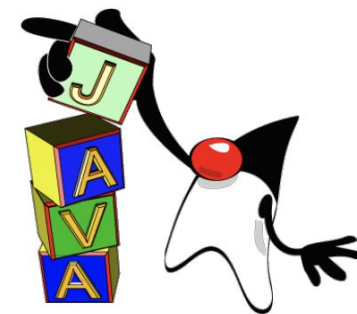
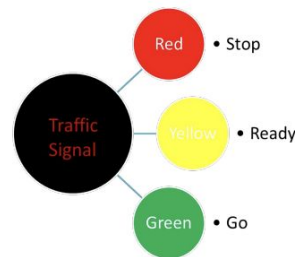
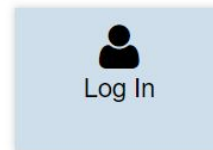


# Einführung in die Programmiersprache Java III

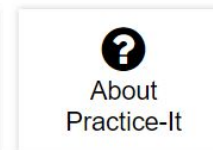




Practice-it is a web application to help you practice solving Java programming problems online. Many of the problems come from the University of Washington's introductory Java courses.

To use Practice-it, first create an account, then choose a problem from our list. Type a solution and submit it to our server. The system will test it and tell you whether your solution is correct.

Version 4.1.13 (2021-03-09)



(To submit a solution for a problem or to track your progress, you must create an account and log in.)

---

Is there a problem? [Contact a site administrator.](#)



# Inhalt

- Enums
- Dokumentation in Java
- Generics
- Interfaces
  - Standard Interfaces in Java

# Konstanten

Konstante Werte sollten einmal definiert und dann mit ihrem Namen benutzt werden.

- Lesbarkeit, Vermeidung von Tippfehlern
- ggf. leichte Änderbarkeit von Werten

```
public final class Math {  
    /** * The {@code double} value that is closer than any  
    other  
    * to  $\pi$ , the ratio of the circumference of a  
    * circle to its diameter.  
    */  
  
    public static final double PI = 3.14159265358979323846;
```



# Aufzählungstypen (Enums)

bestehen aus einer festen (und normalerweise kleinen) Anzahl benannter Konstanten

Bspiele:

- Spielkarten: Karo, Kreuz, Herz, Pik
- Wochentage: Montag, . . . , Sonntag
- Noten: Sehr gut, . . . , Ungenügend
  
- Java5+
- final-Konstanten vom Typ int

# Konstanten und Aufzählungen

```
class Weekdays { //bis Java 5
    public static final int MONDAY = 0;
    public static final int TUESDAY = 1;
    public static final int WEDNESDAY = 2;
    public static final int THURSDAY = 3;
    public static final int FRIDAY = 4;
    public static final int SATURDAY = 5;
    public static final int SUNDAY = 6;
}
```

## Probleme mit diesem Ansatz

- Die Werte sind alle vom Typ `int`
  - `MONDAY` kann benutzt werden, wo eigentlich eine Jahreszahl erwartet würde.
- das Hinzufügen oder das Löschen von Werten ist gefährlich.
  - Nach einem Update kann der Code für eine Option eine komplett andere Bedeutung haben
- die Iteration über alle möglichen Werte einer Art ist fragil
  - Man muss die Anzahl der Werte kennen und wissen, wie sie durchnummeriert sind.

# Aufzählungstypen

- Enums erlauben eine bessere Kodierung endlicher Aufzählungen.

```
public enum Weekday {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,  
    SATURDAY, SUNDAY  
}
```

Vorteile gegenüber Zahlkonstanten:

- einfache Kodierung
- vermeidet typische Fehler
- leicht lesbare Fallunterscheidung
- Iteration



# Aufzählungstypen

- Deklaration der Form `enum A { ... }` wird vom Compiler in normale **Klasse** übersetzt
- Enum-Typen können auch Methoden haben
- Konstanten können assoziierte Werte haben
- `public enum A extends B { ... }` nicht zulässig

```
enum Color {  
    RED, BLUE, YELLOW;  
}
```



```
public final class Color {  
    private Color() {}  
    public static final Color RED = new Color();  
    public static final Color BLUE = new Color();  
    public static final Color YELLOW = new Color();  
}
```

# Switch

## Enums erlauben Fallunterscheidungen mit **switch**

```
boolean isWorkday(Weekday t) {  
    switch(t) {  
        case MONDAY:  
        case TUESDAY:  
        case WEDNESDAY:  
        case THURSDAY:  
        case FRIDAY: return true;  
        case SATURDAY:  
        case SUNDAY: return false;  
        default: // kann gar nicht passieren throw new IllegalArgumentException();  
    }  
}
```

# Iteration

- Enums erlauben Iteration über alle Werte

```
for (Weekday d : Weekday.values()) {  
    System.out.println(d.toString());  
}
```

- gibt aus:

MONDAY/TUESDAY/WEDNESDAY/THURSDAY/FRIDAY/SATURDAY/SUNDAY

- Jede Enum-Klasse hat eine statische Methode `values()`, die eine Collection aller Werte dieses Typs zurückgibt.

# Beispiel

```
public class EnumDemo {  
  
    public enum Food {  
        HAMBURGER(7), FRIES(2), HOTDOG(3), ARTICHOKE(4);  
  
        Food(int price) {  
            this.price = price;  
        }  
  
        private final int price;  
  
        public int getPrice() {  
            return price;  
        }  
    }  
  
    public static void main(String[] args) {  
        for (Food f : Food.values()) {  
            System.out.print("Food: " + f + ", ");  
  
            if (f.getPrice() >= 4) {  
                System.out.print("Expensive, ");  
            } else {  
                System.out.print("Affordable, ");  
            }  
  
            switch (f) {  
                case HAMBURGER:  
                    System.out.println("Tasty");  
                    continue;  
                case ARTICHOKE:  
                    System.out.println("Delicious");  
                    continue;  
                default:  
                    System.out.println("OK");  
            }  
        }  
    }  
}
```

## Zu beachten

- Konstruktoren in Enum-Typen nicht public machen
- Enum-Typen können nicht mithilfe von extends etwas erweitern
- Werte eines Enum-Typs sind automatisch geordnet, wie üblich mit compareTo erfragen
- Parameter an Konstanten können sogar Methoden sein
- statische Methode `values()` liefert Collection der einzelnen Werte, kann z.B. mit Iterator durchlaufen werden



# Dokumentation

- Dokumentation soll helfen,
  - Schnittstellen zu verstehen
  - Entwurfsideen zu erklären
  - implizite Annahmen auszudrucken
- Nicht sinnvoll:
  - `x++; // erhöhe x um eins`



# Was soll man dokumentieren?

- für jede Methode eine Zusammenfassung, was es macht
- Infos über die Parameter
- wie Fehler behandelt und an den Aufrufer der Methode zurückgegeben werden
- Verträge
  - Vorbedingungen (Postcondition)
  - Nachbedingungen (Precondition)

# Tags

- `@author Name`
- `@version text`

## Vor Methoden

- `@param Name Beschreibung`
  - beschreibt einen Parameter einer Methode
- `@return Beschreibung`
  - beschreibt den Rückgabewert einer Methode
- `@throws Exception Beschreibung`
  - beschreibt eine Exception, die von einer Methode ausgelöst werden kann



# Beispiel

```
/**
 * Allgemeine Kontenklasse
 * @author Marcus Licinius Crassus
 * @see NichtUeberziehbaresKonto
 */
public class Konto {

    /**
     * Geld auf Konto einzahlen.
     * <p>
     * Wenn vorher {@code getKontoStand() = x}
     * und {@code betrag >=0},
     * dann danach {@code getKontoStand() = x + betrag}
     * @param betrag positive Zahl, der einzuzahlende Betrag
     * @throws ArgumentNegativ wenn betrag negativ
     */
    public void einzahlen(double betrag);
}
```

# erzeugte htmlDokumentation

Package **Class** Tree Deprecated Index Help

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

## Class Konto

java.lang.Object  
Konto

---

public class **Konto**  
extends java.lang.Object

Allgemeine Kontenklasse

See Also:

NichtUeberziehbaresKonto

---

## Constructor Summary

**Constructors**

Constructor and Description
Konto()

---

## Method Summary

**Methods**

Modifier and Type	Method and Description
void	einzahlen(double betrag) Geld auf Konto einzahlen.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

---

## Constructor Detail

**Konto**

```
public Konto()
```



# Kommentare

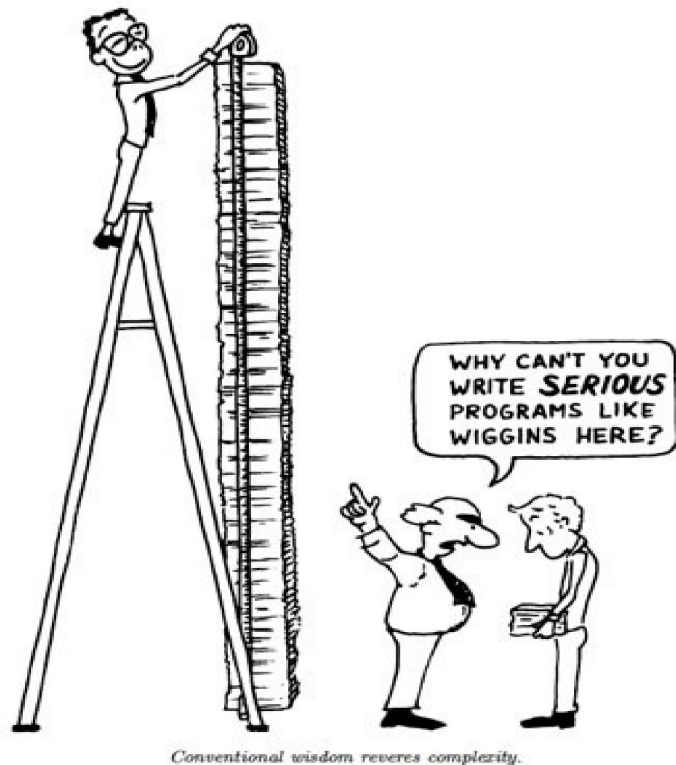
- Javadoc erlaubt spezielle Kommentare
  - die automatisch aus dem Code herausgezogen sind
- man kann die vor Packages, Klassen, Methoden, Variablen verwenden
- Ausgabe

```
javadoc Package
```

```
javadoc Klasse1.java Klasse2.java ...
```

# Generics

- Seit Java 1.5 ermöglicht
- parametrisierte Typen
- Templates?



```
1 Stiva s=new Stiva();  
2  
3 s.pune("Ana");  
4 s.pune(new Persoana("Ana", 23));  
5  
6 Persoana p=(Persoana)s.scoate();  
7 Persoana p2=(Persoana)s.scoate();
```

# Realisierung

- Bis Java 1.4 wurde Typ-Polymorphie durch Vererbung von der Klasse Object realisiert

```
public class OldBox {  
    private Object contents;  
  
    public Box(Object cont) {  
        contents = cont;  
    }  
    public Object getContents() {  
        return contents;  
    }  
    public void setContents (Object o) {  
        contents = o;  
    }  
}
```

```
public class Tester {  
    public static void main(String[] args){  
        OldBox b1 = new OldBox("Apple");  
        String s = (String) b1.getContents();  
        System.out.println(s);  
  
        OldBox b = new OldBox(new Integer(3));  
        int i = (Integer) b.getContents();  
        System.out.println(s);  
    }  
}
```



# Syntax

- Deklaration einer generischen Klasse:

```
[zugriff_mod] class <name> [<Typ1,...>] {  
    private Typ1 var;  
    [attr]  
    [meth]  
}
```

- Beispiele für Instantiierungen von Box:

```
Box<String>
```

```
Box<Integer>
```

```
Box<Box<Integer>>
```

```
Box<int> // Grunddatentypen
```

```
//nicht erlaubt als Typparameter
```



## Beispiel für Box

```
public class Box<T> {  
    private T contents;  
    public Box(T cont) { contents = cont; }  
    public T getContents() { return contents; }  
    public void setContents(T o) { contents = o; }  
}  
  
public class Tester {  
    public static void main(String[] args){  
        Box<String> b = new Box<String>("Apple");  
        String s = b.getContents(); System.out.println(s);  
        Box<Integer> b1 = new Box<Integer>(new Integer(3));  
        int i = b1.getContents(); System.out.println(i);  
    }  
}
```



# Typ-Parameter

- Grunddatentypen sind nicht erlaubt als Typparameter:
  - `Box<int> bi=new Box<int>();`
  - `Box<Integer> bi=new Box<Integer>();`
- `boolean -> Boolean`
- `byte -> Byte`
- `short -> Short`
- `int -> Integer`
- ...





# Autoboxing

- Java 1.5
- automatische Konversion zwischen prim. Typen und Klassen

```
Box<Integer> bi = new Box<Integer>();  
bi.add(23); //autoboxing  
bi.add(new Integer(23));  
  
int val = bi.get();
```



## generische Methoden

- Deklaration einer generischen Methode

```
[zugriff_mod] class <name> [<Typ1,...>] {  
    [zugriff_mod] [<Typ1,...>] <typ> <name> ([params])  
}
```

- Beispiel

```
public class generics {  
    public <T> void f(T x) {  
        System.out.println(x.toString());  
    }  
    public static <T> void copy(T[] elems, Box<T> b) {  
        for (T e:elem) b.add(e);  
    }  
}
```



## generische Methoden

- Aufruf einer generischen Methode

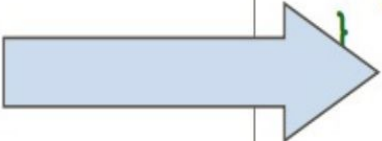
```
public class A {  
    public <T> void print (T x) {  
        System.out.println(x);  
    }  
    public void static main (String[] arr) {  
        A a = new A();  
        a.print(23);  
        a.print("Ana");  
        a.print(new Person("Ana", 23));  
    }  
}
```

## Erasure und Bounds

- Für jede parametrisierte Klasse wird genau eine Klasse erzeugt,
  - die die generischen Typinformationen löscht („type erasure“)
  - und die Typparameter durch die Klasse Object ersetzt
- die Typparameter werden durch ihre oberen Schranken ersetzt (zB Object)
- Interoperabilität
  - Möglichkeit der Übersetzung nach Java (pre 1.5) ohne generische Typen

```
public class A {  
    public String f (Integer ix){  
        Stiva<String> st=new Stiva<String>();  
        Stiva sts=st;  
        sts.pune(ix);  
        return st.varf();  
    }  
}
```

**compilation**



```
public class A {  
    public String f (Integer ix){  
        Stiva st=new Stiva();  
        Stiva sts=st;  
        sts.pune(ix);  
        return (String)st.varf();  
    }  
}
```



# Vererbung und Generische Typen

- man kann von generischen Klassen Unterklassen ableiten
- Diese Unterklassen können, aber müssen nicht selbst generische Klassen sein
- Sei A Subtyp von B und G eine generische Klasse
  - $G<A>$  is kein Subtyp von  $G<B>$



# Vererbung und Generische Typen

```
public class SecuredBox<T,L> extends Box<T> {  
    private L lock;  
  
    public SecuredBox(T content, L lock) {  
        super(content);  
        this.lock = lock;  
    }  
  
    public L getLock() {...}  
    public void setLock(L lock) {...}  
}
```



## Bounds <<Typvariable>> extends <<Typausdruck>>

- nur solche Typparameter, die von <<Typausdruck>> erben, zuzulassen sind

```
public class Apple extends Fruit{}
public class Steak {}
public class Box <T extends Fruit> {}
public Test {
    public static void main(String[] arr) {
        Apple a1 = new Apple();
        Steak s1 = new Steak();
        Box<Apple> aBox = new Box(a1); //OK
        Box<Steak> sBox = new Box(s1); //Fehler
    }
}
```



# Wildcards

- wir haben die Folgende Situation

```
public class Apple extends Fruit{}
public class Box <T extends Fruit> {}
public static void print_box(Box<Fruit> f) {
    System.out.println(f);
}
```
- Apple ist eine Unterklasse von Fruit
- Box<Apple> ist keine Unterklasse von Box<Fruit>
- druckt nur Objekte der Klasse **Box<Fruit>**





# Wildcards

- Man darf einen Platzhalter benutzen
  - name <?>

```
public static void print_box(Box<?> f) {  
    System.out.println(f);  
}
```

```
//...
```

```
Box<Apple> box = new Box<Apple>(new Apple());  
print_box(box);
```



## Bounded Wildcards

- name <? extends <<Typeausdruck>> >
- beschränkt auf Subtypen von <<Typeausdruck>>
- name <? super <<Typeausdruck>> >
- beschränkt auf Obertypen von <<Typeausdruck>>

```
public static void print_box(Box<? extends Fruit> f) {  
    System.out.println(f);  
}  
//...  
Box<Apple> box = new Box<Apple>(new Apple());  
print_box(box);
```



# Exkurs: Stack Implementation

Wir wollen eine generische Stack Klasse umsetzen.

- pop()
- push()
- peek()





# Abstrakte Klasse

- Eine Klasse, in der einige Methoden nicht implementiert werden
  - in JAVA abstract by definition
- diese Methoden heißen abstrakte Methoden
- müssen dann in jeder Unterklasse implementiert werden



# Interface

- Eine spezielle Art von Klassen, in der alle Methoden abstrakt sind
- Dienen zur Spezifikation einer **Schnittstelle**,
  - die dann von verschiedenen Klassen implementiert werden kann
- Eine Klassen kann mehrere Interfaces implementieren



## Ziel von Interfaces

- Interfaces trennen den Entwurf von der Implementierung
- Interfaces legen Funktionalität fest
  - ohne auf die Implementierung einzugehen
- Für Interfaces ist die spätere Verwendung nicht von Bedeutung
  - sondern nur die bereitzustellende Funktionalität
- Als Konsequenz interessiert der Anwender sich nicht für die Implementierungsdetails
  - sondern für die Funktionalität



# Vorteile

- Ist einmal ein Interface vorhanden, so hat man ein klares und kleineres Ziel
- Es lassen sich bessere Tests schreiben, da die Funktionalität genau festgelegt ist
- Während der Implementierung braucht man nicht darüber nachzudenken, wo es dann verwendet wird

## Exkurs: Interfaces vs Klassen

Implementieren Sie Klassen, Interfaces und abstrakte Klassen, welche die folgenden Bedingungen erfüllen

- Es gibt verschiedene Arten von Tieren: Säugetiere und Vögel
- Einige Tiere können fliegen, andere jagen. Einige Tiere können beides
- Wir haben Pinguine, Hühner, Fledermäuse und Adler
- Es gibt auch Jäger, aber sie können nur Vögel jagen, die jagen können. Die Jäger-Klasse stellt eine Methode `hunt()` bereit.

Jedes Tier hat als Attribute Name und Alter.







## aus der Standardbibliothek

- `LinkedList<E>` ist eine Implementierung des Interfaces `List<E>`
- Verwendung
  - `List<E> foo = new LinkedList<E>();`

anstelle von

- `LinkedList<E> foo = new LinkedList<E>();`
- so kann man jederzeit `LinkedList` durch eine andere Implementierung des Interfaces `List` ersetzen

# aus der Standardbibliothek

java/lang/Comparable.java (Kommentare gekürzt):

```
/* Copyright 1997-2006 Sun Microsystems, Inc.  
   LICENSE: GPL2 */  
package java.lang;  
import java.util.*;  
  
/**  
 * Compares this object with the specified object  
 * for order. Returns a negative integer, zero, or  
 * a positive integer as this object is less than,  
 * equal to, or greater than the specified object.  
 */  
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```



## aus der Standardbibliothek

```
public class Person implements Comparable<Person> {  
    private double size;  
    private String name;  
  
    public Person (double size, String name) {  
        this.size = size; this.name = name;  
    }  
    public int compareTo (Person o) {  
        if (size < o.size) return 1;  
        else if (size == o.size) return 0;  
        else return -1;  
    }  
}
```



## Multiple Interfaces

- eine Klasse darf mehrere Interfaces implementieren
- ein Beispiel

```
public interface Grow {  
    public void grow(double x);  
}  
  
public static void main (String[] arr) {  
    Person bob = new Person (1.80, "bob");  
    Person lob = new Person (1,20, "lob");  
    System.out.println(bob.compareTo(lob));  
    lob.grow(10);  
    Grow g = bob; Comparable<Person> cp = bob;  
}
```

## aus der Standardbibliothek

```
public class Person implements Comparable<Person>, Grow {  
    //constructor  
    // variablen  
  
    public grow (double x) {  
        size += x;  
    }  
    public int compareTo (Person o) {  
        if (size < o.size) return 1;  
        else if (size == o.size) return 0;  
        else return -1;  
    }  
}
```

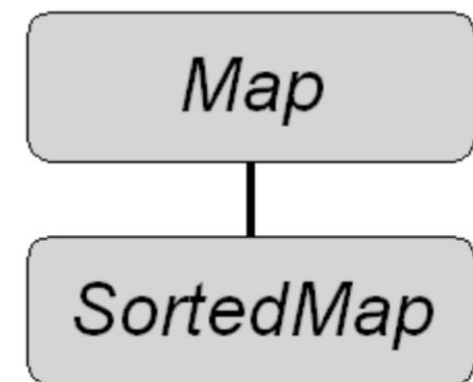
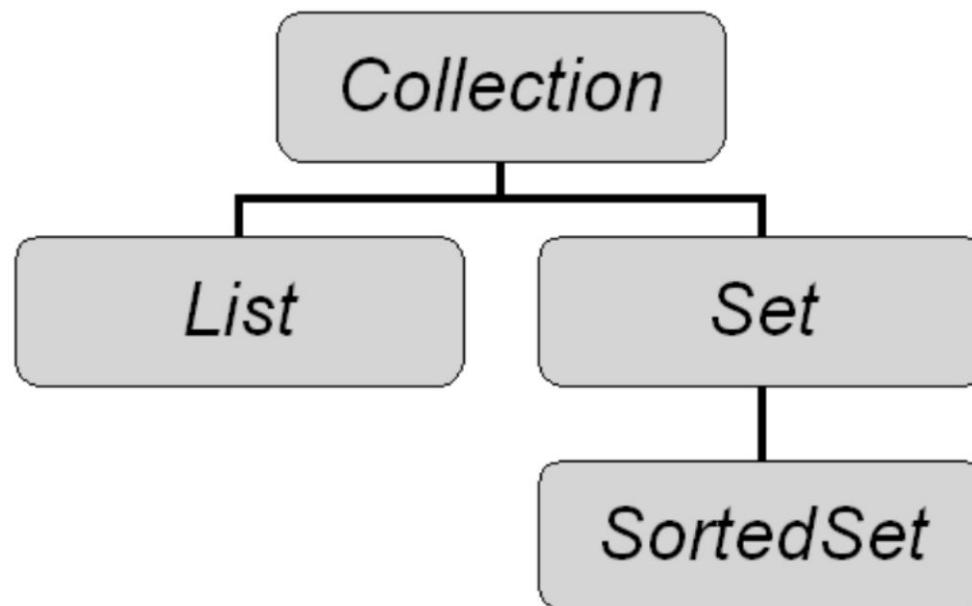


## Methoden der Klasse Object

- `java.lang.Object` enthält eine ganze Menge von Methoden
- `boolean equals (Object obj)`
- `String toString ()`
- Da jede Klasse von `Object` erbt, stehen diese Methoden in jeder Klasse zur Verfügung
  - mit genau dieser Semantik
  - Man kann sie jedoch überschreiben, um eine andere Bedeutung zu realisieren

# Java Collection Framework

- eine Sammlung von Interfaces, die die Organisation von Objekten in “Containers” unterstützt





# Die wichtigsten Elemente

## `java.util.Collection`

- Interface, um eine Gruppe von Objekten zu organisieren
- Basis-Definitionen für Hinzufügen und Entfernen von Objekten

## `java.util.List`

- Collection Interface, das zusätzlich jedem Element eine fixe Position zuweist





# Die wichtigsten Elemente

## `java.util.Set`

- Collection Interface, das keine doppelten Elemente erlaubt

## `java.util.Map`

- Interface, das die Zuordnung von Elementen zu sogenannten Schlüsseln unterstützt
- erlaubt, Elemente mit dem zugehörigen Schlüssel anzusprechen
- Map ist keine Unterklasse von Collection



## Hilfs-Interfaces

### `java.util.Iterator`

- Interface, das Methoden spezifiziert, die es erlauben, alle Elemente einer Collection aufzuzählen
- ersetzt weitgehend das ältere `java.util.Enumeration` Interface

### `java.util.Comparator`

- Interface, das Methoden zum Vergleich von Elementen einer Collection spezifiziert



## Wichtige Methoden

`boolean add(Object obj)`

- füge obj zur Collection hinzu
- return true wenn sich die Collection dadurch verändert hat

`boolean contains(Object obj)`

- return true wenn obj bereits enthalten ist

`boolean isEmpty()`

- return true wenn die Collection keine Elemente enthält



## Wichtige Methoden

`Iterator iterator()`

- return ein Iterator Objekt, mit dem man die Elemente einer Collection aufzählen kann

`boolean remove(Object obj)`

- entfernt ein Element, das equal zu obj ist, falls eins existiert
- return true, falls sich die Collection dadurch verändert hat

`int size()`

- return die Anzahl der Elemente in der Collection



## Weitere Methoden

`boolean addAll(Collection c)`

- fügt zur Collection alle Objekte aus der Collection c hinzu  
boolean

`containsAll(Collection c)`

- true wenn alle Objekte aus der Collection c enthalten sind

`boolean removeAll(Collection c)`

- entfernt alle Objekte aus der Collection, die sich in einer anderen Collection c befinden



## Weitere Methoden

`boolean retainAll(Collection c)`

- behalte nur Objekte, die sich auch in der Collection c befinden

`void clear()`

- entfernt alle Objekte aus der Collection

`Object[] toArray()`

- return die Elemente der Collection in einem Array

`Object[] toArray(Object[] a)`

- return einen Array vom selben (dynamischen) Typ wie a



## java.util.List

- Spezifiziert eine Collection, bei der die Elemente durchnummeriert sind
  - ähnlich wie in einem Array
  - aber mit flexibleren Zugriffsmöglichkeiten
- realisiert als Unterklasse von `java.util.Collection`
  - das heißt, Objekte, die dieses Interface implementieren, müssen alle Collection Methoden unterstützen
  - und zusätzlich noch Methoden, die einen Zugriff über die Position des Elements erlauben



## zusätzliche Methoden

`Object get(int i)`

- gibt das i-te Element zurück

`Object set(int i, Object o)`

- weise dem i-ten Element das Objekt o zu

`int indexOf(Object o)`

- Index des ersten Objekts, für das `equals(o)` gilt
- -1 falls es kein so ein Element gibt





## zusätzliche Methoden

`void add(int i, Object o)`

- fügt o an der i-ten Stelle der Liste ein

`Object remove(int i)`

- entfernt und retourniert das Objekt an der i-ten Stelle List

`subList(int von, int bis)`

- gibt die Teil-Liste beginnend mit von, endend mit bis-1 zurück



# Vordefinierte Listen-Klassen

## LinkedList

- Implementiert eine Liste mit expliziter Verkettung
  - d.h. in den Datenkomponenten wird ein Verweis auf das nächste und vorhergehende Listen-Element abgespeichert
- rekursive Datenstrukturen

## ArrayList

- Implementiert eine Liste mittels eines Arrays
- d.h. die Elemente der Liste werden in einem Array abgespeichert



# Vordefinierte Listen-Klassen

## Vor- und Nachteile:

- ArrayList ist schneller im Zugriff auf indizierte Elemente
  - da sich die Adresse direkt berechnen läßt
- LinkedList ist schneller im Einfügen und Entfernen
  - da die restlichen Einträge der Liste unberührt bleiben.



# Konzept von Iterable

- In vielen Fällen gibt es eine endliche Ansammlung von Elementen, die man alle durchlaufen möchte
  - Datentypen: `List<T>`, `Set<T>`, `Map<T>` . . .
- Vereinheitlichung: Interface `Iterable`
- `Iterator<T> iterator()`
- `Iterator` stellt das gewünschte Durchlaufen bereit



## java.util.Iterator

boolean hasNext()

- überprüft, ob die Collection noch zusätzliche Elemente hat

Object next()

- returns das nächste Objekt in der Collection

void remove()

- entfernt das letzte Element, das vom Iterator retourniert wurde, aus der Collection



## java.util.Iterator

```
Iterator<T> iter = collection.iterator();  
while(iter.hasNext()){  
    T elem = iter.next(); ... // Code, der mit elem etwas  
    macht  
}
```

- Man kann auch kurz schreiben

```
for(T elem : colle){ ... // Code, der mit elem etwas  
macht }
```



## Effizienter mittels Iterator

- Wenn foo zur Klasse `LinkedList<T>` genügt, ist

```
for(T elem : colle){ // Code, der mit elem etwas macht }
```

- erheblich schneller als

```
for(int i = 0; i < colle.size(); ++i){  
    T elem = colle.get(i); ... // Code, der mit elem etwas  
    macht  
}
```



## Vorteile durch Iterable

- Einheitlicher
- Übersichtlicher
- Kürzer
- In manchen Fällen schneller





# Java.util.Set

Spezifiziert eine Menge

- also eine Collection, in der kein Element doppelt vorkommen darf
- implementiert genau die Methoden, die für Collection vorgeschrieben sind
- aber keine zusätzlichen Methoden

## java.util.HashSet

- implementiert das Interface `Set` mit Hilfe einer Hash-Tabelle

```
class HashSetTest{  
    public static void main(String args[]){  
        HashSet<String> set=new HashSet();  
        set.add("One");set.add("Two");  
        set.add("Three"); set.add("Four");  
        Iterator<String> i=set.iterator();  
        while(i.hasNext()) {  
            System.out.println(i.next());  
        }  
    }  
}
```



# java.util.Collections

- Methoden zum Sortieren
  - `static void sort(List list)`
    - Sortieren nach natürlicher Ordnung der Objekte
  - `static void sort(List list, Comparator c)`
    - Sortieren nach dem Comparator-Objekt
- Weitere Methoden zum
  - Suchen
  - Kopieren
  - Mischen



# java.util.SortedSet

- wie `java.util.Set` aber die Elemente sortiert werden müssen
- also wieder nur ein semantischer Unterschied in der Implementierung des Interfaces
- Um sortieren zu können, muss man Elemente vergleichen können



# java.util.Comparator

- ein eigenes Objekt zum Vergleichen
- stellt einzige Methode bereit, die implementiert werden muss
- `int compare(Object o1, Object o2)`
- Rückgabewert: 0,+,-

# java.util.Comparable

- definiert eine totale Ordnung
- `int compareTo(Object o)`
  - vergleicht dieses Objekt mit dem Objekt o
- Rückgabewert: 0, + , -

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<Person> leute = new ArrayList<Person>();
        leute.add(new Person(15, "BOB"));
        leute.add(new Person(28, "DOB"));
        leute.add(new Person(19, "LOB"));
        leute.add(new Person(33, "ZOB"));
        Collections.sort(leute);
        for(Person p : leute)
            System.out.println(p.toString());
    }
}
```

```
import java.util.*;
public class Person implements Comparable<Person> {
    int alter; String name;
    public Person(int alter, String name) {
        this.name = name;
        this.alter = alter;
    }
    public String toString() {
        return(name + " ist " + alter + " Jahre alt");
    }
    @Override
    public int compareTo(Person o) {
        return(this.alter - o.alter);
    }
}
```



# java.util.Map

- realisiert einen assoziativen Speicher
- Schlüssel (Keys):
  - sind beliebige Objekte
  - jeder Schlüssel kann nur maximal einmal vorkommen
- Werte (Values):
  - sind ebenfalls beliebige Objekte
  - jedem Schlüssel wird genau ein Wert zugeordnet



# java.util.Map

- `Object get(Object key)`
- `Object put(Object key, Object val)`
- `Object remove(Object key)`
- `boolean containsKey(Object key)`
- `boolean containsValue(Object val)`
- `Set keySet()`
- `Collection values()`





# java.util.HashMap

- realisiert eine Map mit einer sogenannten Hash-Tabelle
- Jedem Objekt ist eine fixe Zahl zugeordnet, der sogenannte Hash-Cod
- MD5 SHA-1
- Hash-Codes



# Hash-Codes

- für jedes Objekt muss ein Integer Code retourniert werden
- während eines Ablaufs des Programms muss immer der gleiche Code retourniert werden
- bei verschiedenen Abläufen können es auch verschiedene Codes sein
- es ist nicht verlangt, dass das zwei verschiedene Objekte verschiedene HashCodes retournieren
  - wäre aber gut

## java.util.HashMap

```
public class HashMapTest{  
    public static void main(String args[]){  
        HashMap<Integer,String> map=new  
            HashMap<Integer,String>();//Creating HashMap  
        map.put(1,"Mango");map.put(2,"Apple");  
        map.put(3,"Banana"); map.put(4,"Grapes");  
  
        System.out.println("Iterating Hashmap...");  
        for(Map.Entry m : map.entrySet()){  
            System.out.println(m.getKey()+" "+m.getValue());  
        }  
    }  
}
```



# Maps mit sortierten Schlüsseln

## Interface `java.util.SortedMap`

- Eine **Map**, bei der die Schlüssel sortiert bleiben müssen
- d.h. die Schlüssel bilden kein **Set**, sondern ein **SortedSet**

## Klasse `java.util.TreeMap`

- Klasse, die das **SortedMap** Interface implementiert