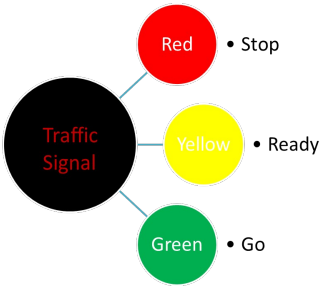
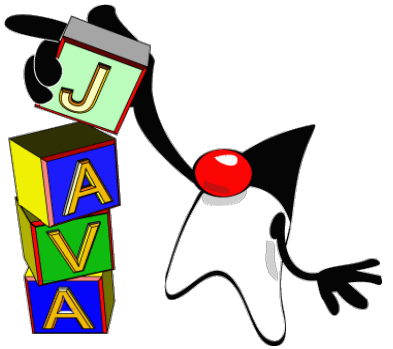
Einführung in die Programmiersprache Java III





Konstanten, Enums

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 <i>pi</i>, the ratio of the circumference of a
* circle to its diameter.
*/
public static final double PI = 3.14159265358979323846;
```

Konstanten

- nicht überall schreiben:

```
for (int x = 0; x < 128; x++) {
for (int y = 0; y < 128; y++) {
```

- Stattdessen
  - variablen
  - oder symbolische Werte:

```
public static final int FIELD_WIDTH = 128;
public static final int FIELD_HEIGHT = 128;
for (int x = 0; x < FIELD_WIDTH; x++) {
for (int y = 0; y < FIELD_HEIGHT; y++) {
...
```

Aufzählungstypen

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

- Bspiele:
  - Spielkarten: Karo, Kreuz, Herz, Pik
  - Wochentage: Montag, . . . , Sonntag
  - Noten: Sehr gut,. . . , Ungenugend

- Java5+
- final-Konstanten vom Typ int

Konstanten und Aufzählungen

Möchte man mit einer festen Anzahl von Werten einer bestimmen Art arbeiten, so kann man diese im Prinzip einfach durchnummerieren.

```
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. Dies ist eine mögliche Fehlerquelle. Beispiel: MONDAY kann benutzt werden, wo eigentlich eine Jahreszahl erwartet würde.
- Das Hinzufügen oder das Löschen von Werten ist gefährlich. Beispiel: Ein E-Mail-Programm hat die Zahlen einer Aufzählung in eine Konfigurationsdatei geschrieben. Bei einem Update kam ein neuer Fall hinzu. Die Zahl, die vorher „behalte Mail“ bedeutete, wurde zu „lösche Mail“.
- 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, SATUDAY, 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 ubersetzt
- Enum-Typen konnen auch Methoden haben
- Konstanten konnen assoziierte Werte haben
- public enum A extends B { ... } nicht zulassig

Fallunterscheidung

- 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.

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 mit Javadoc

Dokumentation

- Wenn man nicht sagt, was ein Programm machen soll, ist die Wahrscheinlichkeit gering, dass es das macht.
- Dokumentation soll helfen,
  - Schnittstellen zu verstehen
  - Entwurfsideen zu erklären
  - implizite Annahmen (z.B. Klasseninvarianten) auszudrucken
- Nicht sinnvoll:
  - x++; // erhohe x um eins

Nicht sinnvoll

```
true ->
  case User.find(%{email: email}) do
    {:ok, [user | _]} ->
        # get a list of all unused tokens
        {:ok, tokens} = Token.find(%{user: user.id, type: token_t
        # some of them might be expired
        # => remove them (should be done on the DB!)
        # this seems rather pointless
        active_tokens = tokens
        |> Enum.filter(fn token ->
            token.expires_at > Timex.to_unix(Timex.now)
        end)

        # one token should be active at most
        token = active_tokens |> List.first

        cond do
          is_nil(token) ->
            #token is either expired or doesn`t exist
            # => generate new token
```

Was soll man dokumentieren?

- für jede Methode eine Zusammenfassung, was sie macht
- welche Werte zulässige Eingaben für eine Methode sind
- wie Methoden mit unzulässigen Eingaben umgehen
- wie Fehler behandelt und an den Aufrufer der Methode zurückgegeben werden
- Verträge (Vorbedingungen, Nachbedingungen, Klasseninvarianten)

Javadoc-Kommentare

- Javadoc erlaubt speziell ausgezeichnete Kommentare automatisch aus dem Code herauszuziehen und ubersichtlich darzustellen
- stehen vor Packages, Klassen, Methoden, Variablen
- spezielle Tags wie @see, @link

javadoc Package

javadoc Klasse1.java Klasse2.java ...

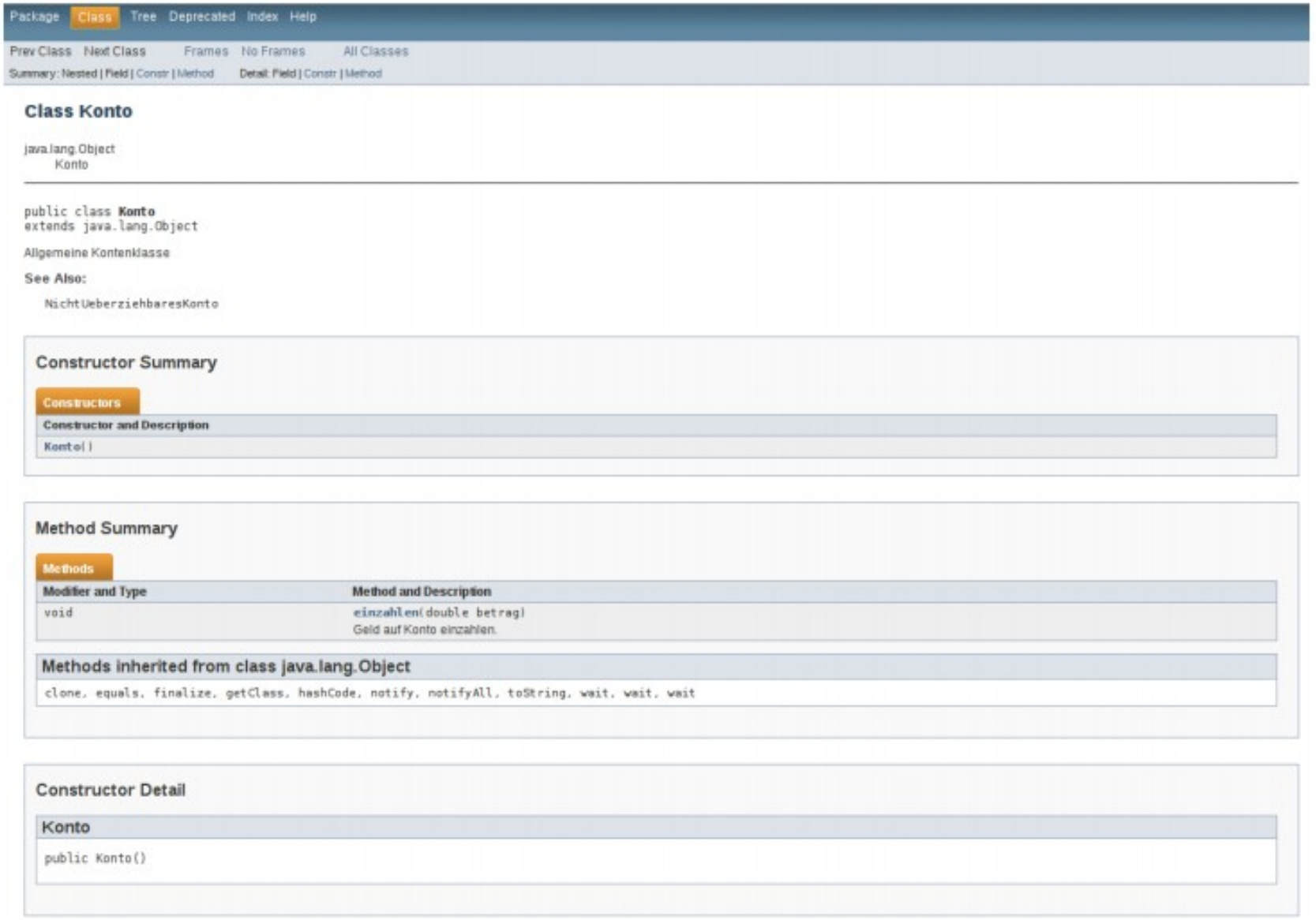
Javadoc-Tags

- Allgemein
  - @author Name
  - @version text
- Vor Methoden
  - @param Name Beschreibung – beschreibe einen Parameter einer Methode
  - @return Beschreibung – beschreibe den Rückgabewert einer Methode
  - @throws Exception Beschreibung – beschreibe eine Exception, die von einer Methode ausgelöst werden kann

Javadoc-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);
}
```

Javadoc-Beispiel: erzeugte html-Dokumentation



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

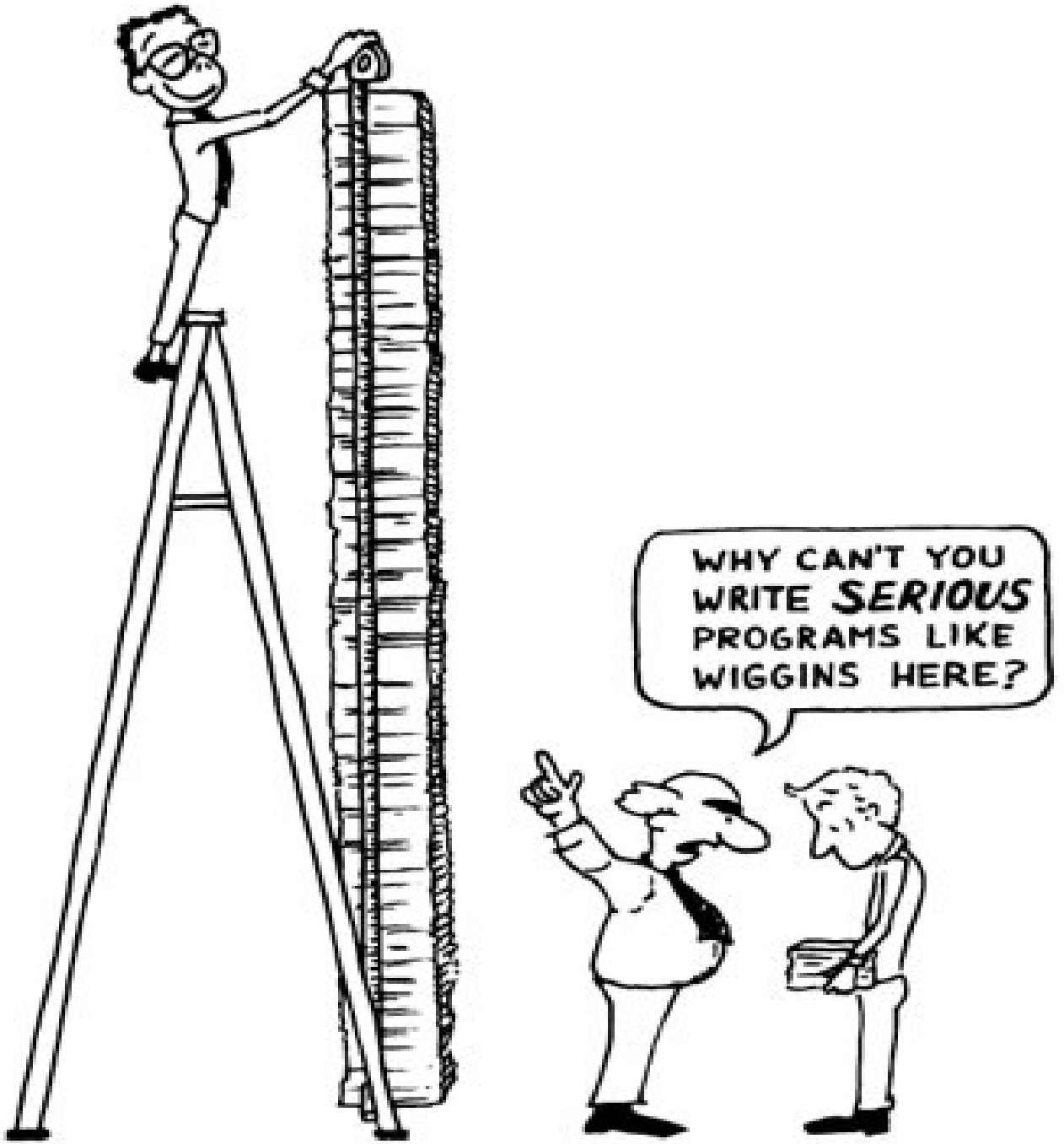
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Konto

public Konto()

Generics



Conventional wisdom reveres complexity.

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();
8
```

generische Klassen

```
1  [acces_mod] class NumeClasa <TipVar1[, TipVar2[, ...]] >{
2   private TipVar1 atribut1;
3   [declaratii atribute]
4   [declaratii si definitii metode]
5  }
6
7
8  public class Stiva<E>{
9   private class Nod<T>{
10  T info;
11  Nod<T> urm;
12  Nod(){info=null; urm=null;}
13  Nod(T info, Nod urm){
14 this.info=info;
15  this.urm=urm;
16  }
17  }
18  Nod<E> varf;
19  //...
20 }
```

Objekte erzeugen

```
1  public class Test{
2   public static void main(String[] args){
3   Stiva<String> ss=new Stiva<String>();
4   ss.pune("Ana");
5   ss.pune("Maria");
6   ss.pune(new Persoana("Ana", 23)); //error at compile-time
7   String elem=ss.scoate(); //NO CAST
8
9   Stiva<Persoana> sp=new Stiva<Persoana>();
10  sp.pune(new Persoana("Ana", 23));
11  sp.pune(new Persoana("Maria", 10));
12
13  Dictionar<String, String> dic=new Dictionar<String, String>();
14  dic.adauga("abc", "ABC");
15  dic.adauga(23, "acc"); //error at compile-time
16  dic.adauga("acc", 23); //error la compile-time
17   }
18 }
19
```

Objekte erzeugen

- unzulässig,Grunddatentypen nicht erlaubt als Typparameter:
  - Stiva<int> si=new Stiva<int>();
  - Stiva<Integer> si=new Stiva<Integer>();
- boolean -> Boolean
- byte -> Byte
- short -> Short
- int -> Integer
- ...

Autoboxing

- Java 1.5
- automatische Konversion zwischen prim. Typ und Klasse

```
1  Stiva<Integer> si=new Stiva<Integer>();
2  si.pune(23); //autoboxing
3  si.pune(new Integer(23));
4
5  int val=si.scoate();
6
7  Character ch = 'x';
8  char c = ch;
```

generische Methoden

- Deklaration einer generischen Methode:

```
1  class NumeClasa[<TypeVar ...>]{
2  [access_mod] <TypeVat1[, TypeVar2[,...]]> TypeR nameMethod([list_param]){
3   }
4   //...
5  }
6
7  public class MetodeGenerice {
8   public <T> void f(T x) {
9   System.out.println(x.toString());
10  }
11  public static <T> void copiaza(T[] elems, Stiva<T> st) {
12  for(T e:elems)
13  st.pune(e);
14  }
15 }
```

generische Methoden

- Methodenaufruf einer generischen Methode

```
1  public class A {
2   public <T> void print(T x) {
3   System.out.println(x);
4   }
5
6   public static void main(String[] args) {
7   A a=new A();
8   a.print(23);
9   a.print("ana");
10  a.print(new Persoana("ana",23));
11  }
12 }
```

generische Methoden

- Methodenaufruf einer generischen Methode

```
1  a.<Integer>print(3);
2  a.<Persoana>print(new Persoana("Ana",23));
3
4  NameClass.<Typ>nameMethod([parameters]);
5  //...
6  Integer[] ielem={2,3,4};
7  Stiva<Integer> st=new Stiva<Integer>();
8  MetodeGenerice.<Integer>copiaza(ielem, st);
9  //
10
11 this.<Typ>nameMethod([parameters]);
12 class A{
13   public <T> void print(T x){...}
14   public void g(Complex x){
15   this.<Complex>print(x);
16   }
17   }
18
```

generic Arrays

```
1  T[] elem=new T[dim]; //error at compile time
2
3  T[] elem=(T[])new Object[dim]; //warning at compile-time
4
5  import java.lang.reflect.Array;
6  public class Stiva <E>{
7   private E[] elems;
8   private int varf;
9   @SuppressWarnings("unchecked")
10  public Stiva(Class<E> tip) {
11  elems= (E[])Array.newInstance(tip, 10);
12  varf=0;
13  }
14  //...
15 }
16 Stiva<Integer> si=new Stiva<Integer>(Integer.class);
17
```

generic Arrays

```
1  public class Stiva <E>{
2    private Object[] elems;
3    private int varf;
4    public Stiva() {
5    elems=new Object[10];
6    varf=0;
7    }
8    public void pune(E elem){
9    elems[varf++]=elem;
10   }
11   @SuppressWarnings("unchecked")
12   public E scoate(){
13   if (varf>0)
14   return (E)elems[--varf];
15   return null;
16   }
17   //...
18 }
19
```

Erasure

- Beim compilieren von Programmen werden die Typparameter durch ihre oberen Schranken ersetz (zB Object)
- Interoperabilität

```
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();
  }
```

Bounds

```
1  public class ListaOrd<E> {
2   private class Nod<E>{
3   E info;
4   Nod<E> urm;
5   public Nod(){ info=null; urm=null; }
6   private Nod(E info, Nod<E> urm) { this.info = info; this.urm = urm; }
7   private Nod(E info) { this.info = info; urm=null; }
8   }
9   private Nod<E> cap;
10  public ListaOrd(){ cap=null;}
11  public void adauga(E elem){
12  if (cap==null){
13  cap=new Nod<E>(elem);
14  return;
15  }
16  if (/*compare elem to cap.info*/){
17  cap=new Nod<E>(elem,cap);
18  }else {...}
19  }
20 }
```

Bounds

```
1  public class ListaOrd {
2   private class Nod{
3   Object info;
4   Nod urm;
5   public Nod(){ info=null; urm=null; }
6   private Nod(Object info, Nod urm){this.info = info; this.urm = urm; }
7   private Nod(Object info) { this.info = info; urm=null; }
8   }
9   private Nod cap;
10  public ListaOrd(){ cap=null;}
11  public void adauga(Object elem){
12  if (cap==null){
13  cap=new Nod(elem);
14  return;
15  }
16  if (/*compare elem to cap.info*/){ //which methods can be called?
17  cap=new Nod(elem,cap);
18  }else {...}
19  }}
20
```

Bounds

- *<<Typvariable>> extends <<Typausdruck>>*
- nur solche Typparameter zuzulassen, die Erben von <<Typausdruck>> sind

```
1  public class Apple extends Fruit{}
2  public class Steak {}
3  public class FruitBox <T extends Fruit>{
4  << analog Box<T> >>
5  public static void main(String[] args) {
6  Apple a1 = new Apple();
7  Steak s1 = new Steak();
8  FruitBox<Apple> aBox = new FruitBox<Apple>(a1); //ok
9  FruitBox<Steak> sBox =
10 new FruitBox<Steak>(s1);//Compilezeitfehler:Bound Mismatch
11 }
12 }
13
```

Wildcards

- *public static void fruitBoxPrint(FruitBox<Fruit> f) { System.out.println(f); }*
- FruitBox<Apple> keine Unterklasse von FruitBox<Fruit> ist
- <<Klassenname>> <?>

```
1  public static void fruitBoxPrint( FruitBox<?> f){
2  System.out.println(f);}
3  public static void main(String[] args) {
4  FruitBox<Apple> aBox = new FruitBox<Apple>(new Apple);
5  fruitBoxPrint(aBox);
6
```

Bounded Wildcards

- *<<Klassenname>> <? extends <<Typausdruck>> >*
- beschränkt die Verwendung auf Subtypen von <<Typausdruck>>
- *<<Klassenname>> <? super <<Typausdruck>> >*
- beschränkt die Verwendung auf Obertypen von <<Typausdruck>>

```
1  public static void fruitBoxPrint1(Box<? extends Fruit> f){
2  System.out.println(f);}
3  public static void main(String[] args) {
4  Box<Apple> aBox = new Box<Apple>(new Apple);
5  fruitBoxPrint1(aBox);
6
```

Interfaces

Abstrakte Klasse

- Eine Klasse, in der einige Methoden nicht definiert werden
- diese Methoden heißen abstrakte Methoden
- müssen dann in jeder Unterklasse implementiert werden

Interface

- Eine spezielle Art von abstrakter Klassen, in der alle Methoden abstrakt sind
- Dienen zur Spezifikation einer Schnittstelle, die dann von erschiedenen 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
- Beim Implementieren der Klasse ist die spätere Verwendung nicht von Bedeutung, sondern nur die bereitzustellende Funktionalität
- Ein Anwender (eine andere Klasse) interessiert sich nicht für die Implementierungsdetails, sondern für die Funktionalität

Vorteile von Interfaces fur die Zusammenarbeit

- Ein Interface zu entwerfen geht schneller, als eine Klasse zu implementieren.
- Ist das Interface einer Klasse festgelegt, inklusive ausreichender Dokumentation, so kann man
  - Diese Klasse implementieren, ohne wissen zu müssen, wo genau und wie genau sie verwendet wird
  - Diese Klasse in weitergehender Implementierung verwenden, auch wenn sie noch nicht ausgeführt werden kann

Vorteile von Interfaces fur die Implementierung

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

Verwendungsbeispiel aus der Standardbibliothek

- LinkedList<E> ist eine Implementierung des Interfaces List<E>
- Verwendet man

```
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

Deklarationsbeispiel

`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);
}
```

Implementationsbeispiel

```
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;
  }
}
```

Mehrere Interfaces

- Eine Klasse kann mehrere Interfaces implementieren. Dazu deklarieren wir hier ein eigenes zweites Beispielinterface

```
public interface Growable {
    public void growBy(double x);
}
```

Mehrere Interfaces

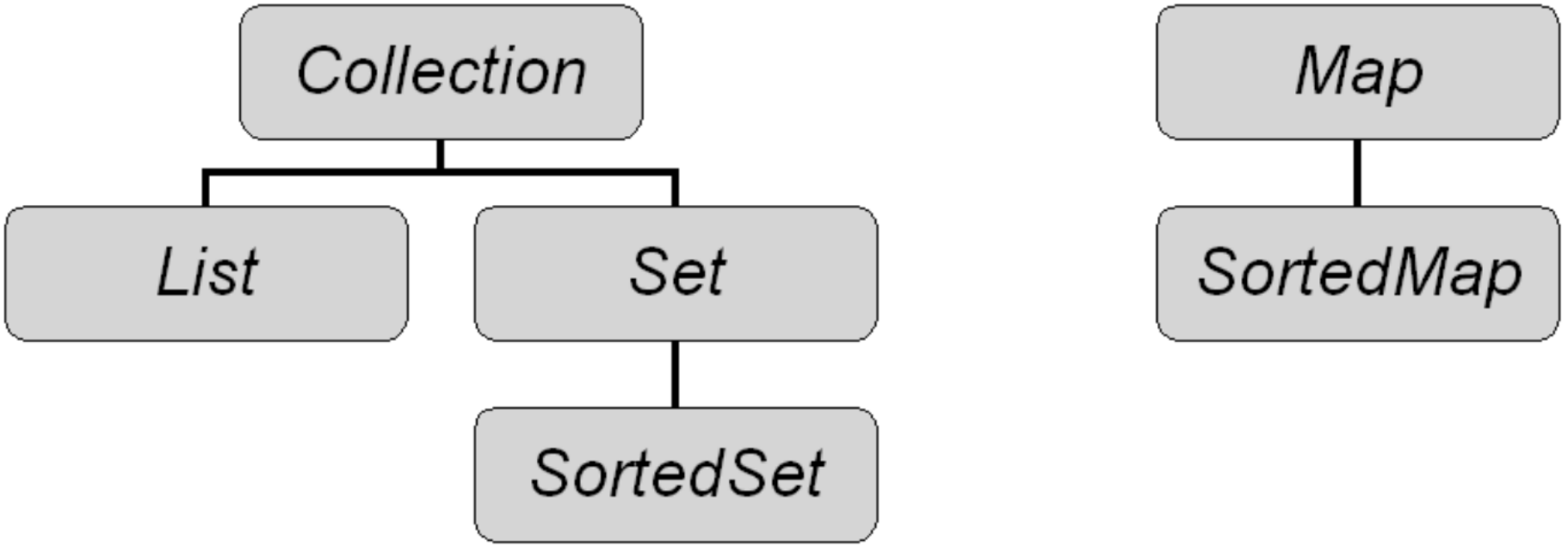
```
public class Person
  implements Comparable<Person>, Growable {
...Instanzvariablen und Konstruktor von vorhin...

public int compareTo(Person o) {
  if(size < o.size)          return  1;
    else if(size == o.size) return  0;
    else                      return -1;
  }

public void growBy(double x) {
  size = size + x;
  }
}

public static void main(String[] args) {
  Person hans = new Person(1.89, "Hans");
  Comparable<Person> hanscompare = hans;
  Growable hansgrow = hans;
  }
}
```

Methoden der Klasse Object

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

Das Java Collection Framework

eine Sammlung von Interfaces, die die Organisation von Objekten in “Container” 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 (Mengen)

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

Nützliche 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 des Collection Interfaces

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 des Collection Interfaces

Iterator iterator()

- return ein Iterator Objekt, mit dem man die Elemente einer Collection auf zählen 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 des Collection Interface

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 des Collection Interface

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()

- Retourniert die Elemente der Collection in einem Array

Object[] toArray(Object[] a)

- retourniert einen Array vom selbem (dynamischen) Typ wie a

Interface java.util.List

- Spezifiziert eine Collection, bei der die Elemente durchnumeriert 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

Die wichtigsten zusätzlichen Methoden

Object get(int i)

- retourniere das i-te Element

Object set(int i, Object o)

- weise dem i-ten Element das Objekt o zu

i*nt indexOf(Object o)*

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

Die wichtigsten zusätzlichen 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)

- retourniert die Teil-Liste beginnend mit von, endend mit bis-1

Vordefinierten 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

Vordefinierten 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.

Iterable

Konzept von Iterable

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

java.util.Iterator

Festgelegte Methoden:

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

Man kann auch kurz schreiben

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

Effizienter mittels Iterator

Wenn foo zur Klasse LinkedList<T> genugt, ist

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

erheblich schneller als

```
for(int i = 0; i < foo.size(); i++){
    T elem = foo.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

```
1  import java.util.*;
2  public class LottoZiehung
3  {
4   public static void main(String[] args)
5   {
6   HashSet zahlen = new HashSet();
7   //Lottozahlen erzeugen
8   while (zahlen.size() < 6) {
9   int num = (int) (Math.random() * 49) + 1;
10  if (zahlen.add(new Integer(num))) {
11  System.out.println("Neue Zahl " + num);
12  }
13  else {
14  System.out.println("DoppelteZahl " + num + " ignoriert");
15  }
16  }
17  //Lottozahlen ausgeben
18  Iterator it = zahlen.iterator();
19  while (it.hasNext()) {
20  System.out.println(
21 ((Integer) it.next()).toString());
22  }
23  }
24 }
25
```

java.util.Collections

- Eine Sammlung von statischen Methoden zum Arbeiten mit 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, nur daß die Elemente sortiert werden müssen
- also wieder nur ein semantischer Unterschied in der Implementierung des Interfaces
- Um sortieren zu können, muß ich Elemente vergleichen können

java.util.Comparator

- ein eigenes Objekt zum Durchführen von Vergleichen
- Einzige Methode, die implementiert werden muß
- *int compare(Object o1, Object o2)*
- Rückgabewert: 0,+,-

java.util.Comparable

- Interface, das angibt, daß auf Objekten, die dieses Interface implementieren, eine totale Ordnung definiert ist.
- int compareTo(Object o)
  - vergleicht dieses Objekt mit dem Objekt o
- Rückgabewert: 0, + , -

Sortierungsbeispiel

```
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);
    }
}
```

Sortierungsbeispiel

BOB ist 15 Jahre alt

LOB ist 19 Jahre alt

DOB ist 28 Jahre alt

ZOB ist 33 Jahre alt

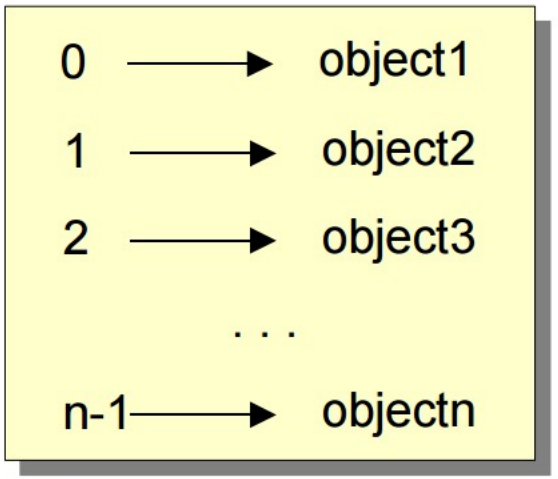
```
import java.util.*;
public class Main {
  public static void main(String[] args) {
    List<Person> leute = new ArrayList<Person>();
    leute.add(new Person(15, "BOB"));
    leute.add(new Person(28, "DOB"));
    leute.add(new Person(19, "LOB"));
    leute.add(new Person(33, "ZOB"));
    Collections.sort(leute);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

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

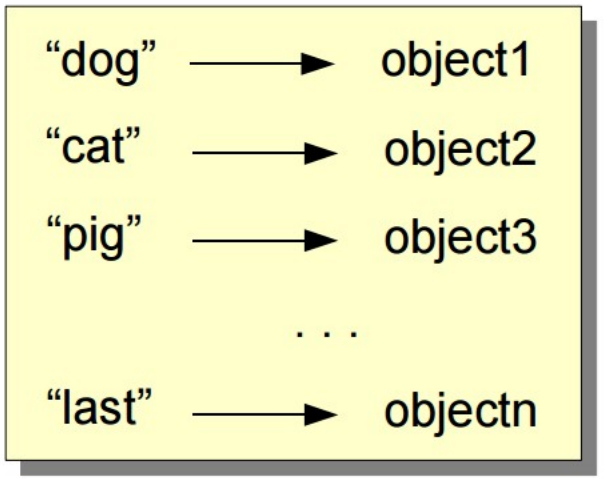
List (bzw. Array)



- Zugriff

```
l.set(2,object3)
Object o = l.get(2);
```

Map



- Zugriff

```
m.put(“pig”,object3)
Object o = m.get(“pig”);
```

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 (wär aber gut)

java.util.HashMap

```
1  import java.util.*;
2  public class MailAliases
3  {
4   public static void main(String[] args)
5   {
6   HashMap h = new HashMap();
7   //Pflege der Aliase
8   h.put("Fritz","f.mueller@test.de");
9   h.put("Franz","fk@b-blabla.com");
10  h.put("Paula","user0125@mail.uofm.edu");
11  h.put("Lissa","lb3@gateway.fhdto.northsurf.dk");
12  //Ausgabe
13  Iterator it = h.keySet().iterator();
14  while (it.hasNext()) {
15  String key = (String)it.next();
16  System.out.println(key + " --> "
17  + (String)h.get(key) );
18  }
19  }
20 }
21
```

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