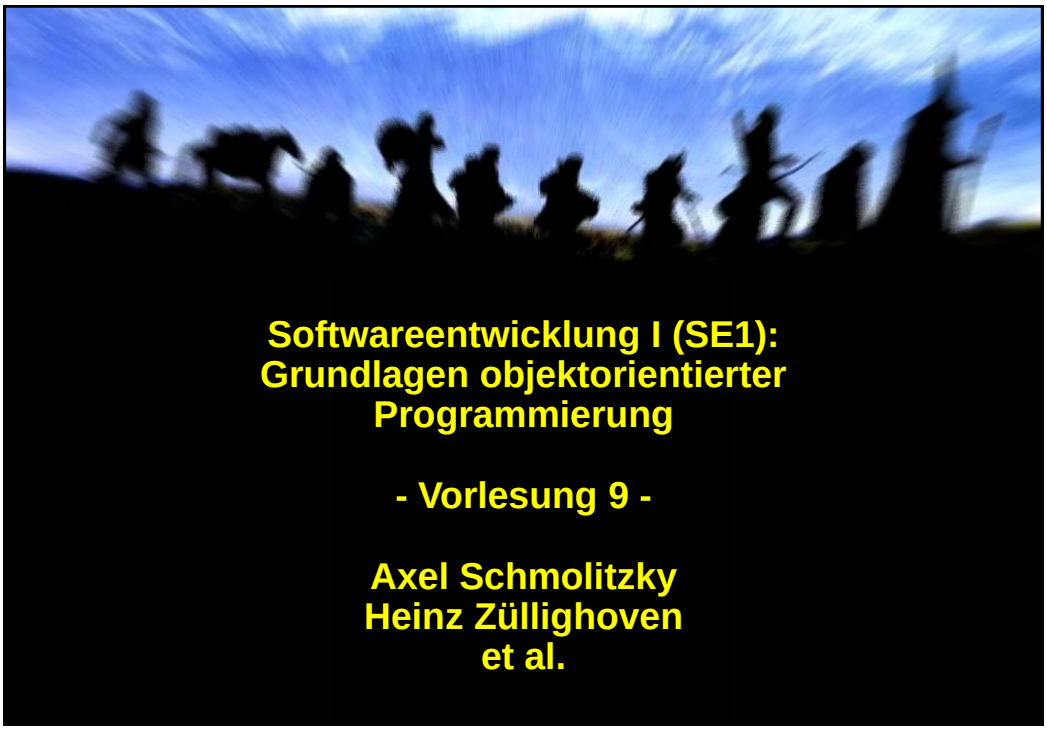


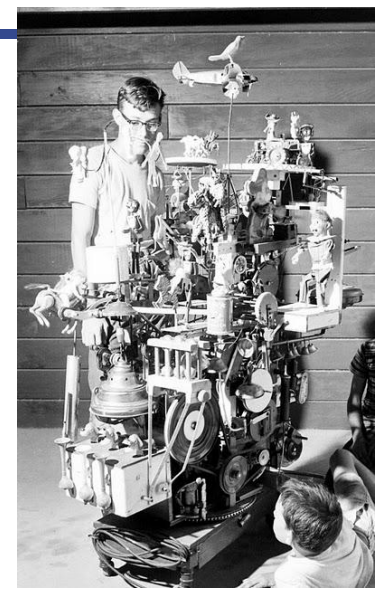
Softwareentwicklung I (SE1): Grundlagen objektorientierter Programmierung

- Vorlesung 9 -

Axel Schmolitzky
Heinz Züllighoven
et al.

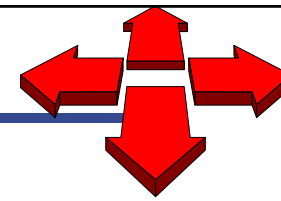
Worte vorweg

- Übersetzbare Programme sind eventuell unsinnig.
- Quelltexte sind wie **Baupläne** für **Maschinen**:
  - Aus einem Bauplan (Quelltext) kann eine Maschine entstehen (lauffähiges System).
  - Der Bauplan kann korrekt sein (mechanisch umsetzbar), aber die resultierende Maschine kann komplett unsinnig sein!
  - Damit wir wissen, was unsere Maschinen tun, müssen wir sowohl die Mechanik ihrer Einzelteile gut kennen als auch das Zusammenspiel dieser Teile.



http://blog.stuttgarter-zeitung.de/wp-content/crazy-machine-1961.jpg

Objektsammlungen

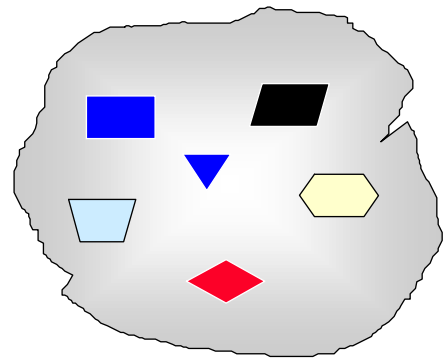


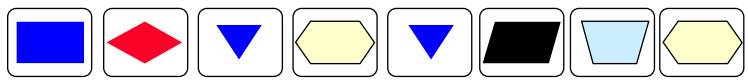
- Bei praktisch jeder größeren Programmieraufgabe werden **gleichartige Objekte zusammengefasst** oder gebündelt und solche **Zusammenfassungen als eigene Objekte** angesehen.
- Solche **Objektsammlungen** werden so häufig benötigt, dass praktisch für jede Programmiersprache vordefinierte Sammlungsbausteine zur Verfügung gestellt werden (entweder in der Sprache oder in ihren Bibliotheken).
- Wir nähern uns Objektsammlungen zunächst über ihre **Benutzung** aus Klientensicht. Erst wenn der **Umgang mit Sammlungen** diskutiert wurde, werden wir uns ihre interne Realisierung ansehen.

SE1 – Level 3 3

Mengen und Listen

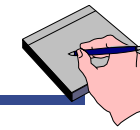
- **Listen** und **Mengen** sind Sammlungen, die in der theoretischen Informatik und in der Softwaretechnik oft verwendet werden.
- **Listen** sind **lineare** Sammlungen von gleichartigen Elementen (Werten), in denen **ein Element mehrfach** auftreten kann.
- **Mengen** sind **ungeordnete** Sammlungen von Elementen (Werten), in denen **jedes Element nur einmal** vorkommt.





SE1 – Level 3 4

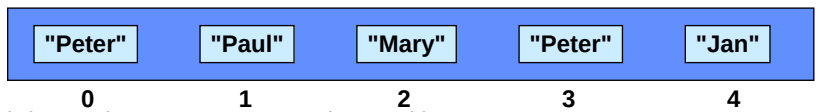
Liste, theoretisch



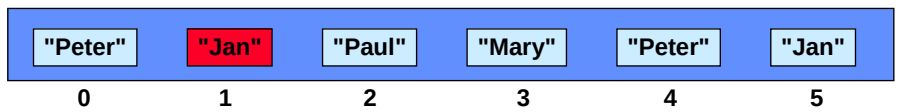
- Listen sind theoretisch betrachtet Aneinanderreihungen von **gleichartigen** Werten zu **Folgen**:
  - die **Reihenfolge** der Listenelemente ist von Bedeutung,
  - ein **Wert** kann in einer Liste **mehrfach** vorkommen.
- Für die theoretische Betrachtung ist es üblich, eine an der Mathematik orientierte **rekursive Definition** einer Liste zu verwenden:
  - Eine Liste ist entweder eine leere Liste (oft notiert als `[]`)
  - oder ein Listenelement gefolgt von einer Liste.
- Listen werden oft sequentiell (d.h. elementweise) durchlaufen, dabei wird eine Operation auf jedes Element der Liste angewendet, und die Reihenfolge wird beachtet.

Umgang mit einer Liste

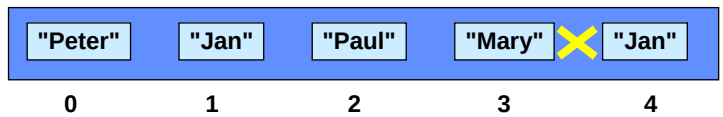
- Beispiel:
  Eine Liste von Strings, etwa Namen für die Einteilung der Pausenaufsicht in einer Schule.



- Einfügen eines Namens an zweiter Position:



- Entfernen des zweiten Eintrags für "Peter":



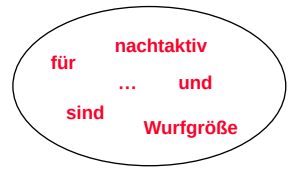
Menge, theoretisch

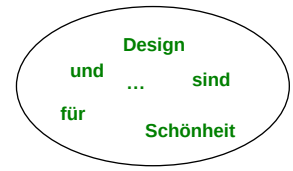
- **Menge** (engl. set): Eine Sammlung von gleichartigen Elementen, wobei jedes Element nur einmal vorkommt.
- Kann ein Element mehrfach vorkommen, spricht man von **Mehrfach- oder Multimengen** (engl. bag).
- Es gelten die bekannten mathematischen Mengenoperationen. Üblich sind:
  - ***insert***: Füge ein Element zur Menge hinzu
  - ***delete***: Entferne ein Element aus der Menge
  - ***element***: Prüfe, ob ein Element in der Menge vorhanden ist
  - ***union***: Vereinige zwei Mengen zu einer neuen.
  - ***intersection***: Bestimme die Schnittmenge zweier Mengen.
  - ***difference***: Bestimme die Differenzmenge zweier Mengen.
  - ***empty***: Prüfe, ob eine Menge leer ist.

SE1 – Level 3 7

Umgang mit einer Menge

- Beispiel **Textanalyse**: Einen längeren Text können wir als eine **Liste von Wörtern** ansehen.
- Wir können ihn aber auch als **Menge von Wörtern** betrachten, wenn uns primär interessiert, **welche Wörter** verwendet werden.
- Wenn wir für mehrere Texte solche Mengen bilden, dann können einige interessante Fragen beantwortet werden:
  - Welche Wörter sind sowohl in **Text 1** als auch in **Text 2** enthalten? Bei inhaltlich verschiedenen Texten werden das eher **Füllwörter** sein.
  - Welche Wörter bleiben übrig, wenn wir diese Füllwörter von den Wörtern eines der Texte abziehen?





http://www.spiegel.de

SE1 – Level 3 8

Maps: Abbildungen von Schlüsseln auf Werte

- Eine **Abbildung** (engl.: **Map**) ist eine **Menge von Schlüssel-Wert-Paaren**:

  Schlüssel 1 → *Wert 1*
  Schlüssel 2 → *Wert 2*
  Schlüssel 3 → *Wert 1*
  …
  Schlüssel n → *Wert m*

  In unserem Beispiel:
  Eine Abbildung von Wörtern auf ihre Häufigkeiten:
  (**String → Integer**)
  Eine Abbildung von Häufigkeiten auf Mengen der Wörter mit der jeweiligen Häufigkeit:
  (**Integer → Set of String**)

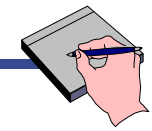
  - Der **Schlüssel** ist (wie die Elemente in einer Menge) **eindeutig**.
  - Der **Wert** kann **beliebig** (muss nicht eindeutig) sein.

- **Umgang**: Über die Angabe eines Schlüssels bekommt ein Klient den zugeordneten Wert geliefert.
- Beispiel Telefonbuch: Name → Telefonnummer (wenn die Menge der Namen eindeutig ist)

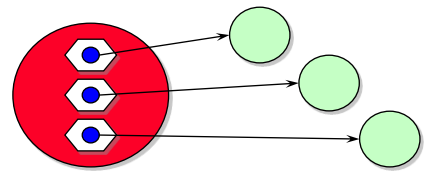
Listen und Mengen, objektorientiert

- Die theoretische Informatik beschreibt Sammlungen mit mathematischen (oft funktionalen) Konzepten.
- Objektorientierte Sammlungen werden eher **zustandsbasiert** betrachtet:
  - So werden z.B. Mengen und Listen als eigenständige Objekte unabhängig von ihren Elementen betrachtet.
  - Eine Menge ist dabei wie ein „ungeordneter Behälter“ für seine Elemente, die eingefügt und herausgenommen werden können.
  - Eine Liste ordnet ihre Elemente in Positionen an. Diese Ordnung kann vordefiniert oder vom Benutzer beeinflusst werden.

Der Begriff „Sammlung“

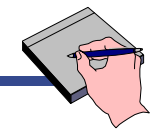


- Eine **Sammlung** von Objekten wird im Java-Umfeld auch unter dem englischen Begriff **Collection** gefasst.
  - Eine Sammlung ist ein Objekt, das eine Gruppe von anderen Objekten zusammenfasst.
  - Sammlungen werden verwendet, um andere Objekte zu speichern, gemeinsam zu manipulieren und Mengen von Objekten an eine andere weiter zu geben.
  - Sammlungen enthalten in der Regel Objekte **vom selben Typ**: eine Menge von Briefen, eine Menge von Konten.
  - Alternativ gebräuchliche Begriffe für Sammlung sind **Behälter** und **Container**.



SE1 – Level 3 11

Wichtige Begriffe zu Sammlungen



- Die Objekte, die in einer Sammlung gehalten werden, werden als die **Elemente** der Sammlung bezeichnet. Jede Sammlung bietet Operationen zum Einfügen und Entfernen von Elementen.
- Der Typ der enthaltenen Elemente (der **Elementtyp**) wird üblicherweise auch als eine Eigenschaft der Sammlung angesehen (Beispiel: eine Sammlung von Strings).
- Die Anzahl der enthaltenen Elemente bezeichnen wir als ihre **Kardinalität**. Bei Sammlungen ist diese üblicherweise nicht nach oben begrenzt: Sammlungen können **beliebig viele Elemente** enthalten.
- Wenn ein Element, das in eine Sammlung eingefügt wird, bereits in der Sammlung enthalten ist, nennt man das einzufügende Element ein **Duplikat**.

SE1 – Level 3 12

Eigenschaften von Sammlungen in Software

- Auch für Sammlungen gilt: Es muss eine deutliche Unterscheidung zwischen ihrer **Schnittstelle** (ihrem Umgang) und ihrer **Implementation** vorgenommen werden.
- Eigenschaften einer **Sammlungs-Schnittstelle**:
  - Umgang mit Duplikaten (erlaubt oder nicht?)
  - Handhabung einer Reihenfolge
- Eigenschaften einer **Sammlungs-Implementation**:
  - verwendete Datenstrukturen (Array, Verkettung, Kombination)
  - Effizienz (wie schnell sind einzelne Operationen in ihrer Ausführung?)

Ein Klient einer Sammlung ist **in erster Linie** an ihrer Schnittstelle interessiert; wie diese realisiert ist, ist hingegen meist nur zweitrangig.

Der Umgang mit Sammlungen

- Für den Umgang mit einer Sammlung ist wichtig, ob und wie eine **Reihenfolge** der Elemente gehandhabt wird. Dafür gibt es verschiedene Möglichkeiten:
  - es ist **keine** Reihenfolge definiert
  - die Reihenfolge ist **benutzerdefiniert** festgelegt
  - die Reihenfolge wird **automatisch** durch die Sammlung erstellt
- Außerdem ist wichtig, wie mit **Duplikaten** umgegangen wird. Wir unterscheiden zwei Möglichkeiten:
  - Duplikate sind **zugelassen** und erhöhen die Kardinalität.
  - Duplikate sind **nicht zugelassen**, das Duplikat wird nicht eingefügt.

Dimensionen möglicher Umgangsformen

	Reihenfolge irrelevant	Reihenfolge benutzerdefiniert	Reihenfolge automatisch
Duplikate zugelassen	Multimenge (Bag) Scrabble-Beutel, Münzen in Geldbörse	Liste (List) Pausenaufsicht, Scrabble-Wort	sortierte Liste (Sorted List) Messwerte
Duplikate nicht zugelassen	Menge (Set) Babynamen	geordnete Menge (Ordered Set) Wunschzettel, Bilder in Galerie	sortierte Menge (Sorted Set) Bundesligatabelle

Sammlungen in Java

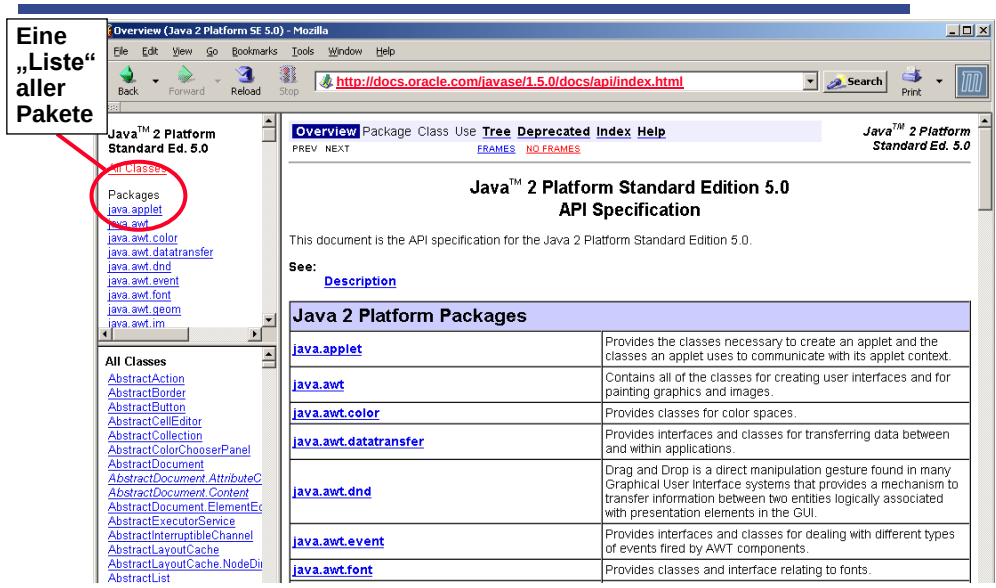
- In den Bibliotheken von Java gibt es eine umfangreiche Unterstützung für Sammlungen: das **Java Collections Framework (JCF)**.
- Dieses Framework besteht aus einer Reihe von **Interfaces** und einer Reihe von **Klassen**, die diese Interfaces implementieren.
- Dieses Framework hat sich über die Jahre so weit etabliert, dass es wie ein Teil der Sprache betrachtet werden kann.
- Im folgenden betrachten wir exemplarisch die beiden Interfaces `List` und `Set`, mit denen Listen und Mengen modelliert werden.

Die Standard-Bibliothek von Java: Das Java API

- Über die Sprache hinaus gehören zu jeder Java-Installation eine Reihe von Klassen und Interfaces. Diese liegen in einer Bibliothek, die als **Java Application Programmer Interface**, kurz **Java API**, bezeichnet wird.
- Diese Bibliothek ist zergliedert in kleinere Einheiten, in so genannte **Pakete** (engl.: packages).
- Das wichtigste Paket ist **`java.lang`**. Es enthält alle Klassen und Interfaces, die als Teil der Sprache angesehen werden. Dazu gehört beispielsweise die Klasse **`String`**, die wir deshalb ohne weiteres benutzen können.
- Klassen und Interfaces aus allen anderen Paketen müssen **importiert** werden, um direkt benutzbar zu sein. Das Java Collections Framework beispielsweise liegt im Paket **`java.util`**.
- Die entsprechenden **Import-Anweisungen** stehen immer zu Anfang einer Java-Übersetzungseinheit:

```
import java.util.Set;
/**
 * Klassenkommentar
 * ...
```


Das Java-API im javadoc-Format



Das Interface `Set` (relevanter Ausschnitt)

Zahl der Elemente in der Collection

Suche von Elementen

Einfügen und Entfernen

```
public interface Set<E> extends Collection<E>
{
  // Sondierende Operationen
  int size();
  boolean isEmpty();
  boolean contains(Object o);

  // Verändernde Operationen
  boolean add(E o);
  boolean remove(Object o);
  void clear();
}
```

Dieses Interface (wie viele andere im JCF auch) ist **generisch** definiert. Im Kopf des Interfaces ist dabei in spitzen Klammern ein Platzhalter für den **Elementtyp** angegeben, hier **`<E>`**. **`E`** kann dann in den Signaturen der Operationen als Typ verwendet werden, hier etwa bei **`add`**.

SE1 – Level 3 19

Beispiel einer Set-Benutzung in Java

```
// Brainstorming: Wir sammeln Babynamen…
// Wir erzeugen ein Exemplar einer Klasse, die das Interface Set implementiert
Set<String> babynamen = new HashSet<String>();
int anzahl = babynamen.size();   // 0 (anfangs ist die Menge leer)
babynamen.add("Klara");
babynamen.add("Anna");
babynamen.add("Annika");
babynamen.add("Anna");
// Wie viele haben wir schon?
anzahl = babynamen.size();  // 3 ("Anna" war beim zweiten Mal ein Duplikat)
if (!babynamen.contains("Julia"))
{
    babynamen.add("Julia");
}
// "Annika" ist nicht gut…
babynamen.remove("Annika");
anzahl = babynamen.size();  // 3
// Eigentlich alles nicht gut, nochmal von vorn…
babynamen.clear();
```

Anmerkung: Dieses Beispiel illustriert den Umgang mit einem Set, wird aber hoffentlich niemals Eingang in seriösen Quelltext bekommen… ☺

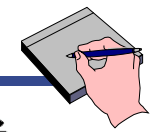
SE1 – Level 3 20

Das Interface `Set`

- Ein Set definierte keine Ordnung der Elemente.
- Die Semantik von `add` ist so definiert, dass **keine Duplikate** eingefügt werden können.
- Gleichheit von Elementen wird mit der Operation `equals` geprüft.
- Formal gilt für alle `e1 != e2` im Set: `!e1.equals(e2)`
- So genannte Massenoperationen (engl.: bulk operations) haben bei Sets die Bedeutung von mathematischen **Mengenoperationen**.

`s1.containsAll(s2):`	s2 Untermenge von s1 ?
`s1.addAll(s2):`	s1 = s1 vereinigt mit s2
`s1.removeAll(s2):`	s1 = s1 – s2
`s1.retainAll(s2):`	s1 = s1 geschnitten mit s2

Die Operation `equals`



- Jede Klasse in Java bietet automatisch alle Operationen an, die in der Klasse `java.lang.Object` definiert sind.
- Unter anderem definiert jeder Referenztyp deshalb die Operation `equals`:

```
public boolean equals(Object other)
```

- **Jedes Objekt** kann über diese Operation gefragt werden, ob es gleich ist mit dem als Parameter angegebenen Objekt. Als Parameter kann eine **beliebige Referenz** übergeben werden.
- Die Standardimplementation vergleicht die Referenz des gerufenen Objektes mit der übergebenen Referenz. Also:

  <Test auf Gleichheit> standardmäßig ***<Vergleich der Identität>***

Picasso: Mädchen vor dem Spiegel

Redefinieren von `equals`

- Für bestimmte Klassen ist es sinnvoll, dass sie eine eigene Definition von Gleichheit festlegen.
- Diese Klassen können die vorgegebene Implementation ändern, indem sie eine alternative Implementation angeben. In der Java-Terminologie **redefinieren** (engl.: to redefine) sie die Operation der Klasse `Object`.
- Entscheidend ist: Durch die Redefinition erhalten die Klienten der Klasse ein anderes Ergebnis beim Aufruf der Operation `equals`.
  - Im Fall von Sammlungen: Ein Set verwendet die (gegebenenfalls redefinierte) Operation des Elementtyps.

Bekanntes Beispiel: die Klasse String

SE1 – Level 3 23

Aufgepasst: `equals` und `hashCode` hängen zusammen!

- In der Klasse `Object` ist in der Dokumentation der Operation `equals` folgender Hinweis zu finden:

  „Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.“

  (siehe: http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html)

- Wenn wir also für die Objekte einer Klasse selbst festlegen wollen, wann zwei Exemplare als gleich anzusehen sind, und deshalb die Methode `equals` redefinieren, dann **müssen** wir auch die Methode `hashCode` (ebenfalls in der Klasse `Object` definiert) so redefinieren, dass sie für zwei gleiche Objekte den gleichen `int`-Wert als Ergebnis liefert.

- Wir nehmen dies erst einmal als **Maßgabe** hin; auf Level 4 von SE1 werden wir die Mechanik von Hash-Verfahren untersuchen und dann diesen Zusammenhang verstehen können.

SE1 – Level 3 24

Das Interface `List` (relevanter Ausschnitt)

Indexbasierter Zugriff auf die Elemente

Indexbasierte Modifikatoren

Bestimmung eines Element-Index

Bildung von Teillisten

```
public interface List<E>
extends Collection<E>
{
  // Alle Operationen wie in Set
  ...

  // zusätzlich: indexbasierte Operationen
  E get(int index);
  E set(int index, E element);
  void add(int index, E element);
  E remove(int index);

  int indexOf(Object o);
  int lastIndexOf(Object o);

  List<E> subList(int from, int to);
}
```

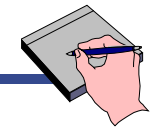
Beispiel einer List-Benutzung in Java

```
// Wir planen die Pausenaufsicht mit einer Liste von Namen…
// Exemplar einer Klasse erzeugen, die das Interface List implementiert
List<String> aufsichtsliste = new LinkedList<String>();
int laenge = aufsichtsliste.size();   // 0 (Liste anfangs leer)
aufsichtsliste.add("Peter");
aufsichtsliste.add("Paul");
aufsichtsliste.add("Mary");
aufsichtsliste.add("Peter");
aufsichtsliste.add("Jan");
// Duplikate sind erlaubt, also:
laenge = aufsichtsliste.size(); // 5

// Jan sollte doch die übernächste machen
aufsichtsliste.add(1,"Jan"); // Einfügen an Position, mit Verschieben des Restes

// Peter hat schon so oft beaufsichtigt…
aufsichtsliste.remove(aufsichtsliste.lastIndexOf("Peter"));
```

Auch dieses Beispiel soll ausschließlich die Kerneigenschaften einer List in Java veranschaulichen.

Die passende Visualisierung findet sich auf Folie 5.

Iterieren über Sammlungen



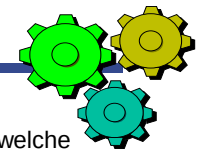
- Seit Java 1.5 gibt es eine **neue for-Schleife** (engl.: for-each loop), mit der sehr elegant über die Elemente einer Collection iteriert werden kann.

```
/**
 * Gib alle Personen in der Liste auf die Konsole aus.
 */
public void listeAusgeben(List<Person> personenliste)
{
    for (Person p: personenliste)
    {
        System.out.println(p.gibName());
    }
}
```

Lies als: „Für jede Person p in der personenliste…“

SE1 – Level 3 27

Vergleich der Schleifenkonstrukte in Java



- Wir kennen nun zwei sehr unterschiedliche for-Schleifen. Wann ist welche anzuwenden?
  - Die **neue for-Schleife** ist ausschließlich für Sammlungen vorgesehen. Wir verwenden sie beispielsweise, wenn wir einheitlich eine Operation auf **allen** Elementen einer Sammlung ausführen möchten.
  - Die **klassische for-Schleife** hingegen ist immer dann gut geeignet, wenn im Schleifenrumpf explizit Zugriff auf den Schleifenzähler benötigt wird oder wenn die Anzahl der Durchläufe vor der Schleifenausführung feststeht.
- Die **while-Schleifen** sollten eher in Fällen verwendet werden, in denen vorab unbekannt ist, wie viele Durchläufe es geben wird (beispielsweise beim Einlesen von Zeilen aus einer Datei oder wenn wir sequentiell in einer Sammlung nach einem Element mit bestimmten Eigenschaften suchen und abbrechen wollen, sobald das Element gefunden wurde).

SE1 – Level 3 28

Wrapper-Klassen in Java

- Als **Elementtyp** für die Sammlungen des Java Collection Framework sind **ausschließlich Referenztypen** zugelassen. Wir können also **nur Objekte** in einer Java Collection verwalten.
- Was aber ist, wenn wir eine **Menge von ganzen Zahlen** in unserer Anwendung brauchen? Oder eine **Liste von booleschen Werten**?
- Damit auch Elemente der primitiven Typen von Java in Sammlungen verwaltet werden können, ist für jeden primitiven Typ eine so genannte **Wrapper-Klasse** definiert:

Primitiver Typ		Wrapper-Klasse
int	→	Integer
boolean	→	Boolean
char	→	Character
long	→	Long
double	→	Double
float	→	Float
short	→	Short
byte	→	Byte

SE1 – Level 3 29

„Boxing“ und „Unboxing“ primitiver Typen

- Ein Wert eines primitiven Typs kann in einem Objekt des zugehörigen Wrapper-Typs „verpackt“ werden (engl.: **boxing**):

```
Integer iWrapper = new Integer(42);
```

- Die Referenz auf dieses Wert-Objekt kann dann in eine Menge eingefügt werden:

```
Set<Integer> intSet = new HashSet<Integer>();
intSet.add(iWrapper);
```

- Über die Operationen des Wrapper-Typs kann der verpackte Wert auch wieder „ausgepackt“ werden (engl.: **unboxing**):

```
int i = iWrapper.intValue();
```

- Für boolesche Werte analog:

```
Boolean bWrapper = new Boolean(true);
boolean b = bWrapper.booleanValue();
```

SE1 – Level 3 30

Auto-Boxing und Auto-Unboxing seit Java 1.5

- Weil das Ein- und Auspacken primitiver Werte mit Wrapper-Objekten zu aufgeblähten Quelltexten führt, wurden mit Java 1.5 einige Sprachregeln eingeführt, die die **automatische Umwandlung** regeln.

```
int i = 42;
Integer iWrapper = i;   // Auto-Boxing
```

- Dies funktioniert auch als aktueller Parameter:

```
List<Integer> intList = new LinkedList<Integer>();
intList.add(i);
```

- Auch das Auspacken wurde vereinfacht:

```
int i = iWrapper;   // Auto-Unboxing
```

- Ähnlich wie bei den Typumwandlungen zwischen den primitiven Typen passiert also sehr viel „hinter den Kulissen“!

SE1 – Level 3 31

Transitivität seit Java 1.5...

- Gleichheit ist üblicherweise **transitiv** definiert, mathematisch formuliert:

  a = b ^ b = c => a = c

- Gegeben folgende Java-Deklarationen:

```
Integer a = new Integer(5);
int b = 5;
Integer c = new Integer(5);
```

- Dann gilt wegen Auto-Unboxing:

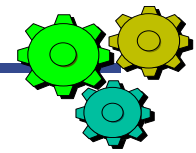
```
a == b  // automatisches Unboxing von a
b == c  //        "          "     "  c
```

- aber:

```
a != c
```

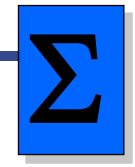
SE1 – Level 3 32

Welche Sammlung ist richtig für meine Zwecke?



1. **Auswahl eines Interfaces**
   - Zuerst sollte klar sein, welcher Umgang mit der Sammlung nötig ist (Duplikate erlaubt, Reihenfolge relevant, etc). Daraus folgt die Entscheidung für ein Sammlungs-Interface. Diese Entscheidung ist **problemabhängig**!
   - Alle Variablen im Klienten-Code sollten ausschließlich vom Typ dieses Interfaces sein.
2. **Auswahl einer Implementation**
   - Um mit diesem Interface wirklich arbeiten zu können, muss auch eine Implementation gewählt werden. Als erster Schritt ist jede Implementation ok, die das gewählte Interface implementiert.
   - Für `List` gibt es zwei Implementationen im JCF: `ArrayList` und `LinkedList`. Für `Set` gibt es ebenfalls zwei: `HashSet` und `TreeSet`.
   - Erst beim Tuning, wenn die Anwendung bereits ihren Zweck erfüllt und korrekt arbeitet, sollten Überlegungen darüber angestellt werden, ob eine andere Implementation eventuell besser geeignet wäre.

Zusammenfassung



- **Sammlungen** von Objekten werden bei praktisch jeder größeren Programmieraufgabe benötigt.
- **Listen** und **Mengen** sind zentrale Sammlungstypen, die viele Anwendungsfälle abdecken.
  - Listen haben eine manipulierbare Reihenfolge der Elemente und lassen Duplikate zu.
  - Mengen verwenden wir für Aufgaben, bei denen wir Duplikate vermeiden wollen und eine Reihenfolge nicht wichtig ist.
- Wir verwenden in Java die Interfaces `List` und `Set` aus dem **Java Collection Framework**, um den **Umgang** mit Listen und Mengen aus Klientensicht zu modellieren.
- Die Implementation dieser Interfaces interessiert uns als Klienten sehr häufig nicht.

