

Software-Entwicklung 1

V11: Sammlungen

Prof. Maalej & Team @maalejw

Überblick



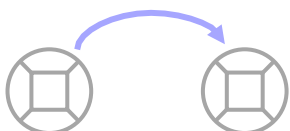
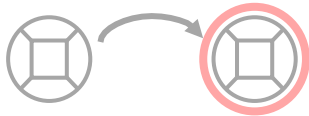
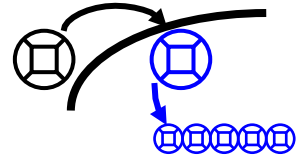
1

Objektsammlungen

2

Implementation von Sammlungen

Inhaltliche Gliederung von SE1

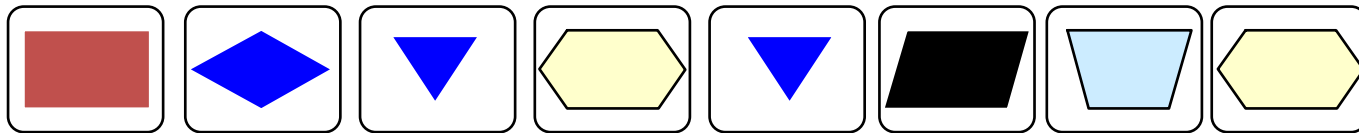
Stufe	Titel	Themen u.a.	Woche
1	 Algorithmisches Denken	Prozedur, Fallunterscheidung, Zählschleife, Bedingte Schleife	1 – 2
2	 Objektorientierte Programmierparadigma	Klasse, Objekt, Konstruktor Methode, Parameter, Feld, Variable, Zuweisung, Basistypen	3 – 5
3	 Benutzung von Objekten	Klasse als Typ, Referenz, UML Schleife, Rekursion, Zeichenketten	6 – 8
4	 Testen, Interfaces, Static, Arrays	Black-Box-Test, Testklasse, Interface, Sammlungen benutzen, Arrays	9 – 10
5	 Sammlungen	Sammlungen implementieren: Array-Liste, verkettete Liste, Hashing; Sortieren; Stack; Graphen	11 – 14

Objektsammlungen

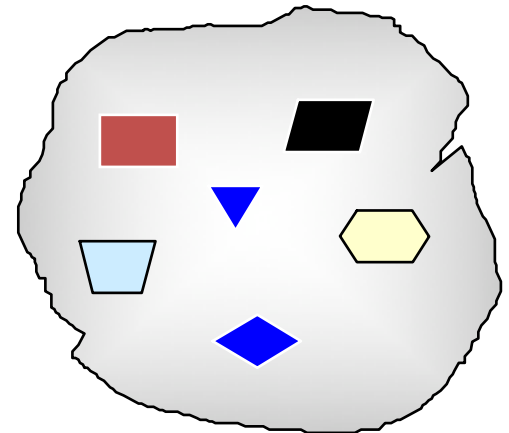
- Bei größeren Programmieraufgaben werden **gleichartige Objekte** zusammengefasst
 - Werden als eigene Objekte angesehen
- Programmiersprachen stellen **vordefinierte** Sammlungsbausteine zur Verfügung
 - Entweder als Teil der Sprache oder der Sprachbibliothek
- Zwei Sichten auf Objektsammlungen
 - Externe Sicht (Klientsicht): wie werden sie verwendet?
 - Interne Sicht: wie werden sie realisiert?

Mengen und Listen...

- ...Sammlungen, die in der theoretischen Informatik und in der Softwaretechnik häufig verwendet
- 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



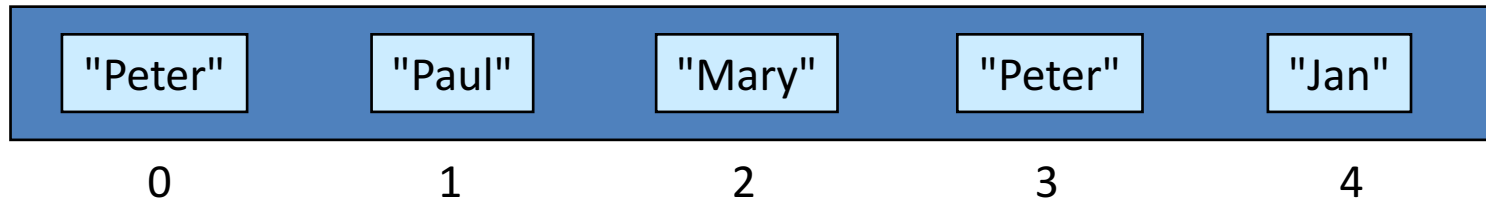
Liste, theoretisch



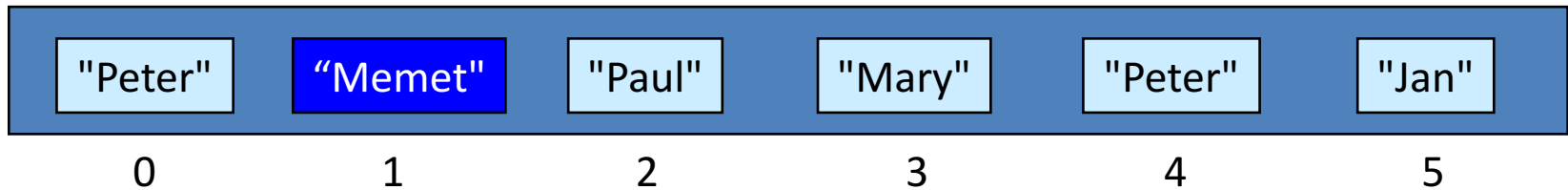
- Listen sind Aneinanderreihungen von **gleichartigen Werten** zu Folgen:
 - Die **Reihenfolge** der Listenelemente ist von Bedeutung
 - Ein Wert kann in einer Liste **mehrfach vorkommen**
- Mathematische (rekursive) Definition einer Liste:
 - Eine Liste ist entweder eine leere Liste (oft notiert als [])
 - 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 (mit Beachtung der Reihenfolge)

Umgang mit einer Liste – Beispiel

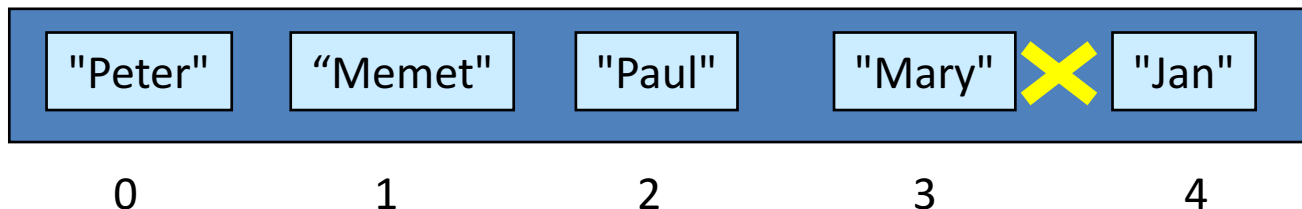
Eine Liste von Strings; Namen für die Einteilung der Pausenaufsicht in einer Schule:



Einfügen eines Namens an zweiter Position:



Entfernen des zweiten Eintrags für "Peter":



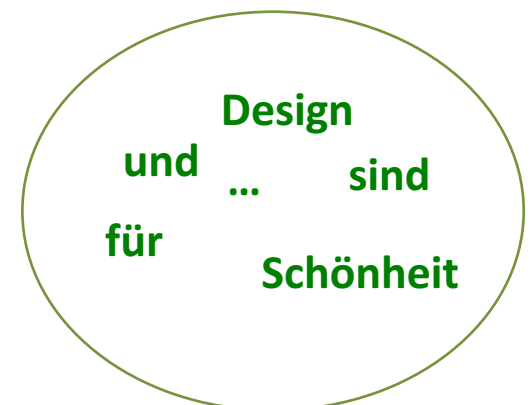
Menge, theoretisch



- 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

Umgang mit Mengen: Beispiel Textanalyse

- Einen längeren Text können wir als eine Liste von Wörtern ansehen
- Wir können ihn 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?
 - Welche Wörter bleiben übrig, wenn wir die Füllwörter von den Wörtern eines der Texte abziehen?



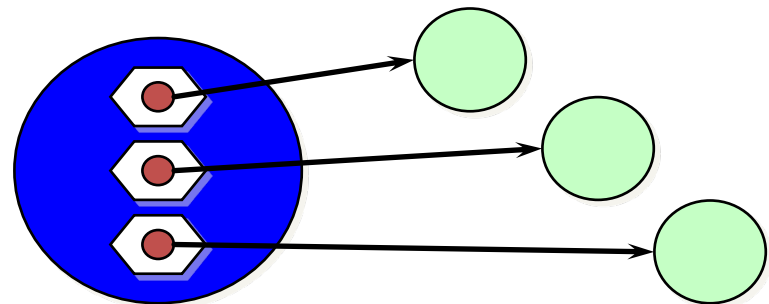
Listen und Mengen, objektorientiert

- Die theoretische Informatik beschreibt Sammlungen mit mathematischen (oft funktionalen) Konzepten
- Objektorientierte Sammlungen werden zustandsbasiert betrachtet:
 - Mengen und Listen als eigenständige Objekte unabhängig von ihren Elementen
 - Eine Menge ist 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 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

Auch bei Sammlungen gilt in Java:
sie enthalten keine Objekte, sondern
Referenzen auf Objekte!



Wichtige Begriffe zu Sammlungen

- **Elemente** der Sammlung sind Objekte, die in einer Sammlung vorhanden sind
- Jede Sammlung bietet **Operationen** zum Einfügen und Entfernen von Elementen
- Der **Elemententyp** (von enthaltenen Elementen) wird als eine Eigenschaft der Sammlung angesehen (z.B.: Sammlung von Strings)
- **Kardinalität** der Sammlung ist die Anzahl der enthaltenen Elemente
 - Sammlungen können i.R. 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**

Eigenschaften von Sammlungen

- Es ist deutlich zwischen der **Schnittstelle** der Sammlung (ihrem Umgang) und ihrer **Implementation** zu unterscheiden
- 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 ausführbar?

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)	Liste (List)	sortierte Liste (Sorted List)
Duplikate nicht zugelassen	Menge (Set)	geordnete Menge (Ordered Set)	sortierte Menge (Sorted Set)

Sammlungen in Java



- Java bietet 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
- Wird als Teil der Sprache betrachtet

Das Java API: die Standard-Bibliothek von Java

- 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 (immer vorhanden)
- Dazu gehört beispielsweise die Klasse String

Das Importieren von Bibliotheken

- 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

Eine
„Liste“
aller
Pakete

Overview (Java 2 Platform SE 5.0) - Mozilla

File Edit View Go Bookmarks Tools Window Help

Back Forward Reload Stop <http://docs.oracle.com/javase/1.5.0/docs/api/index.html> Search Print

Java™ 2 Platform Standard Ed. 5.0

Overview Package Class Use **Tree** Deprecated Index Help

PREV NEXT [FRAMES](#) [NO FRAMES](#)

Java™ 2 Platform Standard Edition 5.0
API Specification

This document is the API specification for the Java 2 Platform Standard Edition 5.0.

See: [Description](#)

Java 2 Platform Packages	
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.

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 Methoden
    int size();
    boolean isEmpty();
    boolean contains(Object o);

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

Dieses Interface (wie andere im JCF) 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 Methoden als Typ verwendet werden, hier etwa bei **add**.

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 2x 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 in seriösen Quelltext sein... 😊

Das Interface Set: zentrale Eigenschaften

- 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 Methode **equals** geprüft
- Formal gilt für alle $e1 \neq e2$ im Set: $!e1.equals(e2)$
- So genannte Massenoperationen (engl.: **bulk operations**) haben bei Sets die Bedeutung von mathematischen Mengenoperationen



<code>s1.containsAll(s2):</code>	<code>s2</code> Untermenge von <code>s1</code> ?
<code>s1.addAll(s2):</code>	<code>s1</code> = <code>s1</code> vereinigt mit <code>s2</code>
<code>s1.removeAll(s2):</code>	<code>s1</code> = <code>s1</code> - <code>s2</code>
<code>s1.retainAll(s2):</code>	<code>s1</code> = <code>s1</code> geschnitten mit <code>s2</code>

Die Methode equals

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

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

- Jedes Objekt kann über diese Methode gefragt werden, ob es gleich ist mit dem als Parameter angegebenen Objekt
- Parameter kann ein beliebiges Objekt sein
- Die Standardimplementierung vergleicht die Referenz des gerufenen Objektes mit der übergebenen Referenz

<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



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 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



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.

Iterieren über Sammlungen



- Seit Java 1.5 gibt es eine neue for-Schleife (engl.: for-each loop), mit der 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());
    }
}
```

„Für jede Person p in der personenliste...“

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).

Wrapper-Klassen in Java

- Als Elementtyp für die Sammlungen des Java Collection Framework sind ausschließlich Referenztypen zugelassen
- Wir können nur Objekte in einer Java Collection verwalten
- Was ist wenn wir eine Menge von **ganzen Zahlen oder** booleschen Werten in unserer Anwendung brauchen?
- Für jeden primitiven Typ gibt es eine so genannte Wrapper-Klasse:

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

„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();
```

Auto-Boxing und Auto-Unboxing seit Java 1.5

Weil das Ein- und Auspacken primitiver Werte mit Wrapper-Objekten zu aufgeblähtem Quellcode führt, wurden mit Java 1.5 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“!

Transitivität seit Java 1.5...

Gleichheit ist üblicherweise transitiv definiert, mathematisch formuliert:

$$a = b \wedge b = c \Rightarrow a = c$$

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
```

Welche Sammlung ist richtig für meine Zwecke?

- Auswahl eines Interfaces
 1. Zuerst sollte klar sein, welcher Umgang mit der Sammlung nötig ist (Duplikate erlaubt, Reihenfolge relevant, etc)
 2. Daraus folgt die Entscheidung für ein Sammlungs-Interface. Diese Entscheidung ist problemabhängig!
 3. Alle Variablen im Klienten-Code sollten ausschließlich vom Typ dieses Interfaces sein
- Auswahl einer Implementation
 1. 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
 2. Für List gibt es zwei Implementationen im JCF: **ArrayList** und **LinkedList**. Für Set gibt es ebenfalls zwei: **HashSet** und **TreeSet**
 3. Erst beim Tuning, wenn die Anwendung bereits ihren Zweck erfüllt und korrekt arbeitet, sollten wir überlegen, ob eine andere Implementation besser geeignet wäre

Vergleich Sammlung und Array

- Die Java-Sammlungen wie List oder Set sind beschränkt auf **Sammlungen von Referenzen**:
 - D.h. es können nur Referenztypen als Elementtypen definiert werden; Arrays erlauben hingegen beide Typfamilien als Elementtyp
- Nach seiner Erzeugung kann ein **Array seine Größe** nicht mehr verändern!
 - List und Set sind dynamische Sammlungen und können beliebig viele Elemente aufnehmen
- Ein Array modelliert **zusammenhängende Speicherzellen**
 - der schreibende Zugriff auf eine Position **verschiebt nicht** alle nachfolgenden Elemente sondern ersetzt das Element an der Position! Ein einfügen muss durch ein verschieben „von Hand“ realisiert werden.
- Arrays werden in der Implementierung von dynamischen Sammlungen verwendet
 - sie stehen auf einem niedrigeren Abstraktionsniveau

Zusammenfassung

1

Sammlungen von Objekten werden bei größeren Programmieraufgabe benötigt

2

Listen (geordnet mit Duplikate) und Mengen (ohne Ordnung und ohne Duplikate) sind zentrale Sammlungstypen, die vieles abdecken

3

Im **Java Collection Framework**, gibt es **List** und **Set** Interfaces um den Umgang mit Listen und Mengen aus Klientensicht zu modellieren

4

Die Implementation dieser Interfaces interessiert uns als Klienten sehr häufig nicht