] a)`



„Early Binding“ und „Late Binding“

Bindung kann für jeweils ein Objekt zu zwei Zeitpunkten stattfinden:

Early binding, static binding, compile-time-binding

References are resolved at compile time:

```
Animal a = new Animal();  
// The compiler can resolve this method call statically:  
a.Roar();
```

Late binding, dynamic binding, run-time-binding

References are resolved at run time:

```
public void MakeSomeNoise(object a) {  
    // Things happen...  
    // You won't know if this works until runtime:  
    ((Animal) a).Roar();  
}
```

Quelle: [StackOverflow](#)

- **Loose coupling** (Lose Kopplung)
- Loose binding - nicht weit verbreitet, vermutlich falscher Begriff
- **Dynamic dispatch**: Laut Wikipedia das gleiche wie „dynamic binding“.

„Dynamic Binding“ und „Dynamic dispatch“

- Single and multiple dispatch - Java unterstützt nur single dispatch
- Are dynamic dispatch and dynamic binding the same thing? - No.
- Aber: Late binding and dynamic single dispatch are, for all intents and purposes, the same. ([Quelle](#))
- Paper „[Multiple Dispatch in Practice](#)“

- Was ihr schreibt:

```
LinkedList<MyType> myList = new LinkedList<MyType>();
```

- Besser wäre:

```
List<MyType> myList = new LinkedList<MyType>();
```

- Nicht vergessen:

```
import java.util.List;
```

Warum ist Polymorphismus toll?

- siehe [Polymorphism and Interfaces](#)
- siehe 9. Tutorium

```
1 List<Country> europe = new ArrayList<Country>();
2 europe.add(new Country(81903000,357121.41,"Germany"));
3 europe.add(new Country(64667000,668763,"France"));
4 europe.add(new Country( 4985900,385199,"Norway"));
5 europe.add(new Country( 9514406,450295,"Sweden"));
6 europe.add(new Country(47212990,504645,"Spain"));
7 europe.add(new Country( 8014000, 41285,"Switzerland"));
8 europe.add(new Country( 36371, 2.02,"Monaco"));
9 Collections.sort(europe, new Comparator<Country>(){
10     @Override
11     public int compare(Country o1, Country o2) {
12         double o1Density = o1.population / o1.area;
13         double o2Density = o2.population / o2.area;
14
15         if (Math.abs(o1Density - o2Density) < 0.00001) {
16             return 0;
17         } else if (o1Density > o2Density) {
18             return 1;
19         } else {
20             return -1;
21         }
22     }
23 });
24 // Now it's sorted according to the logic in the internal comparator
25 System.out.println(europe);
```

Quelle

- Erst mit Java SE 7 kann man `switch` auf eine Variable vom Typ `String` anwenden
- ...16 Jahre nach dem Feature Request! ([Quelle](#))
- Vorher: `Cannot switch on a value of type String.`
`Only int values or enum constants are permitted`
- Problem: Strings sind Objekte
- Java-Puzzle: `new String("abc") != "abc";`
- Java verwendet `.equals()` ([Quelle](#))
- [Mehr infos](#)

- Erst mit Java SE 7 kann man `switch` auf eine Variable vom Typ `String` anwenden
- ...16 Jahre nach dem Feature Request! ([Quelle](#))
- Vorher: `Cannot switch on a value of type String.`
`Only int values or enum constants are permitted`
- Problem: Strings sind Objekte
- Java-Puzzle: `new String("abc") != "abc";`
- Java verwendet `.equals()` ([Quelle](#))
- [Mehr infos](#)

- Erst mit Java SE 7 kann man `switch` auf eine Variable vom Typ `String` anwenden
- ...16 Jahre nach dem Feature Request! ([Quelle](#))
- Vorher: `Cannot switch on a value of type String.`
`Only int values or enum constants are permitted`
- Problem: Strings sind Objekte
- Java-Puzzle: `new String("abc") != "abc";`
- Java verwendet `.equals()` ([Quelle](#))
- [Mehr infos](#)

- Erst mit Java SE 7 kann man `switch` auf eine Variable vom Typ `String` anwenden
- ...16 Jahre nach dem Feature Request! ([Quelle](#))
- Vorher: `Cannot switch on a value of type String.`
`Only int values or enum constants are permitted`
- Problem: Strings sind Objekte
- Java-Puzzle: `new String("abc") != "abc";`
- Java verwendet `.equals()` ([Quelle](#))
- [Mehr infos](#)

- Erst mit Java SE 7 kann man `switch` auf eine Variable vom Typ `String` anwenden
- ...16 Jahre nach dem Feature Request! ([Quelle](#))
- Vorher: `Cannot switch on a value of type String.`
`Only int values or enum constants are permitted`
- Problem: Strings sind Objekte
- Java-Puzzle: `new String("abc") != "abc";`
- Java verwendet `.equals()` ([Quelle](#))
- Mehr infos

- Erst mit Java SE 7 kann man `switch` auf eine Variable vom Typ `String` anwenden
- ...16 Jahre nach dem Feature Request! ([Quelle](#))
- Vorher: `Cannot switch on a value of type String.`
`Only int values or enum constants are permitted`
- Problem: Strings sind Objekte
- Java-Puzzle: `new String("abc") != "abc";`
- Java verwendet `.equals()` ([Quelle](#))
- [Mehr infos](#)

- Erst mit Java SE 7 kann man `switch` auf eine Variable vom Typ `String` anwenden
- ...16 Jahre nach dem Feature Request! ([Quelle](#))
- Vorher: `Cannot switch on a value of type String.`
`Only int values or enum constants are permitted`
- Problem: Strings sind Objekte
- Java-Puzzle: `new String("abc") != "abc";`
- Java verwendet `.equals()` ([Quelle](#))
- [Mehr infos](#)

Früher hat man einen Enum verwendet:

```
switch (Commands.valueOf(command.toUpperCase())) {  
    case ADD:  
        add();  
        break;  
    case SEARCH:  
        search();  
        break;  
    case QUIT:  
        break;  
    default:  
        System.out.println("Invalid input");  
        break;  
}
```

Person.java

```
1 public class Person {
2     String prename;
3     String surname;
4
5     public Person(String prename, String surname) {
6         super();
7         this.prename = prename;
8         this.surname = surname;
9     }
10
11     @Override
12     public void shout() {
13         System.out.println("AAAAAAAAAAAAARGH!");
14     }
15 }
```

```
1 public class Main {
2     public static void main(String[] args) {
3         Person p = new Person("a", "b");
4         p.shout();
5         System.out.println("kein Problem");
6     }
7 }
```


Compiler-Fehler

Exception in thread "main" java.lang.Error:

Unresolved compilation problem:

The method shout() of type Person must override or implement a supertype method

at Person.shout(Person.java:12)

at Main.main(Main.java:4)

Die Annotation `@Override` ...

- Sollte verwendet werden, damit der Compiler euch warnen kann, wenn ihr nichts überschreibt

⇒ Tippfehler werden unwahrscheinlicher

- Anderen ist klar, dass euch klar war, dass ihr etwas überschreibt
- Code ist leichter verständlich

Die Annotation `@Override` ...

- Sollte verwendet werden, damit der Compiler euch warnen kann, wenn ihr nichts überschreibt
- ⇒ Tippfehler werden unwahrscheinlicher
- Anderen ist klar, dass euch klar war, dass ihr etwas überschreibt
 - Code ist leichter verständlich

Die Annotation `@Override` ...

- Sollte verwendet werden, damit der Compiler euch warnen kann, wenn ihr nichts überschreibt
- ⇒ Tippfehler werden unwahrscheinlicher
- Anderen ist klar, dass euch klar war, dass ihr etwas überschreibt
 - Code ist leichter verständlich

Die Annotation `@Override` ...

- Sollte verwendet werden, damit der Compiler euch warnen kann, wenn ihr nichts überschreibt
- ⇒ Tippfehler werden unwahrscheinlicher
- Anderen ist klar, dass euch klar war, dass ihr etwas überschreibt
 - Code ist leichter verständlich

- Für die Klausur bitte `@author` verwenden
- und euren korrekten Namen angeben
- In Eclipse: `Window` `Preferences` `Java` `Code Style` `Code Templates` `Comments` `Types` und dort `${user}` durch euren Namen ersetzen

Folgender Aufbau ist Konvention und sollte eingehalten werden:

- ① Class/interface documentation comment (`/**...*/`)
- ② class or interface statement
- ③ Class/interface implementation comment (`/*...*/`), if necessary
- ④ Class (static) variables
- ⑤ **Instance variables**
- ⑥ **Constructors**
- ⑦ **Methods**

Quelle: [Oracle Coding Conventions](#), 3.1.3

Es gibt in `java.util.*`

- **Interface `List<E>`**

- `add(E e)`, `contains(Object o)`, `get(int index)`,
`remove(Object o)`

- **Class `ArrayList<E>`**

- Stärken: `get`, `add` $\in \mathcal{O}(1)$ - `add` nur amortisiert!
- Schwächen: `remove` $\in \mathcal{O}(n)$

- **Class `LinkedList<E>`**

- Stärken: `add` $\in \mathcal{O}(1)$
- Schwächen: `get`, `remove` $\in \mathcal{O}(n)$, Speicherplatz

⇒ `LinkedList` nur dann, wenn ihr `add` sehr häufig nutzt

Siehe: **`When to use LinkedList<> over ArrayList<>?`**

Es gibt in `java.util.*`

- **Interface `List<E>`**

- `add(E e)`, `contains(Object o)`, `get(int index)`,
`remove(Object o)`

- **Class `ArrayList<E>`**

- Stärken: `get`, `add` $\in \mathcal{O}(1)$ - `add` nur amortisiert!
- Schwächen: `remove` $\in \mathcal{O}(n)$

- **Class `LinkedList<E>`**

- Stärken: `add` $\in \mathcal{O}(1)$
- Schwächen: `get`, `remove` $\in \mathcal{O}(n)$, Speicherplatz

⇒ `LinkedList` nur dann, wenn ihr `add` sehr häufig nutzt

Siehe: [When to use `LinkedList<>` over `ArrayList<>`?](#)

Es gibt in `java.util.*`

- **Interface `List<E>`**

- `add(E e)`, `contains(Object o)`, `get(int index)`,
`remove(Object o)`

- **Class `ArrayList<E>`**

- Stärken: `get`, `add` $\in \mathcal{O}(1)$ - `add` nur amortisiert!
- Schwächen: `remove` $\in \mathcal{O}(n)$

- **Class `LinkedList<E>`**

- Stärken: `add` $\in \mathcal{O}(1)$
- Schwächen: `get`, `remove` $\in \mathcal{O}(n)$, Speicherplatz

⇒ `LinkedList` nur dann, wenn ihr `add` sehr häufig nutzt

Siehe: [When to use `LinkedList<>` over `ArrayList<>`?](#)

Es gibt in `java.util.*`

- **Interface `List<E>`**

- `add(E e)`, `contains(Object o)`, `get(int index)`,
`remove(Object o)`

- **Class `ArrayList<E>`**

- Stärken: `get`, `add` $\in \mathcal{O}(1)$ - `add` nur amortisiert!
- Schwächen: `remove` $\in \mathcal{O}(n)$

- **Class `LinkedList<E>`**

- Stärken: `add` $\in \mathcal{O}(1)$
- Schwächen: `get`, `remove` $\in \mathcal{O}(n)$, Speicherplatz

⇒ `LinkedList` nur dann, wenn ihr `add` sehr häufig nutzt

Siehe: [When to use `LinkedList<>` over `ArrayList<>`?](#)

Es gibt in `java.util.*`

- **Interface `List<E>`**

- `add(E e)`, `contains(Object o)`, `get(int index)`,
`remove(Object o)`

- **Class `ArrayList<E>`**

- Stärken: `get`, `add` $\in \mathcal{O}(1)$ - `add` nur amortisiert!
- Schwächen: `remove` $\in \mathcal{O}(n)$

- **Class `LinkedList<E>`**

- Stärken: `add` $\in \mathcal{O}(1)$
- Schwächen: `get`, `remove` $\in \mathcal{O}(n)$, Speicherplatz

⇒ `LinkedList` nur dann, wenn ihr `add` sehr häufig nutzt

Siehe: [When to use `LinkedList<>` over `ArrayList<>`?](#)

Es gibt in `java.util.*`

- **Interface `List<E>`**

- `add(E e)`, `contains(Object o)`, `get(int index)`,
`remove(Object o)`

- **Class `ArrayList<E>`**

- Stärken: `get`, `add` $\in \mathcal{O}(1)$ - `add` nur amortisiert!
- Schwächen: `remove` $\in \mathcal{O}(n)$

- **Class `LinkedList<E>`**

- Stärken: `add` $\in \mathcal{O}(1)$
- Schwächen: `get`, `remove` $\in \mathcal{O}(n)$, Speicherplatz

⇒ `LinkedList` nur dann, wenn ihr `add` sehr häufig nutzt

Siehe: [When to use `LinkedList<>` over `ArrayList<>`?](#)

Es gibt in `java.util.*`

- **Interface `List<E>`**

- `add(E e)`, `contains(Object o)`, `get(int index)`,
`remove(Object o)`

- **Class `ArrayList<E>`**

- Stärken: `get`, `add` $\in \mathcal{O}(1)$ - `add` nur amortisiert!
- Schwächen: `remove` $\in \mathcal{O}(n)$

- **Class `LinkedList<E>`**

- Stärken: `add` $\in \mathcal{O}(1)$
- Schwächen: `get`, `remove` $\in \mathcal{O}(n)$, Speicherplatz

⇒ `LinkedList` nur dann, wenn ihr `add` sehr häufig nutzt

Siehe: [When to use `LinkedList<>` over `ArrayList<>`?](#)

Es gibt in `java.util.*`

- **Interface `List<E>`**

- `add(E e)`, `contains(Object o)`, `get(int index)`,
`remove(Object o)`

- **Class `ArrayList<E>`**

- Stärken: `get`, `add` $\in \mathcal{O}(1)$ - `add` nur amortisiert!
- Schwächen: `remove` $\in \mathcal{O}(n)$

- **Class `LinkedList<E>`**

- Stärken: `add` $\in \mathcal{O}(1)$
- Schwächen: `get`, `remove` $\in \mathcal{O}(n)$, Speicherplatz

⇒ `LinkedList` nur dann, wenn ihr `add` sehr häufig nutzt

Siehe: [When to use `LinkedList<>` over `ArrayList<>`?](#)

Es gibt in `java.util.*`

- **Interface `List<E>`**

- `add(E e)`, `contains(Object o)`, `get(int index)`,
`remove(Object o)`

- **Class `ArrayList<E>`**

- Stärken: `get`, `add` $\in \mathcal{O}(1)$ - `add` nur amortisiert!
- Schwächen: `remove` $\in \mathcal{O}(n)$

- **Class `LinkedList<E>`**

- Stärken: `add` $\in \mathcal{O}(1)$
- Schwächen: `get`, `remove` $\in \mathcal{O}(n)$, Speicherplatz

⇒ `LinkedList` nur dann, wenn ihr `add` sehr häufig nutzt


Siehe: [When to use `LinkedList<>` over `ArrayList<>`?](#)


List: Beispiel


```
1 import java.util.ArrayList;
2 import java.util.Collections;
3 import java.util.LinkedList;
4 import java.util.List; // nicht java.awt.List!
5
6 public class Main {
7     public static void main(String[] args) {
8         List<Integer> myList = new ArrayList<Integer>();
9         List<Integer> anotherList = new LinkedList<Integer>();
10
11         for (int i = 0; i < 5; i += 1) {
12             myList.add((int) (Math.random() * 100));
13             anotherList.add((int) (Math.random() * 100));
14         }
15
16         System.out.println("myList: " + myList);
17         System.out.println("anotherList: " + anotherList);
18
19         myList.addAll(anotherList);
20         Collections.sort(myList);
21         System.out.println("combined: " + myList);
22
23         for (int myInt : myList) {
24             // do something with myInt...
25         }
26     }
27 }
```


- myList: [41, 35, 9, 51, 35]
- anotherList: [2, 51, 64, 58, 57]
- combined: [2, 9, 35, 35, 41, 51, 51, 57, 58, 64]


- `ArrayList` und `Vector` sind praktisch identisch
 - Einziger Unterschied: `Vector` ist synchronized
- ⇒ In Single-Thread-Anwendungen immer `ArrayList` verwenden
- ⇒ Für die Abschlusssaufgabe auf keinen Fall `Vektor` verwenden
- `Vector` war bei mir mehr als $2\times$ so langsam wie `ArrayList` (→ [Performance of Matrix multiplication in Python, Java and C++](#))

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom 
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $O(1)$ für get und put
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $O(\log n)$ für containsKey, get, put und remove
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $O(1)$ für add und remove

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom 
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $O(1)$ für get und put
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $O(\log n)$ für containsKey, get, put und remove
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $O(1)$ für add und remove

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom 
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $O(1)$ für get und put
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $O(\log n)$ für containsKey, get, put und remove
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $O(1)$ für add und remove

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom 
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $O(1)$ für get und put
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $O(\log n)$ für containsKey, get, put und remove
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $O(1)$ für add und remove

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom 
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $O(1)$ für get und put
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $O(\log n)$ für containsKey, get, put und remove
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $O(1)$ für add und remove

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom **=**
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $O(1)$ für get und put
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $O(\log n)$ für containsKey, get, put und remove
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $O(1)$ für add und remove

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom `=`
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $O(1)$ für `get` und `put`
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $O(\log n)$ für `containsKey`, `get`, `put` und `remove`
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $O(1)$ für `add` und `remove`

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom `=`
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $O(1)$ für `get` und `put`
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $O(\log n)$ für `containsKey`, `get`, `put` und `remove`
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $O(1)$ für `add` und `remove`

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom `=`
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $O(1)$ für `get` und `put`
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $O(\log n)$ für `containsKey`, `get`, `put` und `remove`
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $O(1)$ für `add` und `remove`

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom `=`
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $\mathcal{O}(1)$ für `get` und `put`
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $\mathcal{O}(\log n)$ für `containsKey`, `get`, `put` und `remove`
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $\mathcal{O}(1)$ für `add` und `remove`

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom `=`
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $\mathcal{O}(1)$ für `get` und `put`
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $\mathcal{O}(\log n)$ für `containsKey`, `get`, `put` und `remove`
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $\mathcal{O}(1)$ für `add` und `remove`

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom `=`
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $\mathcal{O}(1)$ für `get` und `put`
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $\mathcal{O}(\log n)$ für `containsKey`, `get`, `put` und `remove`
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $\mathcal{O}(1)$ für `add` und `remove`

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom `=`
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $\mathcal{O}(1)$ für `get` und `put`
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $\mathcal{O}(\log n)$ für `containsKey`, `get`, `put` und `remove`
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $\mathcal{O}(1)$ für `add` und `remove`

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom `=`
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $\mathcal{O}(1)$ für `get` und `put`
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $\mathcal{O}(\log n)$ für `containsKey`, `get`, `put` und `remove`
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $\mathcal{O}(1)$ für `add` und `remove`

- **Dictionary** is obsolete. New implementations should implement the **Map** interface, rather than extending Dictionary.
- Telefonbuch-Anwendungsfälle
 - Schlüssel wird auf Wert abgebildet
 - Beides sind Objekte
 - Schlüssel ist eindeutig, Wert darfs mehrfach geben (sowohl identisch als auch gleich sind ok)
- Interface Map links vom `=`
- **HashMap**:
 - makes no guarantees as to the order of the map
 - $\mathcal{O}(1)$ für `get` und `put`
- **TreeMap**:
 - sorted according to the natural ordering of its keys
 - $\mathcal{O}(\log n)$ für `containsKey`, `get`, `put` und `remove`
- **LinkedHashMap**:
 - predictable iteration order (usually insertion-order)
 - $\mathcal{O}(1)$ für `add` und `remove`

```
Map<Integer, String> d = new HashMap<Integer, String> ();  
d.put(12, "Martin");  
d.put(9, "Marie");  
d.put(18, "Peter");
```

```
String name = d.get(9);  
// name.equals("Marie")!
```

<http://stackoverflow.com/questions/2889777/difference-between-hashmap-linkedhashmap-and-sortedmap-in-java>

Beispiel 2.1

Person.java

```
public class Person {
    String prename;
    String surname;

    public Person(String prename, String surname) {
        super();
        this.prename = prename;
        this.surname = surname;
    }

    @Override
    public String toString() {
        return prename + " " + surname;
    }
}
```

TelephoneNumber.java

```
public class TelephoneNumber {
    String number;

    public TelephoneNumber(String number) {
        super();
        this.number = number;
    }

    @Override
    public String toString() {
        return number;
    }
}
```

Beispiel 2.2

```
1 import java.util.HashMap;
2 import java.util.Map;
3
4 public class Main {
5     public static void main(String[] args) {
6         Map<Person, TelephoneNumber> phonebook =
7             new HashMap<Person, TelephoneNumber>();
8
9         TelephoneNumber a = new TelephoneNumber("01636280491");
10        phonebook.put(new Person("Martin", "Thoma"), a);
11        phonebook.put(new Person("Max", "Mustermann"), a);
12        System.out.println(phonebook);
13    }
14 }
```

Beispiel 2.3: Durch Map iterieren

```
for (Map.Entry<Person, TelephoneNumber>  
    entry : phonebook.entrySet()) {  
    Person k = entry.getKey();  
    TelephoneNumber v = entry.getValue();  
    System.out.println(k + " " + v);  
}
```

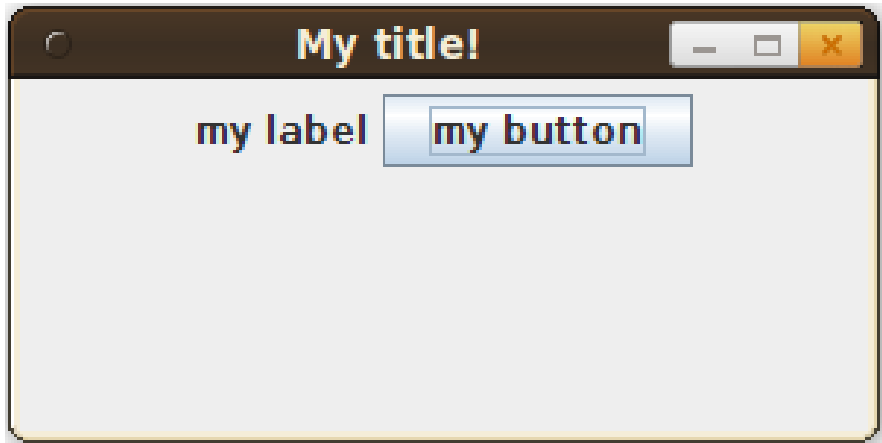
```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Main {
    public static void main(String[] args) {
        JFrame frame = new JFrame("My title!");
        frame.setVisible(true);
        frame.setSize(300, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel = new JPanel();
        frame.add(panel);

        JLabel label = new JLabel("my label");
        panel.add(label);

        JButton button = new JButton("my button");
        panel.add(button);
    }
}
```



ActionListener sind ...

- ein weit verbreitetes Konzept
- Objekte, die auf bestimmte Aktionen warten und
 - dann was machen
 - die Aktion „delegieren“
- das **Interface ActionListener**

ActionListener sind ...

- ein weit verbreitetes Konzept
- Objekte, die auf bestimmte Aktionen warten und
 - dann was machen
 - die Aktion „delegieren“
- das [Interface ActionListener](#)

ActionListener sind ...

- ein weit verbreitetes Konzept
- Objekte, die auf bestimmte Aktionen warten und
 - dann was machen
 - die Aktion „delegieren“
- das [Interface ActionListener](#)

ActionListener sind ...

- ein weit verbreitetes Konzept
- Objekte, die auf bestimmte Aktionen warten und
 - dann was machen
 - die Aktion „delegieren“
- das [Interface ActionListener](#)

ActionListener sind ...

- ein weit verbreitetes Konzept
- Objekte, die auf bestimmte Aktionen warten und
 - dann was machen
 - die Aktion „delegieren“
- das [Interface ActionListener](#)

```
public class test {  
    public static void main(String[] args) {  
        JFrame frame = new JFrame("My title!");  
        frame.setVisible(true);  
        frame.setSize(300, 150);  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        JPanel panel = new JPanel();  
        frame.add(panel);  
  
        JLabel label = new JLabel("my label");  
        panel.add(label);  
  
        JButton button = new JButton("my button");  
        panel.add(button);  
        button.addActionListener(new MyAction());  
    }  
  
    static class MyAction implements ActionListener {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            JFrame frame2 = new JFrame("clicked");  
            frame2.setVisible(true);  
            frame2.setSize(200,200);  
        }  
    }  
}
```

[complete source](#)

Kommende Tutorien: Themen?

- Snake
- Space Invaders
- Breakout
- Tetris
- Minesweeper
- Sokoban
- Swing (Fenster in Java)
- Port-Scanner
- Multiprocessing (Matrix-Multiplikation auf mehreren Kernen)
- Web-Crawler
- PageRank auf reale Daten anwenden
- Chat

⇒ Bitte auf [Doodle](#) wählen!

Bei wem steht noch ...

- Einverständniserklärung nicht im Praktomaten registriert
- Prüfungsanmeldung nicht im Praktomaten registriert

- Was der Prof. über den Verteilger geschrieben hat war Unfug.
- Zulassungsbescheinigung über Studienbüro (blauer Schein)
- Vermutlich kein Übungsschein
- Nur für die Klausur

2. 21.01.2013

1. 28.01.2013: Abschlussprüfunsvorbereitung

- 28.01.2013: Ausgabetermin für Teil 1

0. 04.02.2013: Abschlussprüfunsvorbereitung

- 10.02.2013: Ende der Vorlesungszeit des WS 2012/2013 ([Quelle](#))

Vielen Dank für eure Aufmerksamkeit!

