Programmierungstechnik 2

Einführung in Java

Prof. Dr. Uwe Hartmann Fachhochschule Stralund

Gliederung

1 Einführung
1.1 Objektorientierte Softwareentwicklung
1.2 Eigenschaften von Java
1.3 Programmaufbau
1.4 Erstellen eines Java Programms mit Eclipse

2 Datentypen

3 Einfache Ein- und Ausgabe
3.1 Bildschirmausgabe
3.2 Tastatureingabe

4 Klassen und Klassenmember
4.1 Einführung UML: Klassen
4.2 Einkapselung (encapsulation)
4.3 Klassendeklaration
4.4 Felder
4.5 Methoden
4.5.1 Methodendeklaration
4.5.2 Parameter einer Methode
4.5.3 Rückgabewerte einer Methode
4.5.4 Überladen von Methoden
4.6 Konstruktoren
4.7 Eigenschaften (properties)
4.8 Statische Klassenmember
4.9 Generische Klassen und Methoden
4.9.1 Generische Klassen
4.9.2 Generische Methoden

5 Spezielle Referenztypen
5.1 Die Klasse Object
5.2 Wrapper-Klassen
5.3 Strings
5.3.1 Die Klasse String
5.3.2 Die Klassen StringBuffer und StringBuilder
5.4 Arrays
5.4.1 Array-Typ
5.4.2 Die Klasse Arrays
5.4.3 Sortieren von Arrays
5.5 Aufzählungen
5.6 Die Klasse Class

6 Klassen-Hierarchien
6.1 Vererbung
6.1.1 Ableitungsdeklaration
6.1.2 Konstruktoren
6.1.3 Basisklassen-Member
6.2 Polymorphie
6.3 Abstrakte Klassen und Methoden
6.4 Schnittstellen
6.4.1 Schnittstellendeklaration
6.4.2 Schnittstellenimplementierung

7 Klassenbeziehungen
7.1 Assoziationen
7.2 Iteratoren

8 Exceptions
8.1 Unbehandelte Ausnahmen
8.2 Behandelte Ausnahmen
8.3 Ausnahmen werfen
8.4 Eigene Ausnahmeklassen definieren

9 Streams
9.1 Stream-Klassen: Übersicht
9.2 Bytestream-Klassen
9.2.1 Outputstream-Klassen
9.2.2 Inputstream-Klassen
9.3 Characterstream-Klassen
9.3.1 Writer-Klassen
9.3.2 Reader-Klassen
9.4 Objektserialisierung

10 Collections
10.1 Überblick
10.2 Listen
10.3 Queues
10.4 Sets
10.5 Klasse Collections

11 Datenbankzugriff mit JDBC
11.1 Dateisysteme
11.2 Datenbanksysteme
11.3 SQL
11.4 JDBC
11.4.1 JDBC-Treiber
11.4.2 JDBC-API

Literatur

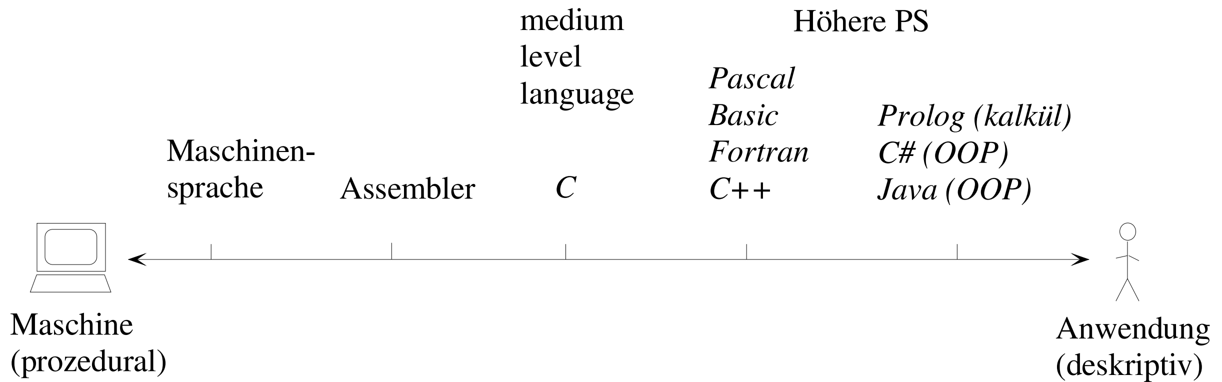
Mathias Hölzl
Java kompakt
Springer Verlag 2013

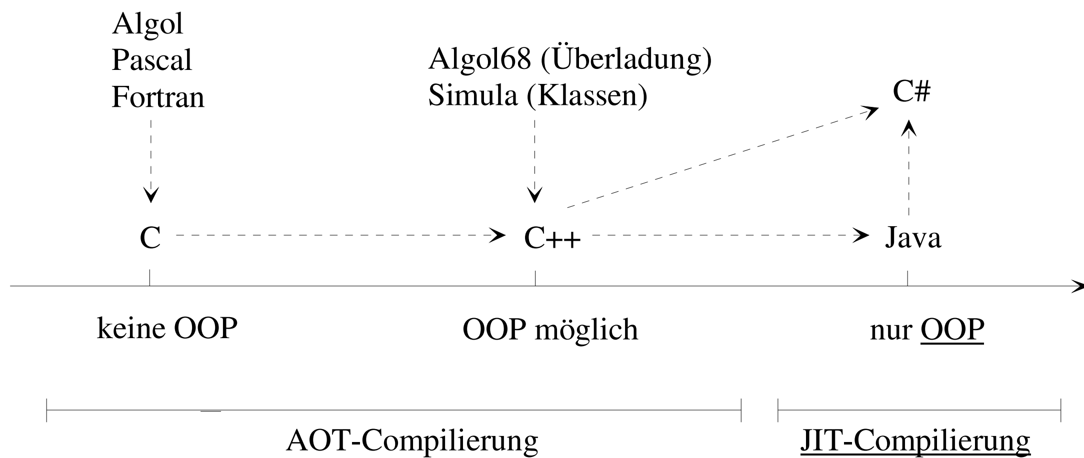
Dietmar Ratz, Jens Scheffler
Grundkurs Programmieren in Java
Carl Hanser Verlag 2011

Cornelia Heinisch
Java als erste Programmiersprache: Ein professioneller Einstieg in die Objektorientierung mit Java
Vieweg+Teubner Verlag 2011

Christian Ullenboom
Java ist auch eine Insel: Das umfassende Handbuch
Galileo Computing 2014

1 Einführung





1.1 Objektorientierte Softwareentwicklung

- **Imperative Programmierung (Softwareentwicklung)**
  Abläufe eines Anwendungsbereiches werden schrittweise in Algorithmen umgesetzt

- **Objektorientierte Programmierung (Softwareentwicklung)**
  Objekte eines Anwendungsbereiches, ihre Zustände, ihr Verhalten und ihre Beziehungen zueinander werden beschrieben und klassifiziert

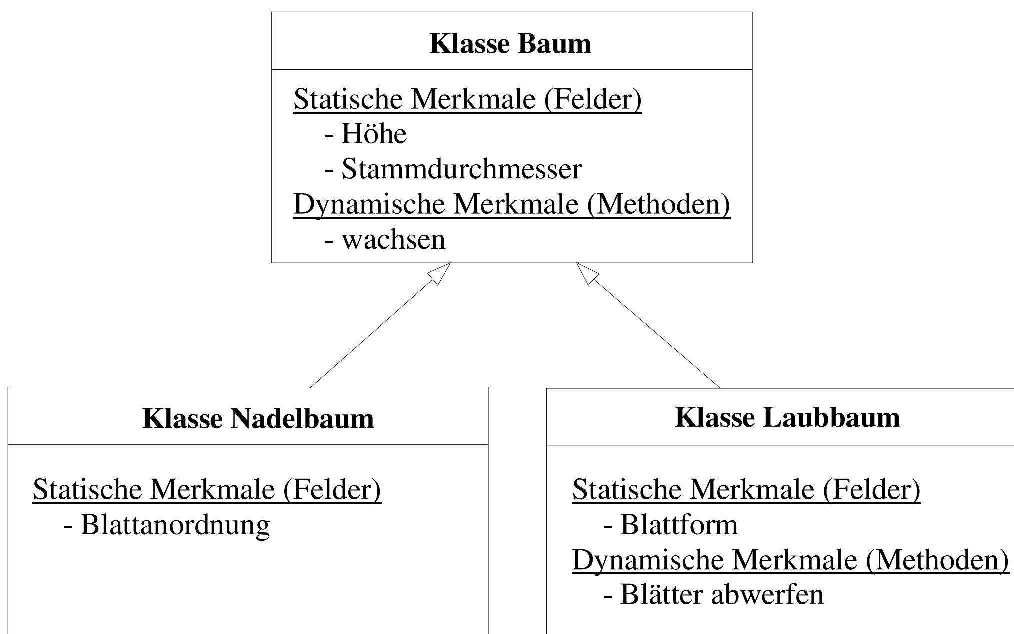
- **Merkmale von Objekten**
  – statische Merkmale: Zustände (--> Daten)
  – dynamische Merkmale: Verhalten (-->Algorithmen)

- **Klasse (extensional)**
  Zusammenfassung (Menge) von Objekten

- **Klasse (intensional) = abstrakter Datentyp**
  Beschreibung der Merkmale der Klassenobjekte durch
  - Datenvariablen (statische Merkmale, Zustände) und
  - Funktionen (dynamische Merkmale, Algorithmen)

Klasse
Eine Klasse ist ein Typ, der Felder und Methoden kapselt. Die Felder und Methoden werden in der Klasse definiert. Felder und Methoden werden unter dem Begriff Member zusammengefasst

Objekt
Ein Objekt ist eine Instanz einer Klasse. Es besitzt konkrete Werte für die Felder und kann die Methoden ausführen



Ableitung/Vererbung
Eine Klasse A kann von einer Klasse B abgeleitet werden. Diese Klasse B bezeichnet man als Basisklasse der abgeleiteten Klasse A. Die Klasse A erbt und besitzt dann alle Felder und Methoden der Basisklasse B. In A können weitere zusätzliche Felder und Methoden definiert werden.

Schnittstelle
Eine Schnittstelle ist ein Typ, in dem Methoden deklariert werden..

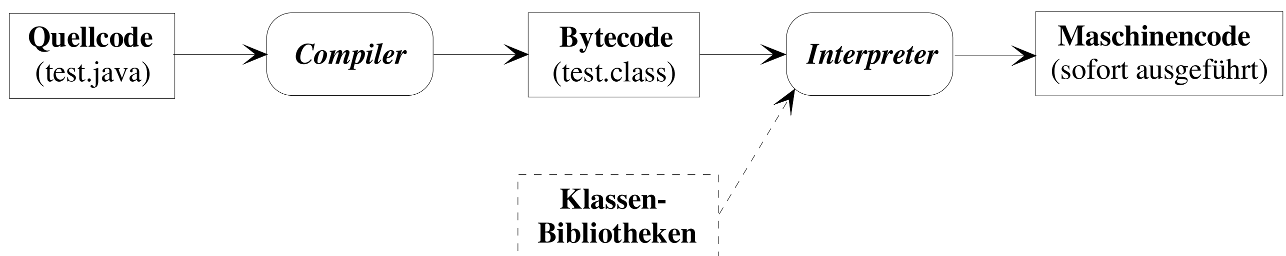
Implementation
Eine Klasse kann mehrere Schnittstellen implementieren. In der Klasse müssen dann alle in den Schnittstellen deklarierten Methoden auch definiert werden

- **Problemstellung bei der objektorientierten Softwareentwicklung**
  Definition/Finden geeigneter Klassen mit ihren Merkmalen und ihren Beziehungen zueinander zur Abbildung der relevanten Objekte eines Anwendungsbereiches in eine Anwendung

1.2 Eigenschaften von Java

Klassischer Übersetzungsprozeß

Java Übersetzungsprozeß



- **Quellcode**
  - Java-Quellprogramm: Menge von Klassen
  - Datei-Suffix: **.java**

- **Java-Compiler**
  - Übersetzt Quellcode in einen maschinenunabhängigen Bytecode.
  - Führt dabei lexikalische und syntaktische Analyse durch
  - Compiler: **javac.exe** aus JDK oder in **Eclipse** integrierter Compiler
  - Datei-Suffix: **.class**

- **Java-Interpreter**
  - sog. ***Java Virtual Machine (JVM)***
  - manchmal auch als auch als Just-In-Time-Compiler (JITter) oder Compreter bezeichnet
  - Übersetzt Bytecode zur Laufzeit in Maschinencode, der danach ausgeführt wird
  - Interpreter: **java.exe** aus JRE
  - JVM lässt sich durch ihren vergleichsweise einfachen Aufbau gut auf aktuelle Hardware-Architekturen abbilden

Software

- **Java Development Kit (JDK)**
  - enthält neben der Laufzeitumgebung Java Runtime Environment (JRE)
    - den Compiler **javac.exe**
    - das Java Dokumentationswerkzeug *javadoc*
    - einen Java Archiver *jar* und weitere Komponenten
  - wird von Oracle kostenlos vertrieben

- **Eclipse**
  - integrierte Entwicklungsumgebung (IDE) für Java zum Editieren und Compilieren von Quellcode (es existieren Plugins für andere Sprachen: C, C++, C#, ...)
  - ursprünglich von IBM entwickelt
  - derzeit als Open-Source von der Eclipse Foundation (Eclipse Community) weiterentwickelt und kostenlos bereitgestellt

- **Java Runtime Environment (JRE)**
  - enthält Virtuelle Maschine **java.exe** sowie
  - die Programmierschnittstelle, die die Standard-Klassenbibliothek mit Klassen für alle Routineaufgaben zur Verfügung stellt
  - wird von Oracle kostenlos vertrieben

- **Standard-Klassenbibliothek**
  - wird oft als **API** (***A**pplication **P**rogram **I**nterface*) bezeichnet
  - wichtigste Komponenten (Pakete):
    - java.lang

      Klassen, die für die grundlegendsten Mechanismen der Programmiersprache Java benötigt werden, z.B. Object, Wrapper, String, System
    - java.util

      verschiedene Klassen, die für praktisch jedes Java-Programm benötigt werden, z.B. Collection, Date, Properties
    - java.io

      Klassen für Ein- und Ausgaben, z.B. InputStream, OutputStream
  - nützliche Referenz auf Java Standard-Klassen: *http://www.dpunkt.de/java/Referenz/*

Unterschiede zwische Java und C

- **Portabilität**
  - auf der Basis des Bytecode können Programme auf unterschiedlichen Maschinen ausgeführt werden

- **Objektorientiertheit**
  - Funktonen können nicht außerkalb von Klassen definiert werden, sondern nur innerhalb als Methoden

- **Speicherverwaltung**
  - keine explizite dynamische Speicherverwaltung möglich
  - Referenzkonzept anstelle Zeigerkonzept
  - im Stack ausschließlich primitive Objekte, Klassenobjekte im Heap
  - Freigabe von Speicherplatz im Heap durch Garbage Collector

- **Fehlermanagement**
  - einheitliches Fehlerbehandlungskonzept über Exceptions

- **Sprachunterschiede**
  - erweiterte for-Anweisung (sog. foreach)
  - Vergleichsausdrücke explizit gefordert
  - neue Notation von Arrays
  - String zur Aufnahme von Zeichenketten

1.3 Programmaufbau

Beispiel (C/C++)

```
#include <stdio.h>
int main(int argc, char* argv[]){
  printf("Hello World");
  return 0;
}
```

Beispiel 1.3.a

```
1  package MyPackage;
2  import java.lang.System;
3  public class MyProgram {
4    public static void main(String[] args){
5      System.out.println("Hello world!");
6    }
7  }
```

- jedes ausführbare Programm besitzt genau eine Klasse mit einer Methode main() als Einstiegspunkt für die Programmausführung

Syntax

CompilitionUnit ::=

$$PackageDeclaration_{opt}\ \ ImportDeclarations_{opt}\ \ TypeDeclarations$$

- Ausführbare Einheit besteht aus 3 Bestandteilen:
  - einer optionalen Package-Deklaration

– einer optionalen Liste von Import-Deklarationen
– einer oder mehrerer Typ-Deklarationen (Klassen- oder Interface-Deklarationen)

Syntax

TypeDeclarations ::=

ClassDeclaration
| *InterfaceDeclaration*
| *TypeDeclaration TypeDeclaration*

Package

- Ein package (Paket) ist eine logische Organisationsstruktur, die verschiedene Typdefinitionen (insbes. Klassendeklarationen) zusammenfasst

Syntax

PackageDeclaration ::=

package *PackageIdentifier*

- Stilregel: Ein Typ (Klasse) sollte immer zu einem definierten packege gehören
- Pakete können selbst hierarchisch geordnet sein
- Zugriff auf einen Typ (Klasse) durch Punktnotation: *PackageIdentifier.TypeIdentifier* oder bei Pakethierarchien: *PackageIdentifier1.PacketIdentifier2. ... .TypeIdentifier*

Import

- Klassen (bzw. Interfaces) aus anderen Paketen können dem Programm durch Import-Deklaration bekanntgemacht werden.
- In dem Programm können dann diese Klassen und deren Inhalte verwendet werden

Syntax

ImportDeclaration ::=

import *PackageIdentifier.TypeIdentifier*
| **import** *PackageIdentifier.**
| *ImportDeclaration ImportDeclaration*

- Import einer einzelnen Klasse aus einem Paket (*PackageIdentifier.TypeIdentifier*) oder aller Klassen aus einem Paket (*PackageIdentifier.**)

1.4 Erstellen eines Java Programms mit Eclipse

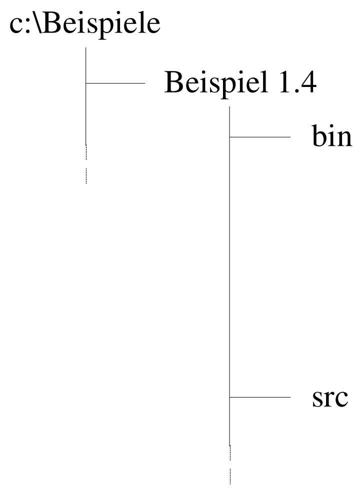
<u>Vorgehensweise</u>

– Erstellen eines Projektes (Beispiel 1.4)
– Deklaration zweier Pakete (KlassenPaket und ProgrammPaket)
– Deklaration zweier Klassen (Ausgabe und Programm)
– Ausführen (Compilieren)

Das Programm und alle erforderlichen Komponenten werden unter einem Verzeichnis c:\Beispiele abgelegt werden.

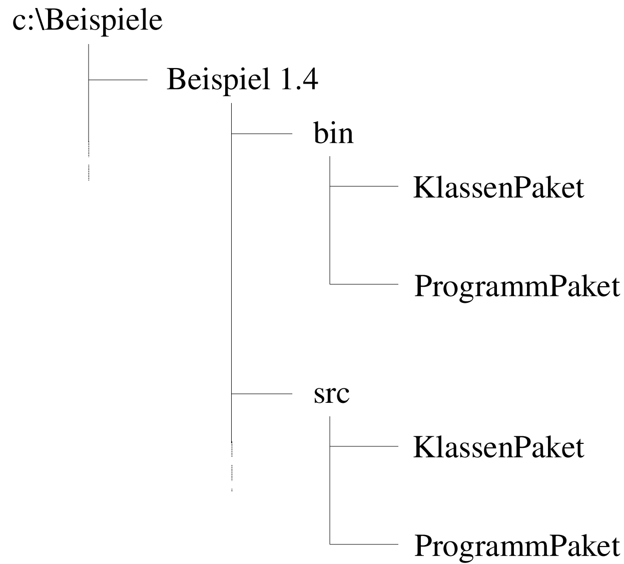
Erstellen des Projektes

- *Eclipse aufrufen --> <u>Workspace</u>:* **c:\Beispiele**
- *File --> New --> Java Project --> <u>Project name</u>:* **Beispiel 1.4**



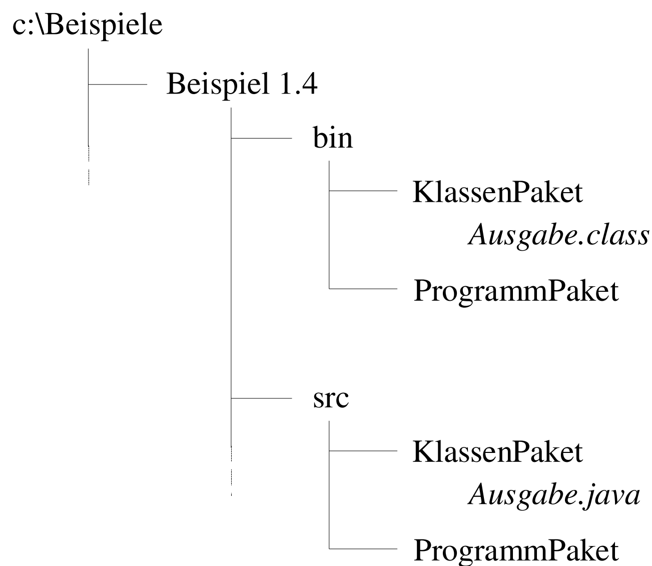
Erstellen der Pakete für das Projekt 'Beispiel 1.4'

- *File --> New --> Package --> <u>Name</u>:* **KlassenPaket**
- *File --> New --> Package --> <u>Name</u>:* **ProgrammPaket**



Erstellen der Klasse 'Ausgabe' im KlassenPaket

- *File --> New --> Class --> Name:* **Ausgabe**



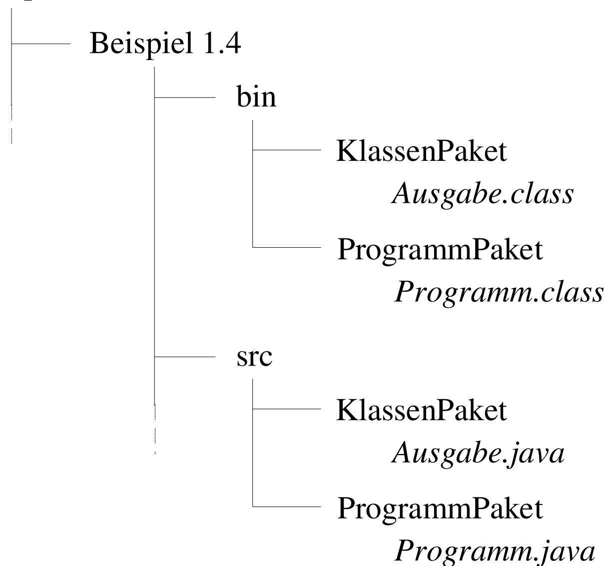
Ausgabe.java

```
1  package KlassenPaket;
2  public class Ausgabe {
3    public String ausgabeString;
4    public void ausgabe(){
5      System.out.println(ausgabeString);
6    }
7  }
```

- Zeilen 2..7: Klassendefinition. Ein Klasse besteht aus
  - Feldern: Zeile 3 und
  - Methoden: Zeilen 4..6

Erstellen der Klasse 'Programm' im ProgrammPaket

- *File --> New --> Class --> Name:* **Programm**



Programm.java

```
1  package ProgrammPaket;
2  import KlassenPaket.*;
3  public class Programm {
4    public static void main(String[] args) {
5      Ausgabe ausgabeObjekt = new Ausgabe();
6      ausgabeObjekt.ausgabeString = "Hello world!";
7      ausgabeObjekt.ausgabe();
8    }
9  }
```

- Zeilen 3..9: Klassendefinition. Die Klasse enthält als einziges Member die Methode main, die ausgeführt werden kann
- in main wird in Zeile 5 ein Objekt der Klasse Ausgabe aus dem KlassenPaket mit dem Namen ausgabeObjekt erzeugt. Da diese Klasse im ProgrammPaket nicht bekannt ist, muß sie aus dem KlassenPaket importiert werden (Zeile 2). Das ausgabeObjekt besitzt damit ein Feld ausgabeString und eine Methode ausgabe(), die in der Klasse Ausgabe definiert wurden
- In Zeile 6 wird dem Feld ausgabeString dieses Objektes der String "Hello World!" zugewiesen
- In Zeile 7 wird die Methode ausgabe() dieses Objektes aufgerufen und somit der Inhalt von ausgabeString auf der Console ausgegeben (siehe Ausgabe.java Zeilen 5..7)

Ausführen des Programms unter Eclipse

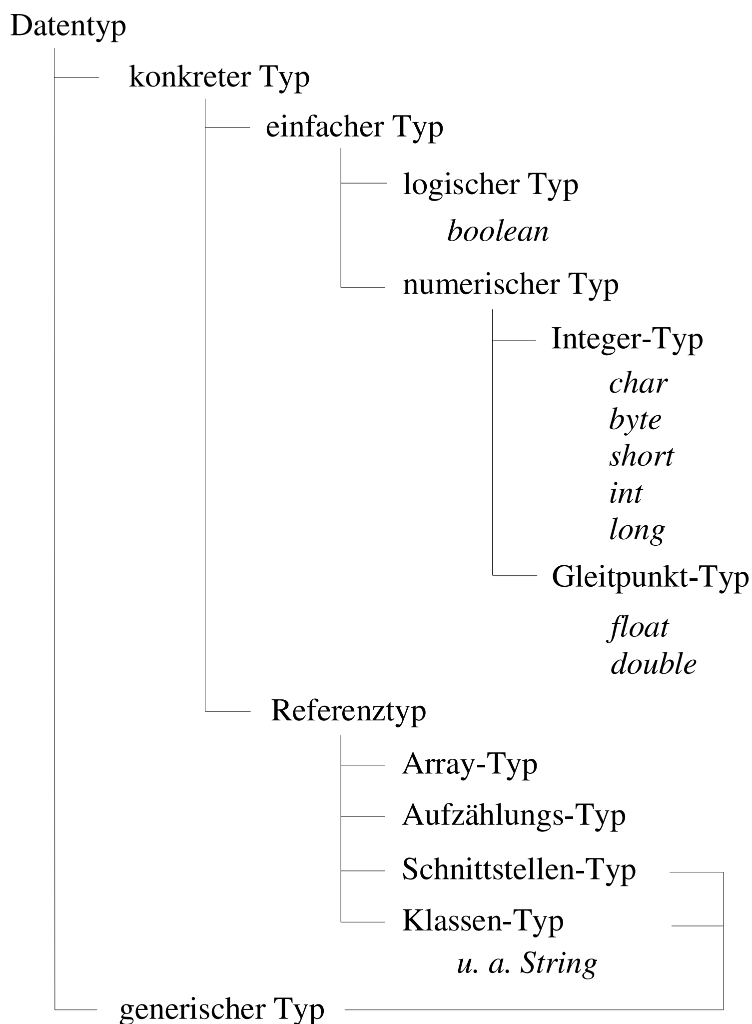
- *Run --> Run*

Ausführen des compilierten Programms (Bytecode) im DOS-Fenster (cmd.exe)

- cmd.exe aufrufen und in das Verzeichnis 'Beispiele\'Beispiel 1.4\bin' gehen

- Aufruf der JVM:
  c:\Beispiele\'Beispiel 1.4\bin> java ProgrammPaket**.**Programm

2 Datentypen



Einfache Datentypen

- sind Sprachbestandteil von Java

- **ganzzahlige Typen**
  - byte, short, int, long, char

- **Gleitpunkttypen**
  - float, double

- **logischer Typ**
  - boolean

Typ	Inhalt	Wertebereich
boolean	true oder false	True und false
char	16 Bit Unicode Zeichen	0x0000 ... 0xFFFF
byte	8 Bit Ganzzahl	-128 ... 127
short	16 Bit Ganzzahl	-32 768 ... 32 767
int	32 Bit Ganzzahl	–2.147.483.648 ... 2.147.483.647
long	64 Bit Ganzzahl	–9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807
float	Gleitpunktzahl	$-3.4 * 10^{38}$ ... $3.4 * 10^{38}$
double	Gleitpunktzahl	$-1.7 * 10^{308}$ ... $1.7 * 10^{308}$

- Deklaration von Variablen von einfachem Datentyp:

  DataType Variable;

  und Wertezuweisung

  Variable = *Value*;

Referenztypen

- sind kein Sprachbestandteil von Java

- **Referenztypen**
  - Klassen-Typ (besondere Klasse: String)
  - Schnittstellen-Typ
  - Array-Typ
  - Aufzählung-Typ

- alle Klassentypen sind von der Klasse **Object** abgeleitet
  --> jede Klasse ist auch vom Typ Objekt

- Deklaration von Referenzvariablen

  ClassName Variable;

  und Erzeugung (Instanziierung) eines Objektes mit **new** und Zuweisung der Referenz auf dieses Objekt an die Referenzvariable

  Variable **= new** *ClassName*();

Speichermanagement

- Objekte vom einfachen Typs werden im Stack oder Heap abgelegt
- Objekte vom Referenztyp werden im Heap abgelegt, die Referenzen selbst im Stack oder Heap

Beispiel 2a

```
public class MyClass {
   public int x;
   public int y;
   public String s = "World";
}
public class Programm {
   public static void main(String[] args){
      int i = 3;
      String s = "Hello";
      Object o1 = 3.14;
      Object o2 = true;
      MyClass c1 = new MyClass();
      MyClass c2 = c1;
   }
}
```

- eine Referenz ist ein Zeiger auf Speicherplatz im Heap, der mit **new** allokiert wurde
- Im Unterschied zu C/C++ gibt es in Java
  - keine zur Zeigerarithmetik äquivalente "Referenzarithmetik"
  - keine zu **delete** äquivalente Möglichkeit des expliziten Freigebens von Speicherplatz

Garbage Collector

- **Garbage Collector** (GC): Prozess, der im Hintergrund einer Anwendung abläuft mit der Aufgabe: Suche nach nichtreferenzierten Objekten im Heap und Freigabe des Speicherplatzes
- wird ausgeführt, wenn

  - Anwendung zeitweilig keine Prozessorleistung in Anspruch nimmt oder
  - Speicherrecourcen für die Anwendung einen Minimalwert unterschreiten
- Voraussetzungen für das Freigeben von Speicher für ein Objekt durch Garbage Collector
  - Gültigkeitsbereich einer Objektreferenz wird verlassen (Block)
  - Nullzuweisung: *objekt-variable* **= null;**
- Der Garbage Collector kann auch explizit im Programm durch

```
System.gc();
```

  aufgerufen werden.

3 Einfache Ein- und Ausgabe

3.1 Bildschirmausgabe

- Methoden **print(argument)** und **println(argument)** der Klasse java.io.PrintStream geben ihr Argument ohne bzw. mit anschließendem Zeilenvorschub auf den Bildschirm aus
- argument vom Typ: boolean, char, int, long, float, double, Object und String
- println() ohne Argument gibt nur Zeilenvorschub aus

Beispiel 3.1a

```
public class Programm {
  public static void main(String[] args){
    boolean b = true;
    System.out.println(b);      // true

    double d = 3.14;
    System.out.println(d);      // 3.14

    String s = "Hello ";
    System.out.print(s);
    System.out.println("World");// Hello World
  }
}
```

- Seit Version 5 Methode **printf(formatzeichenkette, argumentliste)** zur formatierten Ausgabe

Beispiel 3.1b

```
public class Programm {
   public static void main(String[] args){
      double pi = 3.14;
      double e = 2.718;
      System.out.printf("Wert1 = %f \nWert2 = %f", pi, e);  // Wert1 = 3,140000
                                                             // Wert2 = 2,718000
   }
}
```

3.2 Tastatureingabe

- Tastatureingabe erfolgt auf der Basis von Streams

Beispiel 3.2a

```
import java.io.*;

public class Programm {
   public static void main(String[] args) throws IOException {
      String eingabe;
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      eingabe = br.readLine();                      // <--Hallo
      System.out.print("Eingegeben wurde: ");
      System.out.println(eingabe);                  // Eingegeben wurde: Hallo
   }
}
```

- Seit Version 5 vereinfachte Eingabe durch Methoden der Klasse java.util.Scanner
- **Methoden** der Klasse Scanner (Auswahl):

next()	liest einen String
nextByte()	liest ein Byte
nextInt()	liest eine ganze Zahl
nextDouble()	liest einen Double-Wert
...	
close()	schließt einen Scanner und gibt Recourcen frei

Beispiel 3.2b

```
import java.util.Scanner;

public class Programm {
   public static void main(String[] args) {
      String s;
      int i;
```

```
        double d;
        Scanner sc = new Scanner(System.in);
        s = sc.next();                          // <-- Hallo
        System.out.println("EINGABE: " + s);    // EINGABE: Hallo

        i = sc.nextInt();                       // <-- 3
        System.out.println("EINGABE: " + i);    // EINGABE: 3

        d = sc.nextDouble();                    // <-- 3,14 (ACHTUNG: Komma)
        System.out.println("EINGABE: " + d);    // EINGABE: 3.14

        sc.close();
    }
}
```

4 Klassen und Klassenmember

- allgemeine Merkmale von Objekten
  - Zustände
  - Fähigkeiten, zu agieren
  - Fähigkeit, zu reagieren

- Eine **Klasse** ist ein Datentyp (Datenstruktur), in der Zustände als Felder und Fähigkeiten als Methoden gekapselt werden.

- Die Felder und Methoden einer Klasse bezeichnet man als **Member**

- Ein **Objekt** ist eine dynamisch erstellte Instanz einer Klasse

4.1 Einführung UML: Klassen

- Softwareentwicklungsprozeß
  - Anforderungsanalyse
  - Entwuf
  - Implementierung

Unified Modelling Language

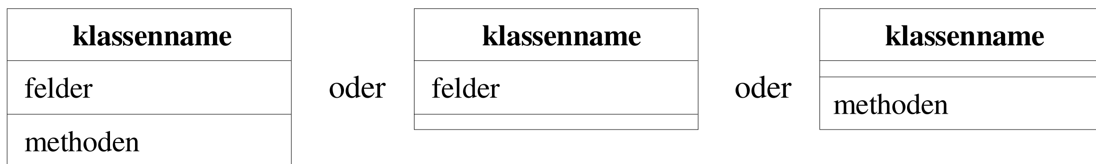
- **UML**
  - Standardisierte Sprache zur Modellierung von Software- und anderen Systemen
  - Definiert die dafür erforderlichen Begriffe und Begiffsbeziehungen
- entwickelt von der Object Management Group (OMG), Standard seit 1998

- Modellelemente der UML
  - Anwendungsfalldiagramm
  - **Klassendiagramm**
  - Aktivitätsdiagramm
  - Kollaborationsdiagramm
  - Sequenzdiagramm
  - Zustandsdiagramm
  - Komponentendiagramm
  - Einsatzdiagramm

- **Klassendiagramm**
  Grafische Darstellung von Klassen und deren Beziehungen untereinander zur (statischen) Modellierung objektorientierter Softwaresysteme

UML: Klassen

- Klassen bestehen aus **Membern**:
  - **Felder** (auch Datenmember, Eigenschaften, Attribute, Datenelemente, Instanzvariablen ...)
  - **Methoden** (auch Operationen, Prozeduren, Routinen, Funktionen, Services, ...)



- **klassenname**
  - Name der Klasse im Singular
  - Darstellung in ***PascalConvention***: beginnt mit Großbuchstaben

- **felder**
  - Folge von Feldbeschreibungen der Form

    modifizierer *feldname* **:** *typ* **=** *initialwert* **{** *eigenschaft* **}**

  - *feldname* in ***camelConvention***: beginnt mit Kleinbuchstaben
  - mögliche *modifizierer*:

+	für public
-	für private
#	für protected
/	für abgeleitetes (berechnetes) Feld

  - *eigenschaft* ist zusätzliche Eigenschaft, z.B. {ReadOnly}
  - alle Bestandteile außer *feldname* sind optional

- **methoden**
  - Folge von Methodenbeschreibungen der Form

    modifizierer *methodenname* **(** *argumentliste* **)** **:** rückgabetyp **{** *eigenschaft* **}**

  - *modifizierer* wie bei Feldern außer /
  - *argumentliste* ist kommagetrennte Folge von Argumentbeschreibungen:

    richtung *argumentname* **:** *typ* **=** *initialwert*

  - *richtung*: **in-**, **out-** oder **inout**-Parameter
  - *methodenname* in ***camelConvention***: beginnt mit Kleinbuchstaben
  - alle Bestandteile außer *methodenname* sind optional

- Notizen
  - Anmerkungen zu Klassen, Feldern oder Methoden
  - als Rechteck mit Eselsohr dargestellt

Beispiel

Konto
- kontonr : int {ReadOnly} - inhaber : String - kontostand : float = 0,0
+ inhaberÄndern (inhaber : string) : void + einzahlung (betrag : float) : void + auszahlung (betrag : float) : void + kontostand ( ) : float + kontendaten (out kontonr : int, out inhaber : String): void

Kontoinhaber soll
volljährig sein

4.2 Einkapselung (encapsulation)

Beispiel

```
int main(){
   int zaehler1 = 1;
   int nenner1 = 2;
   int zaehler2 = 2;
   int nenner2 = 3;
   int zaehler3;
   int nenner3;
   zaehler3 = zaehler1 * nenner2 + zaehler2 * nenner1;
   nenner3 = nenner1 * nenner2;
   printf("%d/%d\n", zaehler3, nenner3);
}
```

Beispiel

```
struct Bruch{
   int zaehler;
   int nenner;
};
Bruch Init(int z, int n){
   Bruch b;
   b.zaehler = z;
   b.nenner = n;
   return b;
}
Bruch Addiere(Bruch b1, Bruch b2){
   Bruch b;
   b.zaehler = b1.zaehler * b2.nenner + b2.zaehler * b1.nenner;
   b.nenner = b1.nenner * b2.nenner;
   return b;
}
int main(){
   Bruch bruch1 = Init(1,2);
   Bruch bruch2 = Init(2,3);
   Bruch bruch3 = Addiere(bruch1, bruch2);
   printf("%d/%d\n", bruch3.zaehler, bruch3.nenner);
}
```

Kapselung: Verfahren, bei dem **Daten** und **Funktionalität** von Objekten zu einem Typ zusammengefaßt werden.

komplexer Datentyp (struct) $\Rightarrow$ abstrakter Datentyp (class)

Ein **abstrakter Datentyp** ist eine Struktur, deren Komponenten

- Daten und
- Funktionen, die auf die Datenkomponenten zugreifen

sind. Gekapselte heißen **Felder**, gekapselte Funktionen heißen **Methoden**

Beispiel

```
struct Bruch{
   int zaehler;
   int nenner;
   Bruch Init(int z, int n){
      Bruch b;
      b.zaehler = z;
      b.nenner = n;
      return b;
   }
   Bruch Addiere(Bruch b1, Bruch b2){
      Bruch b;
      b.zaehler = b1.zaehler * b2.nenner + b2.zaehler * b1.nenner;
      b.nenner = b1.nenner * b2.nenner;
      return b;
   }
};
```

Beispiel 4.2a

```
public class Bruch {
   public int zaehler;
   public int nenner;
   public void init(int z, int n){
      zaehler = z;
      nenner = n;
   }
   public static Bruch addiere(Bruch b1, Bruch b2){
      Bruch b = new Bruch();
      b.zaehler = b1.zaehler * b2.nenner + b2.zaehler * b1.nenner;
      b.nenner = b1.nenner * b2.nenner;
      return b;
   }
}
public class Programm {
   public static void main(String[] args) {
      Bruch bruch1 = new Bruch();
      Bruch bruch2 = new Bruch();
      Bruch bruch3 = new Bruch();

      bruch1.init(1,2);
      bruch2.init(2,3);
      bruch3 = Bruch.addiere(bruch1, bruch2);
      System.out.println(bruch3.zaehler+"/"+bruch3.nenner);    // 7/6

   }
}
```

4.3 Klassendeklaration

Syntax

*ClassDeclaration** ::=

ClassModifiers$_{opt}$ **class** *Identifier* { *ClassBodyDeclarations*$_{opt}$ }

* wird später noch vervollständigt

Syntax

ClassModifiers ::=

public | **protected** | **private**
| **abstract**
| **final**
| *ClassModifiers ClassModifiers*

- Zugriffsmodifizierer public, protected und private beschränken die Sichtbarkeit und damit den Zugriff auf Klassen, Felder und Methoden

- **public**
  keine Zugriffsbeschränkung

- **private** und **protected**
  nur für innere Klassen

- **kein Zugriffsmodifizierer**
  Zugriff auf Klasse nur innerhalb des Pakets

- **final**
  abgeschlossene Klasse, von der keine weitere Klasse abgeleitet werden kann

- **abstract**
  abstrakte Klasse, von der keine Objekte instanziiert werden könen

- die Zugriffsmodifizierer public, protected und private dürfen in einer Klassendeklaration nur einmal auftreten

Klassenmember

Syntax

ClassBodyDeclarations ::=
FieldDeclaration
| ConstructorDeclaration
| *MethodDeclaration*
| *ClassDeclaration*
| *InterfaceDeclaration*
| *ClassBodyDeclarations ClassBodyDeclarations*

- **Felder**
  implementieren den statischen Zustand eines Klassenobjektes

- **Methoden**
  implementieren die Funktionalität von Klassenobjekten

- **Konstruktore**
  werden beim Erzeugen (Instanziieren) Klassenobjekten als erstes aufgerufen

- **Klassen- und Interfacedeklarationen**
  Typdeklarationen innerhalb von Klassen

Referenzen

- Klassenobjekte werden generell durch Aufruf eines Konstruktors mit dem new-Operator erzeugt (instanziiert) und die Referenz auf den Speicher einer Variablen zugewiesen

  class-name variable;
  variable = **new** *class-name*();

  oder

  class-name variable = **new** *class-name*();

- *variable* referenziert das Klassenobjekt im Heap und wird auch als **Instanzvariable** oder **Objektvariable** bezeichnet

- jede deklarierte Klasse erhält vom System einen Standardkonstruktor *class-name*(), der nach Aufruf alle Klassenfelder initialisiert

- der Zugriff auf Klassenmember von außerhalb einer Klasse erfolgt durch Punktnotation

variable.member

4.4 Felder

- Felder sind Variablen- oder Konstantendeklarationen innerhalb von Klassen

Syntax

FieldDeclaration ::=

  $FieldModifiers_{opt}$ *VariableDeclaration*
| $FieldModifiers_{opt}$ *ArrayDeclaration*

Syntax

FieldModifiers ::=

  public | **protected** | **private**
| **static**

- **static**
  Statische Felder sind nicht Teil einer bestimmten Instanz, sondern existieren für eine Klasse nur einmal

Syntax

VariableDeclaration ::=

  Type VariableDeclarator

VariableDeclarator ::=

  Identifier $Initializer_{opt}$
| *VariableDeclarator* **,** *VariableDeclarator*

Initializer ::=

  = *Expression*

Beispiel 4.4a

```
public class ValueSet {
   public int count;
   public int[] values = new int[100];
}

public class Programm {
   public static void main(String[] args){
      ValueSet vs = new ValueSet();
      vs.values[0] = 33;
      vs.count++;
      vs.values[1] = -6;
      vs.count++;
      for(int i=0; i<vs.count; i++)
         System.out.println(vs.values[i]);   // 33  -6
   }
}
```

4.5 Methoden

- Methoden implementieren die Funktionalität von Klassen bzw. Objekten

4.5.1 Methodendeklaration

Syntax

MethodDeclaration ::=

MethodHeader MethodBlock

*MethodHeader** *::=*

MethodModifiers$_{opt}$ *ResultType Identifier* (*FormalParameterList*$_{opt}$)

MethodBlock ::=

{ StatementsOrDeclarations$_{opt}$ }

* wird später noch vervollständigt

- **StatementsOrDeclarations**
  - Steueranweisungen oder
  - (lokale) Variablendeklarationen oder
  - Klassen- und Interfacedeklarationen

Syntax

MethodModifiers ::=

public | **protected** | **private**
| **abstract**
| **static**
| *MethodModifiers MethodModifiers*

- **abstract**
  - abstrakte Methode, darf keinen Methodenblock enthalten
  - darf nicht mit den Modifizierer static und private zusammen auftreten
  - darf nur in abstrakten Klassen definiert werden

- **static**
  - statische Methode, gehört nicht zu einem Klassenobjekt, sondern zur Klasse als Ganze
  - darf nicht mit den Modifizierer static und private zusammen auftreten

Beispiel 4.5a

```
public class ValueSet {
   public int count;
   public int[] values = new int[100];
   public void printValues()
   {
      for (int i = 0; i < count; i++)
         System.out.println(values[i]);
      System.out.println();
   }
}

public class Programm {
   public static void main(String[] args){
      ValueSet vs = new ValueSet();
      vs.values[0] = 33;
      vs.count++;
      vs.values[1] = -6;
      vs.count++;
      vs.printValues();     // 33  -6
   }
}
```

4.5.2 Parameter einer Methode

Syntax

FormalParameterList ::=

  Type Identifier
| *Type* **. . .** *Identifier*
| *FormalParameterList* **,** *FormalParameterList*

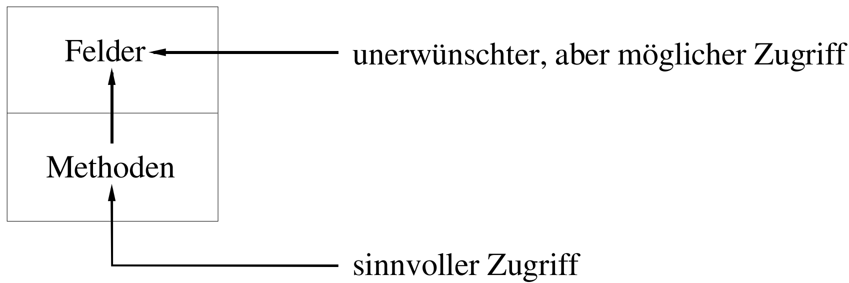
- **Type**
  einfacher oder komplexer Datentyp oder **void**

- **Type . . .**
  – (beliebig langes) Array von Parametern eines Typs (Parameterliste)
  – darf nur einmal und auch nur als letzter Parameter der Liste auftreten

Beispiel 4.5b

```
public class ValueSet {
  public int count;
  public int[] values = new int[100];
  public void printValues() {..}
  public boolean addValue(int value){
    for (int i = 0; i < count; i++)
      if(values[i]==value) return false;
    values[count] = value;
    count++;
    return true;
  }
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet();
    vs.addValue(33);
    vs.addValue(-6);
    vs.printValues();      // 33  -6
  }
}
```

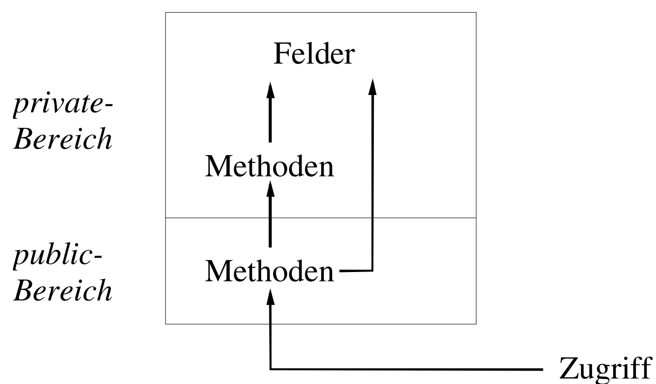
```
public class Programm {
   public static void main(String[] args){
      ValueSet vs = new ValueSet();
      vs.addValue(33);
      vs.addValue(-6);
-->   vs.values[vs.count++] = 33;
      vs.printValues();    // 33  -6  33
   }
}
```



Beispiel

```
public class ValueSet {
   private int count;
   private int[] values = new int[100];
   public void printValues() {..}
   public boolean addValue(int value){..}
}
```

- eine korrekte Klassendefinition verbirgt ihre Felder nach außen



Parameterlisten

- mit

  Type . . . Identifier

  als letztem Parameter einer bietet Java die Möglichkeit, einer Methode beliebig vielen aktuellen Parametern zu übergeben

Beispiel 4.5f

```
public class ValueSet {
   private int count;
   private int[] values = new int[100];
   public void printValues() {..}
   private boolean addValue(int value){..}
   public void addValues(int... values){
      for(int value : values)
         addValue(value);
   }
}
public class Programm {
   public static void main(String[] args){
      ValueSet vs = new ValueSet();
      vs.addValues(33,-6,12,33);
      vs.printValues();    // 33  -6  12
   }
}
```

Werteparameter vs. Referenzparameter

- Java unterscheidet nicht zwischen Werte- und Referenzparametern
- Parameterübergabe ausschließlich durch call-by-value:
- **call-by-value**
  bei Methodenaufruf wird dieser eine Kopie der Werte der formalen Parameter an die aktuellen Parameter übergeben

Beispiel 4.5d

```
public class Programm {
   public static void werteParameterTest(int i){
      i = 6;
      System.out.println(i);
   }
   public static void main(String[] args){
      int i = 3;
      System.out.println(i);    // 3
      werteParameterTest(i);    // 6
```

```
      System.out.println(i);    // 3
   }
}
```

Beispiel 4.5e

```
public class MyClass {
   public int i = 3;
}
public class Programm {
   public static void referenzParameterTest(MyClass mc){
      mc.i = 6;
      System.out.println(mc.i);
   }
   public static void main(String[] args){
      MyClass mc = new MyClass();
      System.out.println(mc.i);        // 3
      referenzParameterTest(mc);       // 6
      System.out.println(mc.i);        // 6
   }
}
```

Beispiel 4.5g

```
public class ValueSet {
   private int count;
   private int[] values = new int[100];
   public void printValues() {..}
   private boolean addValue(int value){..  }
   public void addValues(int... values){..}
   public void union(ValueSet v){
      for(int i=0; i<v.count; i++)
         addValue(v.values[i]);
   }
}
public class Programm {
   public static void main(String[] args){
      ValueSet vs1 = new ValueSet();
      ValueSet vs2 = new ValueSet();
      vs1.addValues(33,-6,12);
      vs2.addValues(27,33,51,2);
      vs1.union(vs2);
      vs1.printValues();   // 33 -6 12 27 51 2
   }
}
```

Beispiel 4.5h(fehlerhaft)

```
public class ValueSet {
  private int count;
  private int[] values = new int[100];
  public void printValues(){..}
  private boolean addValue(int value){..}
  public void addValues(int... values){..}
  public boolean isIn(int[] values){
    boolean isIn;
    for(int i=0; i<values.length; i++){
      isIn = false;
      for(int j=0; j<count; j++)
-->     if(values[i]==values[j]) isIn = true;
      if(isIn==false) return false;
    }
    return true;
  }
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet();
    vs.addValues(33,-6,12,22,5);
    int[] values = {12,22};
    if(vs.isIn(values)==true)
      System.out.println("Alle Werte enthalten");
  }
}
```

Selbstreferenz

- Referenz **this**, mit der innerhalb einer Klasse auf ein Objekt der Klasse selbst verwiesen wird
- Zugriff auf Klassenmember innerhalb der Klasse durch

  this. *member-name*

- in Beispiel 4.5h: `if(values[i]==this.values[j])`

4.5.3 Rückgabewerte einer Methode

Objektreferenzen als Rückgabewert

Beispiel 4.5i

```
public class ValueSet {
  private int count;
  private int[] values = new int[100];
  public void printValues(){..}
  private boolean addValue(int value){..}
```

```
  public void addValues(int... values){..}
  public ValueSet incAll(ValueSet v){
    for(int i=0; i<v.count; i++)
      v.values[i]++;
    return v;
  }
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet();
    vs.addValues(33,-6,12);
    vs.incAll(vs);
    vs.printValues();           // 34 -5 13
    vs.incAll(vs.incAll(vs));
    vs.printValues();           // 36 -3 15
  }
}
```

Beispiel 4.5j

```
public class ValueSet {
  private int count;
  private int[] values = new int[100];
  public void printValues(){..}
  private boolean addValue(int value){..}
  public void addValues(int... values){..}
  public ValueSet incAll(ValueSet v){..}
  public ValueSet incAll(){
    for(int i=0; i<this.count; i++)
      this.values[i]++;
    return this;
  }
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet();
    vs.addValues(33,-6,12);
    vs.incAll();
    vs.printValues();      // 34 -5 13
    vs.incAll().incAll();
    vs.printValues();      // 36 -3 15
  }
}
```

4.5.4 Überladen von Methoden

- Zwei **Methoden heißen überladen**, wenn sie den gleichen Namen besitzen und sich in Anzahl und/oder Typ ihrer Parameter unterscheiden oder

- Compiler und JVM unterscheiden verschiedene Versionen einer Methode anhand von Typ und Anzahl der Parameter

  ⇒ (Name, Typen, Anzahl) = Signatur einer Methode

Beispiel

```
vs.IncAll(vs)    ruft   public ValueSet IncAll(ValueSet v)  auf
vs.IncAll()      ruft   public ValueSet IncAl()  auf
```

- besitzen mehrere überladene Methoden eine Parameterliste sowie weitere Parameter des gleichen Typs, dann ist ein Methodenaufruf wegen Mehrdeutigkeit nicht möglich

Beispiel 4.5k

```
public class MyClass {
  public void MyMethod(int... i){..}
  public void MyMethod(int a, int... i){..}
}
public class Programm {
  public static void main(String[] args){
    MyClass C = new MyClass();
    C.MyMethod(1);          // Mehrdeutigkeit
    C.MyMethod(1,2);        // Mehrdeutigkeit
  }
}
```

- Wird als Aufrufparameter ein Typ verwendet, der nicht definiert ist, versucht der Compiler in den nächst-höherwertigen Typ zu konvertieren
- sind mehrere Methoden möglich, wird die Methode mit dem höchstwertigen Typ aufgerufen

Beispiel 4.5l

```
public class MyClass {
  public void MyMethod(float f){
    System.out.println("M1");
  }
  public void MyMethod(double d){
    System.out.println("M2");
```

```
  }
}
public class Programm {
  public static void main(String[] args){
    MyClass C = new MyClass();
    C.MyMethod(1);     // M1
    C.MyMethod(1.1);   // M2
  }
}
```

4.6 Konstruktoren

Beispiel (fehlerhaft)

```
public class ValueSet {
  private int count;
  private int capacity;
  private int[] values;
  public void printValues(){..}
  private boolean addValue(int value){..}
  public void addValues(int... values){..}
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet();
    vs.addValues(33,-6,12);      //Laufzeitfehler
    vs.printValues();
  }
}
```

Beispiel 4.6a

```
public class ValueSet {
  private int count;
  private int capacity;
  private int[] values;
  public void printValues(){..}
  private boolean addValue(int value){..}
  public void addValues(int... values){..}
  public void initialize(int capacity){
    this.capacity = capacity;
    this.values = new int[capacity];
  }
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet();
    vs.initialize(50);
    vs.addValues(33,-6,12);
    vs.printValues();      // 33 -6 12
  }
}
```

- **Konstruktoren** einer Klasse sind spezielle Methoden, die bei der Instanziierung der Klasse mit new aufgerufen werden und die Felder initialisieren

Konstruktordeklaration

Syntax

> *ConstructorDeclaration ::=*
>
> *ConstructorHeader ConstructorBlock*
>
> *ConstructorHeader** *::=*
>
> *ConstructorModifiers*$_{opt}$ *Identifier* (*FormalParameterList*$_{opt}$)
>
> *ConstructorModifiers ::=*
>
> **public** | **protected** | **private**
>
> *ConstructorBlock ::=*
>
> { ExcplicitConstructorInvocation$_{opt}$ StatementsOrDeclarations$_{opt}$ }

* wird später noch vervollständigt

- name: ein Konstruktor hat generell den gleichen Namen wie die Klasse
- Konstruktoren geben keinen Wert zurück

Instanziierung von Klassenobjekten

- die Instanziierung einer Klasse erfolgt generell durch Aufruf eines Konstruktors:

  object-reference **= new** *constructor* **;**

- Instanziierungsprozeß:
  1. Speicherallokation für das Objekt im Heap
  2. Initialisierung der Felder mit Initialwerten (falls definiert), sonst mit 0, "" oder null
  3. Aufruf des Konstruktors

Beispiel 4.6b

```
public class ValueSet {
   private int count;
   private int capacity;
   private int[] values;
   public ValueSet(int capacity){
      this.capacity = capacity;
      this.values = new int[capacity];
   }
```

```
  public void printValues(){..}
  private boolean addValue(int value){..}
  public void addValues(int... values){..}
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet(50);
    vs.addValues(33,-6,12);
    vs.printValues();       // 33 -6 12
  }
}
```

Standardkonstruktor

- wurde kein Konstruktor explizit deklariert, erzeugt das System implizit einen parameterlosen Standardkonstruktor: *class-name* **( )**
- der vom System erzeugte Standardkonstruktor (auch default constructor) für eine Klasse hat die gleiche Sichtbarkeit wie die Klasse selbst
- sobald ein Konstruktor explizit definiert wurde, wird der implizite Standardkonstruktor nicht mehr erzeugt

Beispiel (fehlerhaft)

```
public class ValueSet {
  private int count;
  private int capacity;
  private int[] values;
  public ValueSet(int capacity){..}
...
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet();    //Fehler
...
  }
}
```

- Konstruktoren können überladen werden

Beispiel 4.6.c

```
public class ValueSet {
  private int count;
  private int capacity;
  private int[] values;
  public ValueSet(){
    this.capacity = 100;
    this.values = new int[100];
  }
```

```
   public ValueSet(int capacity){
      this.capacity = capacity;
      this.values = new int[capacity];
   }
   public void printValues(){..}
   private boolean addValue(int value){..}
   public void addValues(int... values){..}
}
public class Programm {
   public static void main(String[] args){
      ValueSet vs1 = new ValueSet(50);
      vs1.addValues(33,-6,12);
      vs1.printValues();      // 33 -6 12
      ValueSet vs2 = new ValueSet();
      vs2.addValues(66,17);
      vs2.printValues();      // 66 17
   }
}
```

Copykonstruktor

- Konstruktor, der als Parameter ein typgleiches Objekt übernimmt und damit seine Felder initialisiert

Beispiel 4.6.d

```
public class ValueSet {
   private int count;
   private int capacity;
   private int[] values;
   public ValueSet(){..}
   public ValueSet(int capacity){..}
   public ValueSet(ValueSet origin){
      this.capacity = origin.capacity;
      this.values = new int[origin.capacity];
      for (int i = 0; i < origin.count; i++){
         this.values[i] = origin.values[i];
         this.count++;
      }
   }
   public void printValues(){..}
   private boolean addValue(int value){..}
   public void addValues(int... values){..}
}
public class Programm {
   public static void main(String[] args){
      ValueSet vs1 = new ValueSet(50);
      vs1.addValues(33,-6,12);
      vs1.printValues();        // 33 -6 12
      ValueSet vs2 = new ValueSet(vs1);
      vs2.printValues();        // 33 -6 12
   }
}
```

Konstruktorverkettung

Beispiel

```
public ValueSet(int... values){
    this.capacity = 100;
    this.values = new int[100];
    addValues(values);
}
```

Syntax

ExcplicitConstructorInvocations ::=

this (*ArgumentList*$_{opt}$)
| **super** (*ArgumentList*$_{opt}$)

- Über Referenz **this** kann für einen Konstruktor die Funktionalität eines anderen, bereits definierter Konstruktors der gleichen Klasse aufgerufen werden
- Über Referenz **super** kann für einen Konstruktor die Funktionalität eines anderen, bereits definierter Konstruktors der übergeordneten Basisklasse aufgerufen werden
- Verkettungen erfolgen entsprechend der Syntax immer als erste Anweisungen eines Constructor-Blocks

Beispiel 4.6.e

```
public class ValueSet {
    private int count;
    private int capacity;
    private int[] values;
    public ValueSet(){..}
    public ValueSet(int capacity){..}
    public ValueSet(int... values){
        this();
        addValues(values);
    }
    public ValueSet(ValueSet origin){..}
    public void printValues(){..}
    private boolean addValue(int value){..}
    public void addValues(int... values){..}
}
public class Programm {
    public static void main(String[] args){
        ValueSet vs = new ValueSet(36,-3,15);
        vs.printValues();  // 36 -3 15
    }
}
```

4.7 Eigenschaften (properties)

- **Eigenschaften** sind (private) Felder, auf die unmittelbar durch spezielle öffentliche Methoden - Getter und Setter - zugegriffen wird

Getter

- ein **Getter** ist eine öffentliche Methode, die den Wert eines privaten Feldes zurückgibt:

public *type* **get***FieldIdentifier* **( )** *MethodBlock*

Setter

- ein **Setter** ist eine öffentliche Methode, die einem privaten Feld einen Wert zuweist:

public void set*FieldIdentifier* **(** *type fieldIdentifier* **)** *MethodBlock*

Beispiel 4.7a

```
public class ValueSet {
  private int count;
  private int capacity;
  private int[] values;
  public ValueSet(int capacity){..}
  public int getCapacity(){
    return this.capacity;
  }
  public void setCapacity(int capacity){
    int values[] = new int[capacity];
    for(int i=0; i< this.capacity; i++)
      values[i] = this.values[i];
    this.capacity = capacity;
    this.values = values;
  }
  public void printValues(){..}
  private boolean addValue(int value){..}
  public void addValues(int... values){..}
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet(50);
    vs.addValues(36,-3,15);
    System.out.println("Kapazität: " + vs.getCapacity()); // Kapazität: 50
    vs.printValues();                                     // 36 -3 15
    vs.setCapacity(100);
    System.out.println("Kapazität: " + vs.getCapacity()); // Kapazität: 100
    vs.printValues();                                     // 36 -3 15
  }
}
```

- mittels **Settern** lassen sich klassenspezifische Integritätsbedingungen realisieren

4.8 Statische Klassenmember

Beispiel 4.8a

```
public class ValueSet {
   private int count;
   private int capacity;
   private int[] values;
   private int numberOfObjects;
   public ValueSet(int capacity){
      this.capacity = capacity;
      this.values = new int[capacity];
      this.numberOfObjects++;
   }
   public int getNumberOfObjects(){
      return this.numberOfObjects;
   }
}
public class Programm {
   public static void main(String[] args){
      ValueSet vs1 = new ValueSet(50);

   System.out.println("Objektanzahl:"+vs1.getNumberOfObjects());//Objektanzahl:1
      ValueSet vs2 = new ValueSet(30);

   System.out.println("Objektanzahl:"+vs2.getNumberOfObjects());//Objektanzahl:1
   }
}
```

- **statische Member** (Modifizierer **static**) gehören nicht einzelnen Instanzen, sondern allen Instanzen einer Klasse gemeinsam
- **statische Member** können sein:
  - statische Felder (Klassenfelder)
  - statische Methoden (Klassenmethoden)
  - (statische Klassen)

- **Klassenfelder**:
  sind für alle Objekte einer Klasse identisch
- **Instanzfelder**:
  sind objektspezifisch
- **Klassenmethoden**:
  bestimmen das Verhalten einer Klasse unabhängig von einem konkreten Objekt
- **Instanzmethoden**:
  Ausführung ist an ein konkretes Objekt gebunden

Statische Felder

Beispiel 4.8b

```
public class ValueSet {
   private int count;
   private int capacity;
   private int[] values;
-> private static int numberOfObjects;
   public ValueSet(int capacity){
      this.capacity = capacity;
      this.values = new int[capacity];
      this.numberOfObjects++;
   }
   public int getNumberOfObjects(){
      return this.numberOfObjects;
   }
}
public class Programm {
   public static void main(String[] args){
      ValueSet vs1 = new ValueSet(50);

   System.out.println("Objektanzahl:"+vs1.getNumberOfObjects());//Objektanzahl:1
      ValueSet vs2 = new ValueSet(30);

   System.out.println("Objektanzahl:"+vs2.getNumberOfObjects());//Objektanzahl:2
   }
}
```

Statische Methoden

- statische Methoden (Klassenmethoden) dienen der Bereitstellung von Funktionen, die nicht an bestimmte Instanzen der Klasse gebunden sind

- Der Zugriff auf statische Member erfolgt außerhalb einer Klasse durch

  ClassIdentifier **.** *Member*

Beispiel 4.8c

```
public class ValueSet {
   private int count;
   private int capacity;
   private int[] values;
   private static int numberOfObjects;
   public ValueSet(int capacity){
      this.capacity = capacity;
      this.values = new int[capacity];
      this.numberOfObjects++;
   }
```

```
->public static int getNumberOfObjects(){
->   return numberOfObjects;
  }
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs1 = new ValueSet(50);
->  System.out.println("Objektanzahl:"+ValueSet.getNumberOfObjects());
                                                          //Objektanzahl:1
    ValueSet vs2 = new ValueSet(30);
->  System.out.println("Objektanzahl:"+ValueSet.getNumberOfObjects());
                                                          //Objektanzahl:2
  }
}
```

Beispiel 4.8d

```
public class Math {
  public static int plus(int x, int y){
    return x+y;
  }
  public static int minus(int x, int y){
    return x-y;
  }
}
public class Programm {
  public static void main(String[] args){
    int a = 3;
    int b = 2;
    System.out.println("a + b = " + Math.plus(a,b));   // a+b=5
    System.out.println("a - b = " + Math.minus(a,b));  // a-b=1
  }
}
```

UML: Statische Klassenmember

- statische Klassenmember können im Klassendiagramm durch Unterstreichung kenntlich gemacht werden

Beispiel

ValueSet
- count : int - capacity : int - values : int[] - <u>numberOfObjects : int</u>
+ ValueSet(capacity : int) : void + <u>getNumberOfObjects( ) : int</u>

4.9 Generische Klassen und Methoden

Beispiel 4.9a

```
public class ValueSet {
   private int count;
   private int capacity;
-> private Object[] values;
   public ValueSet(int capacity){
      this.capacity = capacity;
->    this.values = new Object[capacity];
   }
   public void printValues(){..}
-> private boolean addValue(Object value){
      for (int i = 0; i < count; i++)
         if(values[i]==value) return false;
      values[count] = value;
      count++;
      return true;
   }
-> public void addValues(Object... values){
->    for(Object value : values)
         addValue(value);
   }
}
public class Programm {
   public static void main(String[] args){
      ValueSet vs1 = new OValueSet(50);
      vs1.addValues(36,-3,15);
      vs1.printValues();      // 36 -3 15
      ValueSet vs2 = new ValueSet(100);
      vs2.addValues("Anna","Hannes","Willi","Paul");
      vs2.printValues();      // Anna Hannes Willi Paul
   }
}
```

Beispiel 4.9b

```
public class Programm {
   public static void main(String[] args){
      ValueSet vs = new ValueSet(50);
      vs.addValues(36,"Anna", 3.14, true);
      vs.printValues();     // 36 Anna 3.14 true
   }
}
```

Beispiel 4.9c (fehlerhaft)

```
public class ValueSet {
   private int count;
```

```
    private int capacity;
    private Object[] values;
    public ValueSet(int capacity){
      this.capacity = capacity;
      this.values = new Object[capacity];
    }
    public void printValues(){
      for (int i = 0; i < count; i++)
        System.out.println(values[i]);
      System.out.println();
    }
    private boolean addValue(Object value){
      for (int i = 0; i < count; i++)
        if(values[i]==value) return false;
      values[count] = value;
      count++;
      return true;
    }
    public void addValues(Object... values){
      for(Object value : values)
        addValue(value);
    }
    public Object getValue(int index){
      return this.values[index];
    }
  }
  public class Programm {
    public static void main(String[] args){
      ValueSet vs = new ValueSet(50);
      vs.addValues(36,"Anna", 3.14, true);
      vs.printValues();         // 36 Anna 3.14 true
      int v;
      v = (int)vs.getValue(0);  // hier o.k.
      v = (int)vs.getValue(1);  // hier Laufzeitfehler
    }
  }
```

- Nachteile der Lösung mit Datentyp Object:
  - Typprüfung kann nicht vom Compiler geleistet werden
  - Boxing/Unboxing führt zu schlechter Performance

4.9.1 Generische Klassen

- Generische Klassen sind Klassen, die Platzhalter für Datentypen besitzen, die erst zur Laufzeit konkretisiert werden (Typparameter)
- Erleichterung, da
  - striktere Typprüfung zur Kompilierzeit möglich
  - explizite Konvertierungen zwischen Datentypen seltener erforderlich
  - keine Notwendigkeit von Boxing-Vorgängen und Typprüfungen zur Laufzeit

Syntax

GenericClassDeclaration ::=

ClassModifiers$_{opt}$ **class** *Identifier* **<** *TypeParameterList* **>** **{** *ClassBodyDeclarations*$_{opt}$ **}**

TypeParameterList ::=

TypeParameter
| *TypeParameterList* **,** *TypeParameterList*

TypeParameter ::=

TypeVariable *TypeBound*$_{opt}$

- Typparameter können innerhalb einer generischen Klasse an jeder Stelle stehen, an denen ein Datentyp stehen kann

- bei der Instanziierung einer generischen Klasse wird jedem Typparameter in der Reihenfolge seines Auftretens ein konkreter Datentypen zugeordnet und damit jeder in der Klasse auftretende Typparameter durch den konkreten Datentyp ersetzt

Syntax

GenericClassInstance ::=

GenericClassIdentifier **<** *ActualTypeList* **>** *Variable*
| *GenericClassIdentifier* **<** *ActualTypeList* **>** *Variable* **= new** *GenericConstructor*

GenericConstructor ::=

GenericClassIdentifier **<** *ActualTypeList* **>** **(** *ActualParameterList*$_{opt}$ **)**
| *GenericClassIdentifier* **<>** **(** *ActualParameterList*$_{opt}$ **)**

ActualTypeList ::=

ReferenceType
| *ActualTypeList* *ActualTypeList*

Einschränkungen

- Aktuelle Typen, die Typparameter ersetzen, dürfen nur Referenztypen sein
- ein Typparameter einer Klasse darf nicht bei der Definition statischer Felder oder Methoden verwendet werden

- von einem Typparameter kann kein Objekt angelegt werden - z.B. mit **new T**

Beispiel 4.9d

```
public class ValueSet<T> {
  private int count;
  private int capacity;
  private Object[] values;
  public ValueSet(int capacity){
    this.capacity = capacity;
    this.values = new Object[capacity];
  }
  public void printValues(){..}
->private boolean addValue(T value){
    for (int i = 0; i < count; i++)
      if(values[i]==value) return false;
    values[count] = value;
    count++;
    return true;
  }
->public void addValues(T... values){
->  for(T value : values)
      addValue(value);
  }
}
public class Programm {
  public static void main(String[] args){
->  ValueSet<Integer> vs1 = new ValueSet<Integer>(50);
    vs1.addValues(36,-3,14);
    vs1.printValues();      // 36 -3 14

->  ValueSet<String> vs2 = new ValueSet<String>(50);
    vs2.addValues("Anna","Hannes");
    vs2.printValues();      // Anna Hannes

//  vs2.addValues(36,3.14,true);       ergibt nun einen Fehler
  }
}
```

Bounded Typ-Parameter

- durch Angabe eine *BoundType* nach der TypeVariable in der Definition einer generischen Klasse wird festgelegt, daß nur die aktuellen Datentypen die Typvariable ersetzen dürfen, die von einer Basisklasse abgeleitet sind:

  TypeBound ::= **extends** *BaseClass*

Beispiel 4.9e

```
public class ValueSet<T extends Number> {
```

```
   private int count;
   private int capacity;
   private Object[] values;
   public ValueSet(int capacity){..}
   public void printValues(){..}
   private boolean addValue(T value){..}
   public void addValues(T... values){..}
}
public class Programm {
   public static void main(String[] args){
      ValueSet<Integer> vs1 = new ValueSet<Integer>(50);
      vs1.addValues(36,-3,14);
      vs1.printValues();        // 36 -3 14

      ValueSet<Double> vs2 = new ValueSet<Double>(50);
      vs2.addValues(3.61,-3.3,1.46);
      vs2.printValues();        // 3.61 -3.3 1.46

//    ValueSet<String> vs3 = new ValueSet<String>(50);   ergibt einen Fehler

   }
}
```

4.9.2 Generische Methoden

- Generische Methoden sind Methoden, die Platzhalter (Typparameter) für Datentypen besitzen, die erst zur Laufzeit konkretisiert werden

Syntax

GenericMethodDeclaration ::=

MethodHeader MethodBlock

MethodHeader ::=

$MethodModifiers_{opt}$ **<** *TypeParameterList* **>** *ResultType Identifier* **(**$FormalParameterList_{opt}$ **)**

TypeParameterList ::=

TypeParameter
| *TypeParameterList* **,** *TypeParameterList*

TypeParameter ::=

TypeVariable $TypeBound_{opt}$

- Typparameter können sowohl Bestandteil der Parameterliste oder des Methodenrumpfs als auch der Rückgabetyp sein

- generische Methoden können Member generischer oder nicht-generischer Klassen sein

- es ist schlechter Stil, in einer Methode den gleichen Typparameter zu definieren, wie in der Klasse

- generische Methoden können wie generische Klassen durch TypeBound-Angabe beschränkt werden

Beispiel 4.9f (erweitert)

```
public class MyClass {
   public static <T extends Number> T selectElement(T[] a, int b){
      return a[b];
   }
}
public class Programm {
   public static void main(String[] args){
      Integer[] x = {10,20,30};
      System.out.println(MyClass.selectElement(x,1));     // 20

      String[] y ={"Hannes","Willi","Paul"};
      System.out.println(MyClass.selectElement(y,2));     // Fehler
   }
}
```

- generische Methoden können durch generische und nicht-generische Methoden überladen werden

Beispiel 4.9g

```
public class MyClass {
   public static <T> T selectElement(T[] a, int b){
      System.out.print("Generische Methode: ");
      return a[b];
   }
   public static Integer selectElement(Integer[] a, int b){
      System.out.print("Nicht-Generische Methode: ");
      return a[0];
   }
}
public class Programm {
   public static void main(String[] args){
      Integer[] x = {10,20,30};
      System.out.println(MyClass.selectElement(x,1));
                                          // Nicht-Generische Methode: 20

      String[] y ={"Hannes","Willi","Paul"};
      System.out.println(MyClass.selectElement(y,2));
                                          // Generische Methode: Paul
```

```
  }
}
```

5 Spezielle Referenztypen

5.1 Die Klasse Object

- alle Referenztypen in Java (Klassen, Schnittstellen, Arrays, ...) sind von der Klasse Object abgeleitet

- damit erben alle Klasse die Methoden der Klasse **java.lang.Object**

Methoden der Klasse Object

`public boolean equals(Object obj)`

Liefert true, wenn obj ein Verweis auf dieses Exemplar ist (Referenzgleichheit). In den Klassen der Standardbibliothek wird diese Methode so überschrieben, dass sie auf Wertegleichheit prüft.

`public int hashCode()`

Liefert Hash-Code eines Objektes

`public String toString()`

Liefert den Namen und den Hashcode eines Objektes

`public Class getClass()`

Liefert ein Class-Objekt, das die Klasse dieses Objektes repräsentiert

`protected void finalize()`

Wird aufgerufen, wenn das Objekt durch den Garbage Collector entfernt wird. Standardmäßig ist diese Methode leer implementiert. Sie muss gegebenenfalls von den Unterklassen überschrieben werden, falls es nötig ist, vor der Beseitigung des Objekts Abschlussaktionen auszuführen.

Beispiel 5.1a für toString()

```
public class MyClass1 {
  private String name ="Meier";
  private String vorname = "Horst";
}
public class MyClass2 {
  private String name ="Meier";
  private String vorname = "Horst";
  public String toString(){
    return vorname+" "+name;
  }
}
```

```
public class Programm {
   public static void main(String[] args){
      double x = 3.14;
      System.out.println(x);         // 3.14

      MyClass1 c1 = new MyClass1();
      System.out.println(c1);        // Beispiel.MyClass1@7c6768

      MyClass2 c2 = new MyClass2();
      System.out.println(c2);        // Horst Meier
   }
}
```

Gleichheit von Objekten

- Zwei Objekte obj1 und obj2 sind **typgleich**, wenn sie vom selben Typ sind aber unterschiedliche Instanzwerte (Feldwerte) besitzen

Beispiel 5.1b (1)

```
public class MyClass {
   private double x;
   public MyClass(double x){
      this.x = x;
   }
}
public class Programm {
   public static void main(String[] args){
      MyClass c1 = new MyClass(3.14);
      MyClass c2 = new MyClass(5.21);

      System.out.println("c1: "+c1.hashCode());         // c1: 32635808
      System.out.println("c2: "+c2.hashCode());         // c2: 20732290

      if(c1==c2)
         System.out.println("c1==c2 is true");
      else
         System.out.println("c1==c2 is false");         // c1==c2 is false
      if(c1.equals(c2)==true)
         System.out.println("c1.equals(c2) is true");
      else
         System.out.println("c1.equals(c2) is false");  // c1.equals(c2) is false
   }
}
```

- Zwei Objekte obj1 und obj2 sind **gleich** (Wertegleich), wenn sie vom selben Typ sind, gleiche Instanzwerte besitzen, aber ihre Referenzen auf unterschiedliche Speicheradressen verweisen

Beispiel 5.1b (2)

```
public class MyClass {
```

```
   private double x;
   public MyClass(double x){
      this.x = x;
   }
}
public class Programm {
   public static void main(String[] args){
      MyClass c1 = new MyClass(3.14);
      MyClass c2 = new MyClass(3.14);

      System.out.println("c1: "+c1.hashCode());          // c1: 32635808
      System.out.println("c2: "+c2.hashCode());          // c2: 29293232

      if(c1==c2)
         System.out.println("c1==c2 is true");
      else
         System.out.println("c1==c2 is false");          // c1==c2 is false
      if(c1.equals(c2)==true)
         System.out.println("c1.equals(c2) is true");
      else
         System.out.println("c1.equals(c2) is false");  // c1.equals(c2) is false
   }
}
```

- Zwei Objekte obj1 und obj2 sind **identisch** (Referenzgleich), wenn sie vom selben Typ sind und ihre Referenzen auf dieselbe Speicheradresse verweisen

Beispiel 5.1b (3)

```
public class MyClass {
   private double x;
   public MyClass(double x){
      this.x = x;
   }
}
public class Programm {
   public static void main(String[] args){
      MyClass c1 = new MyClass(3.14);
      MyClass c2 = c1;

      System.out.println("c1: "+c1.hashCode());          // c1: 32635808
      System.out.println("c2: "+c2.hashCode());          // c2: 32635808

      if(c1==c2)
         System.out.println("c1==c2 is true");           // c1==c2 is true
      else
         System.out.println("c1==c2 is false");
      if(c1.equals(c2)==true)
         System.out.println("c1.equals(c2) is true");   // c1.equals(c2) is true
      else
         System.out.println("c1.equals(c2) is false");
   }
}
```

- Für den Wertevergleich kann equals() überschrieben werden

Beispiel 5.1c

```
public class MyClass {
  private double x;
  public MyClass(double x){
    this.x = x;
  }
  public boolean equals(Object o){
    if(!(o instanceof MyClass))
      return false;
    if(o == this)
      return true;
    if(this.x ==((MyClass)o).x)
      return true;
    else
      return false;
  }
}
public class Programm {
  public static void main(String[] args){
    MyClass c1 = new MyClass(3.14);
    MyClass c2 = new MyClass(3.14);

    System.out.println("c1: "+c1.hashCode());         // c1: 32635808
    System.out.println("c2: "+c2.hashCode());         // c2: 29293232

    if(c1==c2)
      System.out.println("c1==c2 is true");
    else
      System.out.println("c1==c2 is false");          // c1==c2 is false
    if(c1.equals(c2)==true)
      System.out.println("c1.equals(c2) is true");    // c1.equals(c2) is true
    else
      System.out.println("c1.equals(c2) is false");
  }
}
```

5.2 Wrapper-Klassen

- Eine **Wrapper-Klasse** ist eine Klasse, die in objektorientierten Sprachen primitive Datentypen umhüllt
- Vorteil: Wrapper-Datentypen besitzen grundsätzlich alle objektorientierte Eigenschaften, insbesondere
  - können sie Methoden besitzen

  - sind sie von Object abgeleitet
  - können sie selbst Ableitungen besitzen
- alle Wrapper-Klassen sind im Paket **java.lang** definiert

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

- alle numerischen Wrapper-Klassen (Byte ..Double) besitzen die gemeinsame Basisklasse **Number**
- alle Wrapper-Klassen implementieren die Schnittstelle **Comparable**
- alle numerischen Wrapper-Klassen (Byte ..Double) und Character implementieren die Schnittstelle **Serializable**

Erzeugen von Wrapper-Objekten

- 3 Möglichkeiten:
  - durch statische **valueOf(val)**-Methoden mit Übergabe eines String oder primitiven Ausdrucks
  - durch **Konstruktor** der Wrapper-Klasse
  - durch direkte **Zuweisung** eines primitiven Wertes (Boxing)

Beispiel 5.2a

```
Integer  i1 = Integer.valueOf("3");     // valueOf()
Double   d1 = Double.valueOf(3.2+3.3);  // valueOf()

Integer  i2 = new Integer(6);           // Konstruktor
Double   d2 = new Double(6.1);          // Konstruktor

Integer  i3 = 7;                        // direkte Zuweisung
Double   d3 = 7.1;                      // direkte Zuweisung
```

Konvertierung von Wrapper-Objekten in einen String

- 2 Möglichkeiten:
  - durch die statische Klassenmethode **toString(val)** mit Übergabe eines primitiven Ausdrucks
  - durch die Instanzmethode **toString()**

Beispiel 5.2a (weiter)

```
int    i1 = 3;
```

```
System.out.println(Integer.toString(i1)); // statische Methode
double d1 = 3.14;
System.out.println(Double.toString(d1));  // statische Methode

Integer i2 = 3;
System.out.println(i2.toString());        // Objekt-Methode
Double  d2 = 3.14;
System.out.println(d2.toString());        // Objekt-Methode
```

Vergleichsmethoden

- 3 Möglichkeiten:
  - durch die statische Klassenmethoden **compare(val1 , val2)** mit Übergabe zweier primitiver Ausdrücke, liefert

    -1 wenn val1 < val1,

    1 wenn val1 > val2,

    0 wenn val1 == val2
  - durch die Instanzmethode **compareTo(Val)** mit Übergabe eines Wrapper-Objektes, liefert

    -1 wenn Objekt < Val,

    1 wenn Objekt > Val,

    0 wenn Objekt == Val
  - durch die Instanzmethode **equals(Val)** mit Übergabe eines Wrapper-Objektes, liefert

    true wenn Objekt == Val,

    false sonst

Beispiel 5.2a (weiter)

```
int i1 = 4;
int i2 = 6;
System.out.println(Integer.compare(i1,i2));                             // -1

Double d1 = 3.14;
Double d2 = 1.22;
System.out.println(d1.compareTo(d2));                                   // 1

Character c1 ='A';
Character c2 = 'A';
System.out.println(c1.equals(c2));                                      // true

System.out.println(Integer.valueOf(i1).equals(Integer.valueOf(i2)));// false
```

Klasse Integer (beispielhaft)

<u>Konstruktoren</u>

```
public Integer(int value)
```

Erzeugt ein neues Integer-Objekt mit dem Wert value.

```
public Integer(String s)
```
Erzeugt ein neues Integer-Objekt mit dem Wert eines String-Objekts zur Basis 10.

Methoden (Auszug)

```
public int compareTo(Integer anotherInteger)
```
Vergleicht den Wert dieses Objekts mit dem von anotherInteger. Der Rückgabewert ist null, wenn die Werte gleich sind. Er ist kleiner als null, wenn der Wert dieses Objekts kleiner als der von anotherInteger ist. Der Rückgabewert ist größer als null, wenn der Wert dieses Objekts größer als anotherInteger ist.

```
public byte  byteValue()
public short shortValue()
public int   intValue()
public long  longValue()
public float floatValue()
public double doubleValue()
```
Liefert den Wert des Objekts als byte ... double zurück.

```
public static String toBinaryString(int i)
public static String toOctalString(int i)
public static String toHexString(int i)
```
Konvertiert i in einen Binär/Oktal/Hexadezimal-String, wobei i als vorzeichenlos interpretiert wird, und liefert ihn zurück.

```
public String toString()
public static String toString(int i)
```
Liefert den Wert des Integer-Objekts bzw. von i als String zurück, wobei die Basis 10 verwendet wird.

```
public static int parseInt(String s)
```
Konvertiert den String s in einen int-Wert und liefert ihn in einem neuen Integer-Objekt zurück, wobei die Basis 10 verwendet wird.

```
public static Integer valueOf(String s)
public static Integer valueOf(String s, int basis)
```
Konvertiert den Inhalt von s in einen int-Wert und liefert ihn in einem neuen Integer-Objekt zurück. Die Ziffern werden zur Basis basis interpretiert (fehlt die, zur Basis 10). Der Parameter kann auch ein int-Ausdruck sein.

```
public boolean equals(Object obj)
```
Liefert genau dann true, wenn obj ein Exemplar der Klasse Integer ist, das denselben Wert besitzt wie dieses Objekt, sonst false.

Beispiel 5.2a (weiter)

```
Integer i = 123456789;
System.out.println(i.byteValue());                    // 21
System.out.println(i.longValue());                    // 123456789
System.out.println(i.doubleValue());                  // 1.23456789E8
```

```
System.out.println(Integer.toHexString(123456789)); // 75bcd15
System.out.println(Integer.valueOf("15",16));       // 21
```

Boxing und Unboxing

Beispiel 5.2b

```
public class MyClass {
  public int w;
}
public class Programm {
  public static void main(String[] args){
    MyClass c1 = new MyClass();
    c1.w = 8;
    MyClass c2 = c1;
    System.out.println(c1.w+" "+c2.w); // 8  8
    c2.w=3;
    System.out.println(c1.w+" "+c2.w); // 3  3
  {
{
```

Beispiel 5.2b (weiter)

```
public class Programm {
  public static void main(String[] args){
    Object o1 = new Object();
    o1 = 8;
    Object o2 = o1;
    System.out.println(o1+" "+o2);   // 8  8
    o2 = 3;
    System.out.println(o1+" "+o2);   // 8  3
  }
}
```

Beispiel 5.2b (weiter)

```
public class Programm {
  public static void main(String[] args){
    double d = 8.3;
    System.out.println(d);    // 8.3
    Double o;
    o = d;
    System.out.println(o);    // 8.3
  }
}
```

- **Boxing**

Zuweisung eines Wertes eines primitiven Typs an ein entsprechendes Wrapper-Objekt ('Verpacken' eines primitiven Objekts)

Wrapper-Objekt = primitiver Wert

- durch das System wird Speicherplatz in Heap allokiert und der Wert des primitiven Typs dort abgelegt (Kopie)

Beispiel 5.2b (weiter)

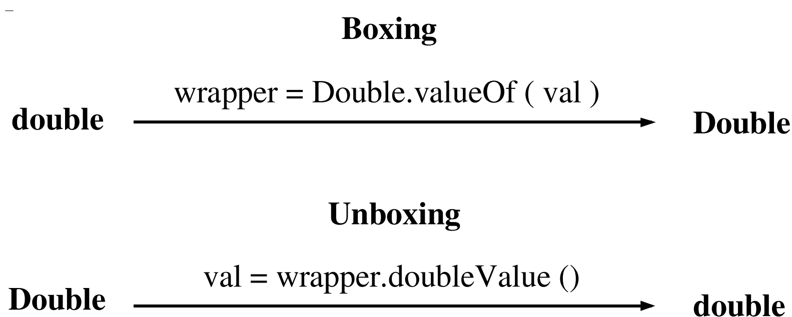
```
public class Programm {
  public static void main(String[] args){
    Double o = new Double(8.3);
    System.out.println(o);    // 8.3
    double d;
    d = o;
    System.out.println(d);    // 8.3
  }
}
```

- **Unboxing**

  Zuweisung eines Wrapper-Objekts an einen entsprechenden primitiven Typs ('Entpacken' eines Wrapper- Objekts)

  primitiver Typ = Wrapper-Objekt

- durch das System wird der Wert des Wrapper-Objektes aus dem Heap in den primitiven Typ kopiert



Beispiel 5.2b (weiter)

```
public class Programm {
  public static void main(String[] args){
    int i1 = 1234;
    int i2 = 1234;
    System.out.println(i1==i2);    // true

    Integer j1 = 1234;
```

```
        Integer j2 = 1234;
        System.out.println(j1==j2);    // false

        Integer k1 = 32;
        Integer k2 = 32;
        System.out.println(k1==k2);    // true(pool)

        Integer l1 = new Integer(32);
        Integer l2 = new Integer(32);
        System.out.println(l1==l2);    // false
    }
}
```

5.3 Strings

- Java-Syntax definiert keinen primitiven Datentyp *string*

- 3 Möglichkeien der Definition von Zeichenketten
  - Klasse **String** (unveränderbar, threadsicher)
  - Klasse **StringBuffer** (veränderbar, threadsicher)
  - Klasse **StringBuilder** (veränderbar, nicht threadsicher)

- Zeichenketten sind Referenztypen

5.3.1 Die Klasse String

Konstruktoren (Auszug)

`public String()`
Erzeugt ein neues String-Objekt, das mit einem leeren String initialisiert wird.

`public String(String value)`
Erzeugt ein neues String-Objekt, das mit einer Kopie des Inhalts von value initialisiert wird.

`public String(StringBuffer buffer)`
Erzeugt ein neues String-Objekt, das mit dem momentanen Wert eines StringBuffer-Objekts buffer initialisiert wird. Dessen Inhalt wird nicht kopiert, sondern der Datenbereich wird gemeinsam genutzt.

`public String(char[] value)`
Erzeugt ein neues String-Objekt, das mit einem char-Array initialisiert wird. Das Array wird hierbei in einen internen Puffer kopiert.

Beispiel 5.3a

```
String s1 = new String("der erste String");

char[] f = {'d','e','r',' ','z','w','e','i','t','e'};
String s2 = new String(f);
```

Operatoren

Zuweisungsoperator **=**

- bewirkt das Erzeugen eines String-Objektes (char-Folge + Längenwert) im Heap und Referenz darauf (String-Variable)
- werden mehrere gleiche Strings (identische char-Folgen) erzeugt, so wird nur einmal Speicherplatz im Heap allokiert. Alle String-Referenzen zeigen dann auf denselben Heap-Bereich

Beispiel 5.3a (weiter)

```
String s1 = "Hello";
String s2 = s1;

s2 = "World";

s2 = "Hallo";

s2 = "Hello";
```

- Objekte der Klasse String (Zeichenketten) sind immutable, d.h. sie können auch durch ihre Methoden nicht überschrieben werden
- Wird der Wert einer String-Variablen geändert, so wird ein neues String-Objekt erzeugt und das alte gelöscht

Stringverkettungsoperator **+**

- verkettet mehrere Strings miteinander oder Strings mit arithmetischen Ausdrücken und gibt einen String zurück
- Verkettungsausdrücke werden von links nach rechts abgearbeitet

Syntax

StringConcatenation ::=
  String **+** *String*
  | *String* **+** *ArithmeticExpression*
  | *ArithmeticExpression* **+** *String*

Beispiel 5.3a (weiter)

```
int i = 3;
double d = 3.2;
String s1 = "Hallo";
String s2;

s2 = s1 + s1 +", World";
System.out.println(s2);     // Ausgabe: 'HalloHallo, World'

s2 = i + " ist nicht " + d;
System.out.println(s2);     // 3 ist nicht 3.2

s2 = i + d +" ist anders ";
System.out.println(s2);     // 6.2 ist anders
```

***Stringverkettungsoperator* +=**

verkettet einen linken String mit einem rechten oder mit rechtem arithmetischen Ausdruck

Syntax

StringConcatenation ::=
String += *String*
| *String* += *ArithmeticExpression*

Methoden (Auszug)

```
public static String valueOf(boolean v)
public static String valueOf(char v)
public static String valueOf(int v)
public static String valueOf(long v)
public static String valueOf(float v)
public static String valueOf(double v)
```

Liefert den in einen String konvertierten Wert von v.

```
public boolean equals(Object obj)
```

Liefert true, wenn obj ein Exemplar der Klasse String ist, das denselben Wert besitzt wie dieses Objekt, sonst false.

```
public String concat(String str)
```

Liefert einen neuen String, der die Verkettung dieses Strings mit str enthält.

```
public int compareTo(String anotherString)
```

Liefert einen int-Wert, der kleiner als null ist, wenn dieser String lexikalisch kleiner ist als anotherString, null, wenn der String gleich anotherString ist, und einen Wert größer als null, wenn der String lexikalisch größer ist als anotherString.

```
public int indexOf(String str)
public int indexOf(int ch)
```

Liefert die erste Position, an der String str bzw. das Zeichen ch vollständig im String enthalten ist. Ist str/ch nicht im String enthalten, wird -1 geliefert.

```
public boolean isEmpty()
```
Liefert true, wenn der String leer ist, sonst false.

```
public int length()
```
Liefert die Länge des Strings.

```
public String substring(int beginIndex)
public String substring(int beginIndex, int endIndex)
```
Liefert einen neuen String, der nur die Zeichen ab der Position beginIndex und ggf. bis zur Position endIndex. endIndex bezeichnet hierbei den Index hinter dem letzten Zeichen, das kopiert werden soll.

Beispiel 5.3b

```
import java.util.Date;
public class Programm {
   public static void main(String[] args){
      String s1 = "Aktuelles Datum: ";
      String s2 = new Date().toString();
      String s3 = s1.concat( s2 );
      System.out.println(s3); // Aktuelles Datum: Tue Aug 05 13:18:29 CEST 2014
   }
}
```

Beispiel 5.3c

```
public class StringUtil {
   public static String substringBefore(String s, char c)
   {
      int pos = s.indexOf(c);
        return pos >= 0 ? s.substring(0,pos) : s;
   }
}
public class Programm {
   public static void main(String[] args){
      String s = "index.html";
      System.out.println(StringUtil.substringBefore(s,'.'));   // index
   }
}
```

Beispiel 5.3c (weiter)

```
public class StringUtil {
   ...
```

```
  static int numberOfParts( String s, String part )
  {
    if (s==null || s.isEmpty() || part==null || part.isEmpty())
      return 0;
    int count = 0;
    int pos = 0;
    while((pos = s.indexOf(part, pos)) != -1){
      count++;
      pos += part.length();
    }
    return count;
  }
}
public class Programm {
  public static void main(String[] args){
    int count = StringUtil.numberOfParts("Hallo, World! Hallo,Hallo!","Hallo");
    System.out.println(count);     // 3
  }
}
```

5.3.2 Die Klassen StringBuffer und StringBuilder

- **Eigenschaften** der Klassen StringBuffer und StringBuilder
  - Länge der Zeichenketten in einem StringBuffer/StringBuilder-Objekt ist nicht festgelegt
  - Länge vergrößert sich automatisch beim Anfügen weiterer Zeichen, wenn die aktuelle Länge nicht ausreicht
  - Inhalt der Objekte lässt sich verändern

- StringBuilder besitzt die gleichen Methoden wie StringBuffer, sind nur nicht synchronisiert

- StringBuilder besitzt eine etwas höhere Performance als StringBuffer

Konstruktoren

`public StringBuffer()`

Erzeugt ein neues Stringbuffer-Objekt mit einer voreingestellten Kapazität von 16 Zeichen.

`public StringBuffer(int length)`

Erzeugt ein neues Stringbuffer-Objekt mit einer voreingestellten Kapazität von length Zeichen.

`public StringBuffer(String str)`

Erzeugt ein neues Stringbuffer-Objekt mit dem Inhalt str. Die Kapazität ergibt sich aus der Länge von str plus 16.

Beispiel 5.3d

```
public class Programm {
  public static void main(String[] args){
    StringBuffer s1 = new StringBuffer();
    StringBuffer s2 = new StringBuffer(3);
  }
}
```

<u>Methoden</u> (Auswahl)

```
public int capacity()
```

Liefert die momentane Kapazität. (lenght() liefert die Länge der momentan gespeicherten Zeichenkette)

```
public void setLength(int newLength)
```

Setzt den String-Puffer auf die Länge newLength. Falls der String mehr Zeichen enthält, werden diese abgeschnitten. Fehlende Zeichen werden durch das Zeichen mit dem Codewert null ('\0') aufgefüllt.

```
public StringBuffer append(int i)
public StringBuffer append(long l)
public StringBuffer append(float f)
public StringBuffer append(double d)
```

Konvertiert einen int...double-Wert in einen String und hängt diesen am Ende des StringBuffer an.

```
public StringBuffer append(char c)
public StringBuffer append(String str)
public StringBuffer append(StringBuffer sb)
```

Hängt einen char..StringBuffer-Wert am Ende des StringBuffer an.

```
public StringBuffer insert(int offset, int i)
public StringBuffer insert(int offset, long l)
public StringBuffer insert(int offset, float f)
public StringBuffer insert(int offset, double d)
```

Konvertiert einen int...double-Wert in einen String und fügt ihn an Position offset ein.

```
public StringBuffer insert(int offset, char c)
public StringBuffer insert(int offset, String str)
```

Fügt das Zeichen c bzw. den String str an Position offset ein.

```
public StringBuffer delete(int start, int end)
```

Löscht die Zeichen von Position start einschließlich bis Position end ausschließlich aus dem String und liefert einen Verweis auf das aufgerufene Objekt. Wenn end größer gleich der momentanen Länge ist, werden alle Zeichen ab start einschließlich gelöscht.

Beispiel 5.3d (weiter)

```
public class Programm {
  public static void main(String[] args){
    StringBuffer s1 = new StringBuffer();
    StringBuffer s2 = new StringBuffer(3);
```

```
        System.out.println(s1.capacity());          // 16
        System.out.println(s2.capacity());          // 3

        s2.append("Hallo").append(" World");
        System.out.println(s2+"  "+s2.capacity());// Hallo World  18
    }
}
```

Beispiel 5.3d (weiter)

```
public class Programm {
    public static void main(String[] args){
        StringBuffer s1 = new StringBuffer("Hallo");
        StringBuffer s2 = new StringBuffer("Hallo");
        if(s1==s2)
            System.out.println("gleich");
        else
            System.out.println("ungleich");     // ungleich

        if(s1.equals(s2))
            System.out.println("gleich");
        else
            System.out.println("ungleich");     // ungleich


        if(s1.toString().equals(s2.toString()))
            System.out.println("gleich");       // gleich
        else
            System.out.println("ungleich");
    }
}
```

Beispiel 5.3e

```
public class Programm {
    public static void main(String[] args){
        String s="*";
        long before;
        long after;

        before = System.currentTimeMillis();
        for(int i=0; i<50000; i++)
            s = s + "*";
        after = System.currentTimeMillis();
        System.out.println(after-before);       // 2294

        before = System.currentTimeMillis();
        StringBuffer tmp = new StringBuffer(s);
        for(int i=0; i<50000; i++)
            tmp.append("*");
        s = tmp.toString();
        after = System.currentTimeMillis();
        System.out.println(after-before);       // 0
```

```
   }
}
```

- bei häufigen Änderungen eines Strings sollten diese über ein StringBuffer/StringBuilder-Objekt ausgeführt werden

5.4 Arrays

5.4.1 Array-Typ

- **Array** ist ein Datentyp, keine Klasse

- Arrays besitzen eine öffentliche Instanzvariable **lenght**, die die Anzahl der Elemente eines Array liefert

Syntax

SimpleArrayDeclaration ::=

  Type **[ ]** *Identifier* *Instantiator*$_{opt}$

Instantiator ::=

  = **{** *ValueList* **}**
 | **=** **new** *Type* **[** *Size* **]**

- ***Identifier*** ist eine Referenzvariable
- durch den ***Instatiator*** wird Speicherplatz im Heap der Größe *Size**sizeof(*Type*) allokiert

Eindimensionale Arrays

Beispiel

```
int[] a = {1,2,3};

// entspricht
int[] a = new int[3];
a[0] = 1;
a[1] = 2;
a[2] = 3;
```

Beispiel

```
int a[] = new int[3];
int a[3];                // Fehler
```

Beispiel

```
double[] a = {1.1, 2.2, 3.3};

for(int i=0; i<a.length; i++)
  System.out.println(a[i]);

//entspricht
for(double element : a)
  System.out.println(element);
```

Mehrdimensionale Arrays

Beispiel

```
int[][] a = new int[3][2];

//entspricht
int[][] a = new int[3][];
a[0] = new int[2];
a[1] = new int[2];
a[2] = new int[2];
```

Beispiel

```
int[][] a = new int[3];      // Fehler: 2. Dimension fehlt
int[][] a = new int[][2];    // Fehler: zunächst 1. Dimension instanziieren
int[][] a = new int[][];     // Fehler: Größe fehlt
```

Jagged Arrays

- **Jagged Arrays** sind mehrdimensionale Arrays, deren Elemente mit unterschiedlich langen Arrays instanziiert werden

Beispiel 5.4a

```
int[][] a = new int[3][];
a[0] = new int[2];
a[1] = new int[4];
a[2] = new int[1];

for(int i=0; i<a.length; i++)
  for(int j=0; j<a[i].length; j++)
    a[i][j] = i+j;

for(int[] i : a){
  for(int j : i)
```

```
    System.out.print(j+" ");
  System.out.println();
}
```

Ausgabe

```
  0 1
  1 2 3 4
  2
```

5.4.2 Die Klasse Arrays

- **Arrays** ist eine Utility-Klasse, die ausschließlich statische Methoden besitzt

Methoden (Auswahl)

type ::=

```
  boolean | byte | char | short | int  | float | long | double | Object
```

```
public static String toString(type[] a)
```

Liefert eine String-Repräsentation des Feldes eines Typs.
Im Fall des Object-Typs ruft die Methode auf jedem Objekt im Feld toString() auf.

```
public static String deepToString(Object[] a)
```

Liefert eine String-Repräsentation des Feldes. Im Fall des Objekttyps ruft die Methode auf jedem Objekt im Feld toString() auf.

```
public static boolean equals(type[] a1, type[] a2)
```

Liefert true, falls a2 dieselbe Länge und dieselben Elemente wie a1 hat, sonst false.

```
public static void fill(type[] a, type val)
```

Füllt das Array a vollständig mit dem Wert val.

```
public static void sort(type[] a)
```

Sortiert das übergebene Array aufsteigend.

Beispiel 5.4a (weiter)

```
int[] a1 = {1,2,3};
int[] a2 = a1;
int[] a3 = {1,2,3};

System.out.println(a1.equals(a2));        // true
System.out.println(a1.equals(a3));        // false
System.out.println(Arrays.equals(a1, a3));// true
```

Beispiel 5.4a (weiter)

```
int[][] f = {{1,2},{3,4},{5,6}};

System.out.println(Arrays.toString(f));    // [[I@1888759, [I@6e1408, [I@e53108]
System.out.println(Arrays.deepToString(f));// [[1, 2], [3, 4], [5, 6]]
```

5.4.3 Sortieren von Arrays

- Array-Inhalte können mittels Arrays.sort() sortiert werden

Beispiel 5.4b

```
import java.util.Arrays;
public class Programm {
  public static void main(String[] args) {
    Integer[] array = {3,7,1,5};

    for(Integer i: array)
      System.out.print(i+" ");  // 3 7 1 5
    System.out.println();

    Arrays.sort(array);

    for(Integer i: array)
      System.out.print(i+" ");  // 1 3 5 7
    System.out.println();
  }
}
```

Beispiel 5.4c

```
public class Person {
  private String nachname;
  private String vorname;
  public Person(String n, String v){
    nachname = n;
    vorname = v;
  }
  public void print(){
    System.out.println(nachname+", "+vorname);
  }
}
public class Programm {
  public static void main(String[] args) {
    Person[] array = {new Person("Mann","Klaus"),
                       new Person("Hesse","Herrmann"),
                       new Person("Mann","Thomas")};

    for(Person elem: array)
      elem.print();      // Mann, Klaus
```

```
                              // Hesse, Herrmann
                              // Mann, Thomas

//   Arrays.sort(array);  // Person cannot be cast to java.lang.Comparable

      for(Person elem: array)
        elem.print();
   }
}
```

- Objekte einer Klasse sind vergleichbar, wenn die Klasse die Schnittstelle Comparable implementiert
- Implementiert eine Klasse die Schnittstelle Comparable, dann ist ein Array dieser Klasse sortierbar

Schnittstelle Comparable<E>

- Comparable besitzt eine einzige Methode

  `public int compareTo(E o)`

  Vergleicht ein Objekt mit o. Der Rückgabewert ist kleiner als null, null oder größer als null, je nachdem, ob dieses Objekt im Sinne der Ordnung kleiner, gleich oder größer als o ist.

- E ist ein generischer Typ
- Klassen, deren Objekte verglichen werden sollen, müssen compareTo() implementieren

Beispiel 5.4d

```
public class Person implements Comparable<Person>{
   private String nachname;
   private String vorname;
   public Person(String n, String v){
      nachname = n;
      vorname = v;
   }
   public void print(){
      System.out.println(nachname+", "+vorname);
   }
   public int compareTo(Person p){
      if(this.nachname.compareTo(p.nachname)==0)
         return this.vorname.compareTo(p.vorname);
      else
         return this.nachname.compareTo(p.nachname);
   }
}
import java.util.Arrays;
public class Programm {
   public static void main(String[] args) {
      Person[] array = {new Person("Mann","Klaus"),
                        new Person("Hesse","Herrmann"),
                        new Person("Mann","Thomas")};
```

```
    for(Person elem: array)
      elem.print();       // Mann, Klaus
                          // Hesse, Herrmann
                          // Mann, Thomas

    Arrays.sort(array);

    for(Person elem: array)
      elem.print();       // Hesse, Herrmann
                          // Mann, Klaus
                          // Mann, Thomas
  }
}
```

5.5 Aufzählungen

- Variablen eines Aufzählungstyps können nur Werte aus der Menge der Aufzählungskonstanten zugewiesen werden

Beispiel 5.5a

```
public enum Farbe {
  ROT,
  BLAU,
  GELB
}
public class Programm {
  public static void main(String[] args){
    Farbe f;
    f = Farbe.ROT;
    System.out.println(f);  // ROT
  }
}
```

interne Umsetzung der Enumeration 'Farbe'

```
public final class Farbe extends Enum{
  public static final Farbe ROT = new Farbe("ROT",0);
  public static final Farbe BLAU = new Farbe("BLAU",1);
  public static final Farbe GELB = new Farbe("GELB",2);
  private Farbe(String s, int i){
    super(s,i);
  }
}
```

- Ein Aufzählungstyp ist eine von Enum abgeleitete Klasse. Die Aufzählungskonstanten sind Referenzen auf Objekte des Aufzählungstyps. Für jede Aufzählungskonstante existiert nur ein Objekt (final)

- Ein Aufzählungstyp kann zusätzliche Felder und Methoden besitzen

Beispiel 5.5b

```
public enum Farbe {
   ROT,
   BLAU,
   GELB;
   private boolean eingeschaltet = false;
   public boolean getEingeschaltet(){
      return eingeschaltet;
   }
   public void setEingeschaltet(boolean b){
      this.eingeschaltet = b;
   }
}
public class Programm {
   public static void main(String[] args){
      Farbe Lampe1 = Farbe.BLAU;
      Farbe Lampe2 = Farbe.ROT;
      Farbe Lampe3 = Farbe.GELB;

      Lampe1.setEingeschaltet(true);
      Lampe3.setEingeschaltet(true);

      if(Lampe1.getEingeschaltet())
         System.out.println(Lampe1 + " ist an"); // BLAU ist an
      if(Lampe2.getEingeschaltet())
         System.out.println(Lampe2 + " ist an");
      if(Lampe3.gerEingeschaltet())
         System.out.println(Lampe3 + " ist an"); // GELB ist an
   }
}
```

5.6 Die Klasse Class

- **Reflektion**
  Abfrage von Typinformationen zur Laufzeit

- Die Klasse **Class** stellt Methoden zur Analyse von Datentypen einer Anwendung bereit

<u>Methode der Klasse Object</u>

```
public Class getClass()
```

Liefert ein Class-Objekt, das die Klasse dieses Objektes repräsentiert

<u>Methoden der Klasse Class</u> (Auswahl)

```
public native String getName()
```
Liefert den Namen der durch dieses Objekt repräsentierten Klasse mit dem vollen Namen des Pakets.

Die nachfolgenden Klassen Constructor, Field und Method besitzen ebenfalls eine Methode getName()

```
public Constructor[] getConstructors()
```
Liefert ein Array von Constructor-Objekten, die die als public deklarierten Konstruktoren repräsentieren.

```
public Field[] getFields()
```
Liefert ein Array von Field-Objekten, die die als public deklarierten Datenelemente repräsentieren.

```
public Method[] getMethods()
```
Liefert ein Array von Method-Objekten, die die Methoden der Klasse repräsentieren.

```
public native Class getSuperclass()
```
Liefert ein Class-Objekt für die Oberklasse der repräsentierten Klasse. Falls dieses Exemplar die Klasse Object oder ein Interface repräsentiert, wird null zurückgegeben.

Beispiel 5.6a

```
public class MyClass {
  public int intFeld;
  public float floatFeld;
  private double doubleFeld;
  public void Methode1(String a, int b){}
  public void Methode2(float c){}
}
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class Programm {
  public static void main(String[] args){
    MyClass o = new MyClass();

    System.out.println("KLASSE");
    System.out.println("  "+o.getClass().getName());

    System.out.println("FELDER");
    for(Field f : o.getClass().getFields())
      System.out.println("  "+f.getType().getName()+" "+f.getName());

    System.out.println("KONSTRUKTOREN");
    for(Constructor c : o.getClass().getConstructors())
      System.out.println("  "+c.getName());
```

```
    System.out.println("METHODEN");
    for(Method m : o.getClass().getMethods())
      System.out.println("  "+m.getName());
  }
}
```

AUSGABE

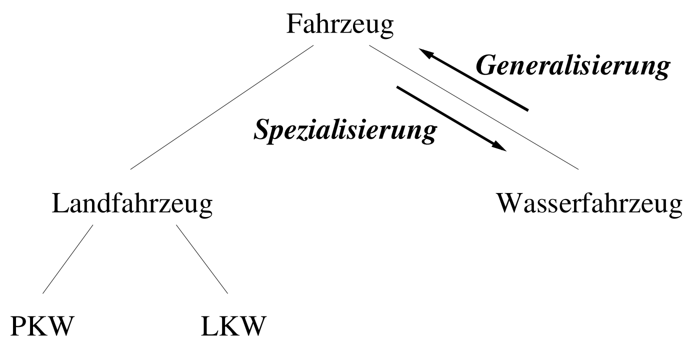
```
  KLASSE
    Beispiel.MyClass
  FELDER
    int intFeld
    float floatFeld
  KONSTRUKTOREN
    Beispiel.MyClass
  METHODEN
    Methode1
    Methode2
    getClass
    hashCode
    equals
    toString
    notify
    notifyAll
    wait
    wait
    wait
```

6 Klassen-Hierarchien

- **Klassenhierarchien** sind Beziehungen von Klassen im Sinne von Spezialisierung/Generalisierung

Beispiel

Ober- und Unterbegriffe in der Sprache



- Eine von einer Basisklasse abgeleitete Klasse **erbt** alle Member (Felder und Methoden) der Basisklasse. Zusätzlich können weitere Member definiert werden.

Beispiel

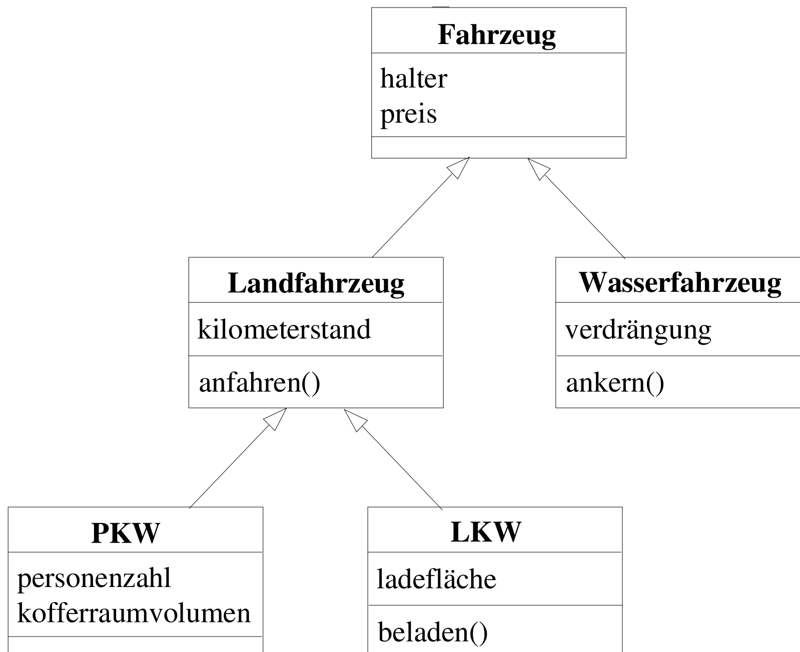
Eigenschaften von Fahrzeug (Basisbegriff)
- Halter
- Preis

Eigenschaften von Wasserfahrzeug (abgeleiteter Begriff)
- *Halter*
- *Preis*
- Verdrängung

UML: Hierarchien

- Hierarchien werden im Klassendiagramm durch einen nicht ausgefüllten Pfeil von der Unterklasse (abgeleitete Klasse) zur Oberklasse (Basisklasse) dargestellt

Beispiel



6.1 Vererbung

6.1.1 Ableitungsdeklaration

Syntax (Vervollständigung)

*ClassDeclaration** *::=*

ClassModifiers$_{opt}$ **class** *Identifier* *Extension*$_{opt}$ **{** *ClassBodyDeclarations*$_{opt}$ **}**

Extension ::=

extends *BaseClass*

* wird später weiter vervollständigt (Interfaces)

- in Java ist nur Einfachvererbung möglich
- Der Modifizierer **final** ist in der Basisklasse nicht zulässig

Beispiel

```
class Fahrzeug
{
    int halter;
    float preis;
}
```

```
class Landfahrzeug extends Fahrzeug
{
    int kilometerstand;
    anfahren(){ };
}
class PKW extends Landfahrzeug
{
    int personenzahl;
    int kofferraumVolumen;
}
```

Beispiel 6.1a (fehlerhaft)

```
public class ValueSet {
  private int count;
  private int[] values = new int[100];
  private boolean addValue(int value){..}
  public void addValues(int... values){..}
}
public class NamedValueSet extends ValueSet{
  private String name;
  public void addNamedValues(String name, int... values){
    this.name = name;
    for(int value : values)
      addValue(value);
  }
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet();
    vs.addValues(33,-6,12);

    NamedValueSet nvs = new NamedValueSet();
    nvs.addNamedValues("Menge 1", 27,33,51,2);
  }
}
```

- ein mit **protected** in einer Basisklasse deklariertes Member erlaubt den Zugriff aus allen davon abgeleiteten Klassen

Beispiel (Änderungen im vorhergehenden Beispiel)

```
public class ValueSet {
  private int count;
  private int[] values = new int[100];
->protected boolean addValue(int value){..}
  public void addValues(int... values){..}
}
```

6.1.2 Konstruktoren

- Konstruktoren einer Basisklasse werden nicht an die abgeleitete Klasse vererbt

Beispiel 6.1b (fehlerhaft)

```
public class ValueSet {
  private int count;
->protected int capacity;
->protected int[] values;
  public ValueSet(int capacity){
    this.capacity = capacity;
    this.values = new int[capacity];
  }
  private boolean addValue(int value){
    for (int i = 0; i < count; i++)
      if(values[i]==value) return false;
    values[count] = value;
    count++;
    return true;
  }
  public void addValues(int... values){
    for(int value : values)
      addValue(value);
  }
}
public class NamedValueSet extends ValueSet{
  private String name;
  public NamedValueSet(int capacity, String name){
    this.capacity = capacity;
    this.values = new int[capacity];
    this.name = name;
  }
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet(50);
    vs.addValues(33,-6,12);

    NamedValueSet nvs = new NamedValueSet(50, "Menge 1");
    nvs.addValues(27,33,51,2);
  }
}
```

Implizite Konstruktorverkettung

- **implizite Top-down-Konstruktorverkettung**
  Wird bei der Instanziierung eines Klassenobjektes einer abgeleiteten Klasse kein Konstruktor der Basisklassen explizit aufgerufen, so werden zunächst die parameterlosen Standard-Konstruktoren aller in einer Hierarchie vorausgehenden Basisklassen aufgerufen

Beispiel 6.1c

```
public class A {
   private int a;
   public A(){
      System.out.println("A() aufgerufen");
   }
}
public class B extends A{
   private int b;
   public B(){
      System.out.println("B() aufgerufen");
   }
}
public class C extends B{
   private int c;
   public C(int c){
      this.c = c;
      System.out.println("C() aufgerufen");
   }
}
public class Programm {
   public static void main(String[] args){
      C cobj = new C(3);
   }
}
```

Beispiel (Ergänzung zum vorhergehenden Beispiel)

```
public class ValueSet {
   private int count;
   protected int capacity;
   protected int[] values;
-> public ValueSet(){}
   public ValueSet(int capacity){..}
   private boolean addValue(int value){..}
   public void addValues(int... values){..}
}
```

explizite Konstruktorverkettung

- **explizite Konstruktorverkettung**
  Aufruf eines Basisklassenkonstruktoren aus einer abgeleiteten Klasse heraus mit **super**

Syntax

ExcplicitConstructorInvocations ::=

  this (*ArgumentList*$_{opt}$)
| **super** (*ArgumentList*$_{opt}$)

- Über Referenz **super** kann für einen Konstruktor die Funktionalität eines anderen, bereits definierter Konstruktors der übergeordneten Basisklasse aufgerufen werden, über die Referenz **this** die eines Konstruktors der gleichen Klasse
- Verkettungen erfolgen entsprechend der Syntax immer als erste Anweisungen eines Constructor-Blocks
- Argumente: Liste aktueller Parameter und bestimmt durch ihre Anzahl und den Datentyp den konkreten Konstruktor der Basisklasse

Beispiel 6.1d

```
public class ValueSet {
  private int count;
  protected int capacity;
  protected int[] values;
  public ValueSet(int capacity){..}
  private boolean addValue(int value){..}
  public void addValues(int... values){..}
}
public class NamedValueSet extends ValueSet{
  private String name;
  public NamedValueSet(int capacity, String name){
->  super(capacity);
    this.values = new int[capacity];
    this.name = name;
  }
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet(50);
    vs.addValues(33,-6,12);

    NamedValueSet nvs = new NamedValueSet(50, "Menge 1");
    nvs.addValues(27,33,51,2);
  }
}
```

6.1.3 Basisklassen-Member

Überladen von Basisklassen-Methoden

- **überladene Basisklassen-Methoden**

  Methode einer Basisklasse wird in einer abgeleiteten Klasse mit gleichem Namen und anderer Signatur definiert

Beispiel 6.1e

```
public class A {
  public void methode(int x){}
```

```
  }
public class B extends A{
  public void methode(int x, int y){}
  }
public class Programm {
  public static void main(String[] args){
    B bobj = new B();
    bobj.methode(3);
    bobj.methode(3,7);
  }
}
```

Überschreiben von Basisklassen-Membern

- **Überschriebene Member**
  Member (Felder, Eigenschaften und Methoden), die mit identischem Namen und Parameterliste sowohl in einer Basisklasse als auch in einer abgeleiteten Klasse definiert sind

Beispiel 6.1f

```
public class ValueSet {
->protected int count;
  protected int capacity;
  protected int[] values;
  public ValueSet(int capacity){..}
  private boolean addValue(int value){..}
  public void addValues(int... values){..}
  public void printValues(){
    for (int i = 0; i < count; i++)
      System.out.println(values[i]);
    System.out.println();
  }
}
public class NamedValueSet extends ValueSet{
  private String name;
  public NamedValueSet(int capacity, String name){..}
->public void printValues(){
    System.out.println(name+":");
    for (int i = 0; i < count; i++)
      System.out.println(values[i]);
    System.out.println();
  }
}
public class Programm {
  public static void main(String[] args){
    ValueSet vs = new ValueSet(50);
    vs.addValues(33,-6,12);
    vs.printValues();           // 33 -6 12

    NamedValueSet nvs = new NamedValueSet(50, "Menge 1");
```

```
      nvs.addValues(27,33,51,2);
      nvs.printValues();          // Menge 1: 27 33 51 2
   }
}
```

Memberzugriff mit super

- der Zugriff auf ein (überschriebenes) Member (Feld, Methode) einer Basisklasse aus einer abgeleiteten Klasse heraus erfolgt durch

  super . *FieldName*
  super . *MethodName* (*Parameters*)
- super ist eine implizite Referenz und damit an eine konkrete Klasseninstanz gebunden
- super ist sinnvoll bei Zugriff auf überschriebene Methoden

Beispiel 6.1f (modifiziert)

```
public class ValueSet {
->private int count;
   protected int capacity;
   protected int[] values;
   public ValueSet(int capacity){..}
   private boolean addValue(int value){..}
   public void addValues(int... values){..}
   public void printValues(){
      for (int i = 0; i < count; i++)
         System.out.println(values[i]);
      System.out.println();
   }
}
public class NamedValueSet extends ValueSet{
   private String name;
   public NamedValueSet(int capacity, String name){..}
   public void printValues(){
      System.out.println(name+":");
->    super.printValues();
   }
}
```

6.2 Polymorphie

Frühes Binden (statisches Binden)

Beispiel

```
public class MyClass {
   public void methode(int x){
      System.out.println("Methode(int) aufgerufen");
```

```
  }
  public void methode(double y){
    System.out.println("Methode(double) aufgerufen");
  }
}
public class Programm {
  public static void main(String[] args){
    MyClass obj1 = new MyClass();
    MyClass obj2 = new MyClass();

    obj1.methode(3);      //Methode(int) aufgerufen
    obj2.methode(3.14);   //Methode(double) aufgerufen
  }
}
```

- **Frühes Binden**
  Zuweisung von Methoden zu ihren Objekten bereits zum Zeitpunkt der Übersetzung

Beispiel 6.2a

```
public class ClassA {
  public void methode(){
    System.out.println("Methode von A aufgerufen");
  }
}
public class ClassB {
  public void methode(){
    System.out.println("Methode von B aufgerufen");
  }
}
public class Programm {
  public static void main(String[] args){
    ClassA objA = new ClassA();
    ClassB objB = new ClassB();
    objA.methode();      // Methode von A aufgerufen
    objB.methode();      // Methode von B aufgerufen
  }
}
```

Beispiel 6.2b

```
public class ClassA {
  public void methode(){
    System.out.println("Methode von A aufgerufen");
  }
}
public class ClassB extends ClassA{
  public void methodeB(){
    System.out.println("Methode von B aufgerufen");
  }
}
```

```
public class Programm {
   public static void main(String[] args){
      ClassA objA = new ClassA();
      ClassB objB = new ClassB();
      objA.methode();      // Methode von A aufgerufen
      objB.methode();      // Methode von B aufgerufen
   }
}
```

Spätes Binden (dynamisches Binden)

- **Substitutionsprinzip**
  Referenzen auf Objekte einer Basisklasse können auch Objekte einer davon abgeleiteten Klasse referenzieren.

- **Polymorpie**
  Objektreferenzen können in einer Anwendung kontextabhängig unterschiedliche Datentypen referenzieren

Beispiel 6.2c

```
public class ClassA {
   public void methode(){
      System.out.println("Methode von A aufgerufen");
   }
}
public class ClassB extends ClassA {
   public void methode(){
      System.out.println("Methode von B aufgerufen");
   }
}
public class Programm {
   public static void main(String[] args){
      ClassA obj1 = new ClassA();
->    ClassA obj2 = new ClassB();
      obj1.methode(); // Methode von A aufgerufen
      obj2.methode(); // Methode von B aufgerufen
   }
}
```

- **Spätes Binden**
  Zuweisung von Methoden zu ihren Objekten anhand ihres dynamischen Typs erst zur Laufzeit

Beispiel 6.2d

```
public class ClassA {
   public void methode(){
      System.out.println("Methode von A aufgerufen");
   }
}
```

```
public class ClassB extends ClassA {
  public void methode(){
    System.out.println("Methode von B aufgerufen");
  }
}
import java.util.Scanner;
public class Programm {
  public static void main(String[] args){
    ClassA obj = null;

    Scanner s = new Scanner(System.in);
    int ereignis = s.nextInt();      // <-- 1 oder 2

    if(ereignis==1)
      obj = new ClassA();
    if(ereignis==2)
      obj = new ClassB();
    obj.methode();
  }
}
```

Beispiel (fehlerhaft)

```
public class ClassA {
}                                    // ClassA enthält keine methode()
public class ClassB extends ClassA {
  public void methode(){
    System.out.println("Methode von B aufgerufen");
  }
}
public class Programm {
  public static void main(String[] args){
    ClassA obj = new ClassB();
    obj.methode();
  }
}
```

- Java unterstützt kein **Duck Typing**

- Der **statische Typ** einer Referenzvariablen ist der, der ihr bei ihrer Definition zugewiesen wird
- Der **dynamische Typ** einer Referenzvariablen ist der Typ des Objektes, mit dem sie instanziiert wird

- Zwei Schritte beim dynamischen Binden *var.methode(param*$_1$*, ... param*$_n$*)* :
  1. auf der Grundlage des **statischen** Typs T von *var* wird entschieden, welche Signatur der Methoden der Klasse T und ihrer Oberklassen für einen Aufruf in Frage kommen.
  2. auf der Grundlage des **dynamischen** Typs D von *var* wird überprüft, ob eine Methode mit der in 1. ermittelten Signatur in D implementiert ist. Ist das nicht der Fall, werden die Oberklassen von D untersucht, bis eine entsprechende Methode gefunden ist, die dann ausgeführt wird

Beispiel 6.2e

```
public class ClassA {
   public void methode(){
      System.out.println("Methode von A aufgerufen");
   }
}
public class ClassB extends ClassA {
}
public class ClassC extends ClassB {
   public void methode(){
      System.out.println("Methode von C aufgerufen");
   }
}
public class ClassD extends ClassC {
}
public class Programm {
   public static void main(String[] args){
      ClassB obj = new ClassD();
      obj.Methode();              // Methode von C aufgerufen
   }
}
```

- Polymorphie hat ihre besondere Bedeutung bei sog. inhomogenen Kollektionen

Beispiel 6.2f

```
public class ValueSet {
   protected int count;
   protected int capacity;
   public boolean addValue(int value){ return false; }
   public boolean addValue(String value){ return false; }
   public void printValues(){ }
}
public class IntValueSet extends ValueSet {
   private int[] values;
   public IntValueSet(int capacity){
      super.capacity = capacity;
      this.values = new int[capacity];
   }
   public boolean addValue(int value){
      for (int i = 0; i < super.count; i++)
         if(values[i]==value) return false;
      values[super.count] = value;
      super.count++;
      return true;
   }
   public void printValues(){
      for (int i = 0; i < super.count; i++)
         System.out.println(values[i]);
      System.out.println();
   }
}
public class StringValueSet extends ValueSet {
```

```
   private String[] values;
   public StringValueSet(int capacity){
      super.capacity = capacity;
      this.values = new String[capacity];
   }
   public boolean addValue(String value){
      for (int i = 0; i < super.count; i++)
         if(values[i]==value) return false;
      values[super.count] = value;
      super.count++;
      return true;
   }
   public void printValues(){
      for (int i = 0; i < count; i++)
         System.out.println(values[i]);
      System.out.println();
   }
}
public class Programm {
   public static void main(String[] args){
      ValueSet[] vs = new ValueSet[5];

      vs[0] = new IntValueSet(20);
      vs[0].addValue(3);
      vs[0].addValue(-6);
      vs[0].addValue(15);

      vs[1] = new StringValueSet(50);
      vs[1].addValue("Anna");
      vs[1].addValue("Hannes");
      vs[1].addValue("Willi");

      for(ValueSet v : vs)
         if(v!=null)
            v.printValues();
   }
}
```

6.3 Abstrakte Klassen und Methoden

Abstrakte Klasse

- Klasse, von der keine Instanzen erzeugt werden können
- Abstrakte Klassen werden durch den Modifizierer **abstract** gekennzeichnet
- eine abstrakte Klasse kann abstrakte Methoden enthalten

Abstrakte Methode

- Methodendeklaration, die keinen Anweisungsblock besitzt und durch den Modifizierer **abstract** gekennzeichnet ist
- eine von einer abstrakten Basisklasse abgeleitete Klasse muß

   - alle abstrakten Methoden der Basisklasse überschreiben oder
   - selbst als abstrakte Klasse mit **abstract** gekennzeichnet werden
- eine Methode, die eine abstrakte Methode einer Basisklasse überschreibt, zeigt polymorphes Verhalten

Beispiel 6.3a

```
public abstract class ValueSet {
   protected int count;
   protected int capacity;
   public abstract boolean addValue(int value);
   public abstract boolean addValue(String value);
   public abstract void printValues();
}
public class IntValueSet extends ValueSet {
   private int[] values;
   public IntValueSet(int capacity){..}
   public boolean addValue(int value){..}
   public void printValues(){..}
public class StringValueSet extends ValueSet {
   private String[] values;
   public StringValueSet(int capacity){..}
   public boolean addValue(String value){..}
   public void printValues(){..}
}
public class Programm {
   public static void main(String[] args){..}
}
```

Beispiel 6.3b

```
public abstract class ValueSet {
   protected int count;
   protected int capacity;
-> public abstract boolean addValue(Object value);
   public abstract void printValues();
}
public class IntValueSet extends ValueSet {
   private int[] values;
   public IntValueSet(int capacity){..}
-> public boolean addValue(Object value){
->    if(!(value instanceof Integer))
->          System.out.println("Fehler");    // steht für Exception
      for (int i = 0; i < super.count; i++)
        if(values[i]==(int)value) return false;
      values[super.count] = (int)value;
      super.count++;
      return true;
   }
   public void printValues(){..}
}
public class StringValueSet extends ValueSet {
```

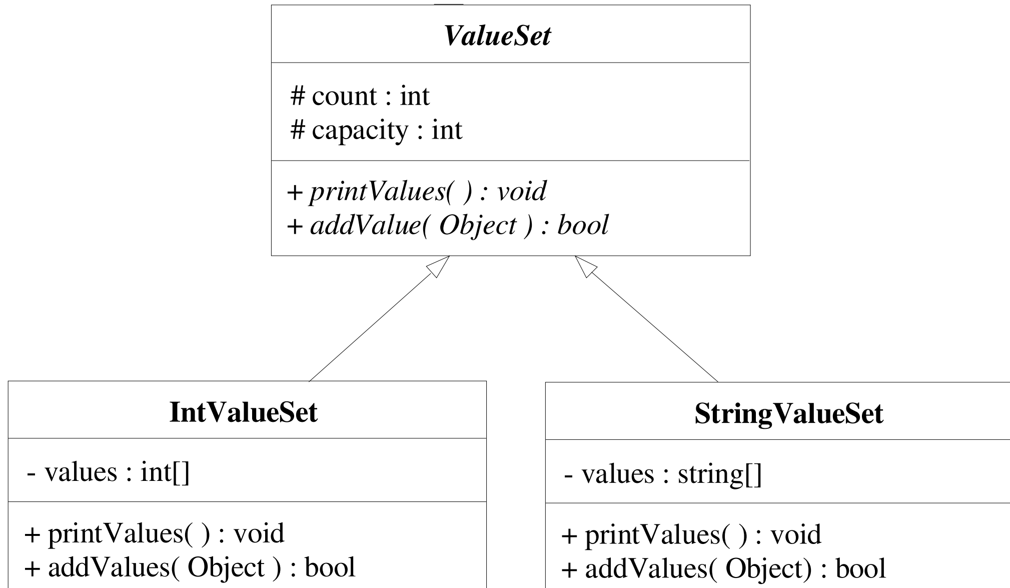
```
   private String[] values;
   public StringValueSet(int capacity){..}
-> public boolean addValue(Object value){
->   if(!(value instanceof String))
->     System.out.println("Fehler");          // steht für Exception
     for (int i = 0; i < super.count; i++)
       if(values[i]==(String)value) return false;
     values[super.count] = (String)value;
     super.count++;
     return true;
   }
   public void printValues(){..}
}
public class Programm {
   public static void main(String[] args){..}
}
```

- besitzt eine Klasse eine abstrakte Methode, dann muß sie selbst als abstract definiert werden

UML: Abstrakte Klassen

- Abstrakte Klasse/Methoden werden im Klassendiagramm kursiv dargestellt

Beispiel



6.4 Schnittstellen

- Schnittstellen (interfaces) dienen der Deklaration von Methodenschemata. Klassen, die diese Schnittstellen implementieren, müssen die in den Schnittstellen deklarierten Methoden definieren (enthalten)
- Schnittstellen sind Sprachmittel für den Entwurf einer Anwendung

6.4.1 Schnittstellendeklaration

Syntax

InterfaceDeclaration ::=

public$_{opt}$ **interface** *Identifier Extension*$_{opt}$ **{** *InterfaceBodyDeclarations*$_{opt}$ **}**

Extension ::=

extends *BaseInterfaces*

- *BaseInterfaces*: eine Schnittstelle kann von mehreren (kommagetrennten) Basisschnittstellen abgeleitet werden: die Schnittstelle erbt dann alle Memberdeklarationen der Basisschnittstellen

- Der Name einer Schnittstelle endet oft auf -ble (Accessible, Adjustable, Runnable).

Syntax

InterfaceBodyDeclarations ::=

ConstantDeclaration
| *MethodHeader*
| *InterfaceBodyDeclarations InterfaceBodyDeclarations*

- *ConstantDeclaration*
  - Konstante sind automatisch **public**, **static** und **final**

- *MethodHeader*
  - es wird nur der Methodenkopf einer Methode definiert
  - Methoden sind automatisch **public** und **abstract**

- Member einer Schnittstelle sind die von den Basisschnittstellen vererbten Member und die in der Schnittstelle selbst deklarierten Member.

- Schnittstellenmethoden enthalten keine Anweisungsblöcke

6.4.2 Schnittstellenimplementierung

- Schnittstellen können von Klassen implementiert werden.

Syntax (Vervollständigung)

ClassDeclaration ::=

ClassModifiers$_{opt}$ **class** *Identifier Extension*$_{opt}$ *Implementations*$_{opt}$
{ *ClassBodyDeclarations*$_{opt}$ }

Extension ::=

extends *BaseClass*

Implementations ::=

implements *Interfaces*

- Eine Klasse kann mehrere (kommagetrennten) Schnittstellen implementieren
- Klassen, die eine oder mehrere Schnittstellen implementieren, müssen alle in den Schnittstellen deklarierten Methoden definieren, es sei denn, daß die Klasse selbst abstract ist
- die von einer Schnittstelle übernommenen und in der Klasse definierten Methoden
  - müssen in Name, Parametern und Rückgabetyp mit der Schnittstellenmethode übereinstimmen
  - dürfen nur public implementiert werden
  - zeigen polymorphes Verhalten

Beispiel 6.4a

```
public interface IFliegen {
   void start();
}
public interface IKraftstoff {
   void tanken(int menge);
}
public class MotorFlieger implements IFliegen, IKraftstoff {
   public void start(){
      System.out.println("Motorflieger startet");
   }
   public void tanken(int menge){
      System.out.println("Motorflieger tankt "+menge+" Liter");
   }
}
public class SegelFlieger implements IFliegen{
   public void start(){
      System.out.println("Segelflieger startet");
```

```
   }
}
public class Programm {
   public static void main(String[] args) {
      MotorFlieger m = new MotorFlieger();
      SegelFlieger s = new SegelFlieger();
      m.tanken(200);   // Motorflieger tankt 200 Liter
      m.start();       // Motorflieger startet
      s.start();       // Segelflieger startet
   }
}
```

- innerhalb einer Vererbungshierarchie werden Schnittstellen vererbt

Beispiel 6.4b

```
public interface IGegenstand {
   String getBesitzer();
   void setBesitzer(String besitzer);
}
public interface IFliegen {
   void start();
}
public class Luftfahrzeug implements IGegenstand, IFliegen {
   private String besitzer;
   public String getBesitzer(){
      return this.besitzer;
   }
   public void setBesitzer(String besitzer){
      this.besitzer = besitzer;
   }
   public void start(){
      System.out.println("Flieger von "+besitzer+" startet");
   }
}
public class MotorFlieger extends Luftfahrzeug{
}
public class SegelFlieger extends Luftfahrzeug{
}
public class Programm {
   public static void main(String[] args) {
      Luftfahrzeug[] lf = new Luftfahrzeug[10];
      lf[0] = new MotorFlieger();
      lf[0].setBesitzer("Müller");
      lf[1] = new SegelFlieger();
      lf[1].setBesitzer("Meier");
      lf[0].start();   // Flieger von Müller startet
      lf[1].start();   // Flieger von Meier startet
   }
}
```

- Werden Schnittstellen-Member einer Basisklasse in den abgeleiteten Klassen überschrieben, zeigen sie dort polymorphes Verhalten

Beispiel 6.4c

```
public interface IGegenstand {
  String getBesitzer();
  void setBesitzer(String besitzer);
}
public interface IFliegen {
  void start();
}
public class Luftfahrzeug implements IGegenstand, IFliegen {
  protected String besitzer;
  public String getBesitzer(){
    return this.besitzer;
  }
  public void setBesitzer(String besitzer){
    this.besitzer = besitzer;
  }
  public void start(){
    System.out.println("Flieger von "+besitzer+" startet");
  }
}
public class MotorFlieger extends Luftfahrzeug{
  public void start(){
    System.out.println("Motorflieger von "+besitzer+" startet");
  }
}
public class SegelFlieger extends Luftfahrzeug{
  public void start(){
    System.out.println("Segelflieger von "+besitzer+" startet");
  }
}
public class Programm {
  public static void main(String[] args) {
    Luftfahrzeug[] lf = new Luftfahrzeug[10];
    lf[0] = new MotorFlieger();
    lf[0].setBesitzer("Müller");
    lf[1] = new SegelFlieger();
    lf[1].setBesitzer("Meier");
    lf[2] = new Luftfahrzeug();
    lf[2].setBesitzer("Schulz");
    lf[0].start();   // Motorflieger von Müller startet
    lf[1].start();   // Segelflieger von Meier startet
    lf[2].start();   // Flieger von Schulz startet
  }
}
```

- Abstrakte Kassen, die Schnittstellen implementieren, können deren Methoden abstrakt definieren

Beispiel 6.4d (vgl. 6.3b)

```
public interface Printable {
   void printValues();
}
public interface Insertable {
   boolean addValue(Object value);
}
public abstract class ValueSet implements Printable, Insertable{
   protected int count;
   protected int capacity;
->public abstract boolean addValue(Object value);
->public abstract void printValues();
}
public class IntValueSet extends ValueSet {
   private int[] values;
   public IntValueSet(int capacity){..}
   public boolean addValue(Object value){..}
   public void printValues(){..}
}
public class StringValueSet extends ValueSet {
   private String[] values;
   public StringValueSet(int capacity){..}
   public boolean addValue(Object value){..}
   public void printValues(){..}
}
public class Programm {
   public static void main(String[] args){
      ValueSet[] vs = new ValueSet[5];
      vs[0] = new IntValueSet(20);
      vs[0].addValue(3);
      vs[0].addValue(-6);
      vs[0].addValue(15);
      vs[1] = new StringValueSet(50);
      vs[1].addValue("Anna");
      vs[1].addValue("Hannes");
      vs[1].addValue("Willi");
      for(ValueSet v : vs)
         if(v!=null)
            v.printValues();
   }
}
```

Schnittstellen als Datentyp

- Schnittstellen sind vom Referenztyp, von denen Referenzvariablen gebildet werden können
- Referenzvariablen vom Schnittstellentyp können auf Objekte verweisen, deren Klassen diese Schnittstellen implementieren

Beispiel 6.4e

```java
public interface MyInterface {
   void method1();
}
public class MyClass implements MyInterface{
   public void method1(){
      System.out.println("Methode 1 aufgerufen");
   }
   public void method2(){
      System.out.println("Methode 2 aufgerufen");
   }
}
public class Programm {
   public static void main(String[] args){
      MyClass obj1 = new MyClass();
      obj1.method1();    // Methode 1 aufgerufen
      obj1.method2();    // Methode 2 aufgerufen

      MyInterface obj2 = new MyClass();
      obj2.method1();    // Methode 1 aufgerufen
      obj2.method2();    // Fehler
   }
}
```

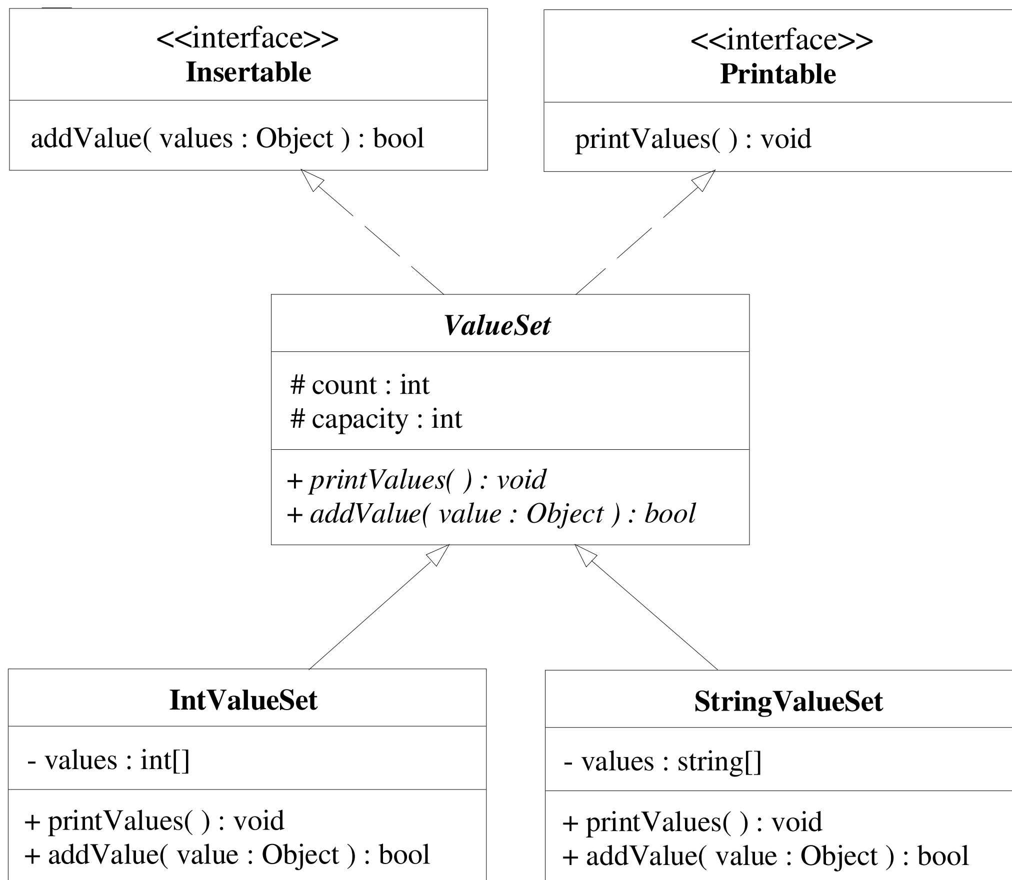
- Über Referenzvariablen vom Schnittstellentyp, die auf Objekte einer Klasse verweisen, können nur die Methoden der Schnittstelle aufgerufen werden.

UML: Schnittstellen

- Schnittstellen werden durch den Stereotyp <<interface>> über dem Schnittstellennamen gekennzeichnet

⇒ Stereotyp: in << >> eingeschlossene Bezeichner, durch die unterschiedliche Verwendungen von Modellelementen gekennzeichnet werden können

- die Implementation einer Schnittstelle wird durch einen nicht ausgefüllten Pfeil mit gestrichelter Linie dargestellt


<<interface>>
Insertable
addValue(values : Object) : bool
<<interface>>
Printable
printValues( ) : void
ValueSet
count : int
capacity : int
+ printValues( ) : void
+ addValue(value : Object) : bool
IntValueSet
- values : int[]
+ printValues( ) : void
+ addValue(value : Object) : bool
StringValueSet
- values : string[]
+ printValues( ) : void
+ addValue(value : Object) : bool

7 Klassenbeziehungen

7.1 Assoziationen

Beispiel 7.1.a

```
public class Person {
   private String name;
   private String wohnort;
   private String strasse;
   private int nr;
   public Person(String name, String wohnort, String strasse, int nr){
      this.name = name;
      this.wohnort = wohnort;
      this.strasse = strasse;
      this.nr = nr;
   }
}
public class Programm {
   public static void main(String[] args) {
      Person p1 = new Person("Cindy Meier", "HRO", "Bergstr.", 3);
      Person p2 = new Person("Bert Meier", "HRO", "Bergstr.", 3);
   }
}
```

Beispiel 7.1.b

```
public class Adresse {
   private String ort;
   private String strasse;
   private int nr;
   public Adresse(String ort, String strasse, int nr){
      this.ort = ort;
      this.strasse = strasse;
      this.nr = nr;
   }
}
public class Person {
   private String name;
   private Adresse wohnsitz;
   public Person(String name, Adresse wohnsitz){
      this.name = name;
      this.wohnsitz = wohnsitz;
   }
}
public class Programm {
   public static void main(String[] args) {
      Adresse a1 =  new Adresse("HRO", "Bergstr.", 3);
      Adresse a2 =  new Adresse("M", "Parkstr.", 1);
      Person p1 = new Person("Cindy Meier", a1);
      Person p2 = new Person("Bert Meier", a1);
      Person p3 = new Person("Josef Motzen", a2);
   }
}
```

UML: Assoziation

- **Assoziation**
  - Beziehung zwischen Klassen
  - wird im UML-Klassendiagramm durch Linie zwischen den assoziierten Klassen angegeben

- **Assoziationsrichtung**
  - Richtung, aus der ein Zugriff von Objekten der einen Klasse (source) auf Objekte der anderen Klasse (target) erfolgen kann
  - eine Assoziation kann unidirektional oder bidirektional sein
  - Assoziationsrichtungen werden durch Pfeile am Ende der verbindenden Linie in Richtung von der source-Klasse zur target-Klasse angegeben
  - keine Pfeile oder Pfeile in beide Richtungen: bidirektionale Assoziation

- **Kardinalität (Multiplizität)**

  gibt als Zahl n/* oder Bereich n .. m an, wieviel Objekte der einen Klasse mit wieviel Objekten der andren assoziiert sein können/müssen

  z.B.

  0 .. 1 - kann mit keinem oder einem Objekten assoziiert sein
  1 - muß mit genau einem Objekt assoziiert sein
  3 .. 4 - muß mit mindestens 3 und kann mit bis zu 4 Objekten assoziiert sein
  * - kann mit keinem oder beliebig vielen Objekten assoziiert sein (synonym: 0 .. *)
  1 .. * - muß mit einem und kann mit bis zu beliebig vielen Objekten assoziiert sein

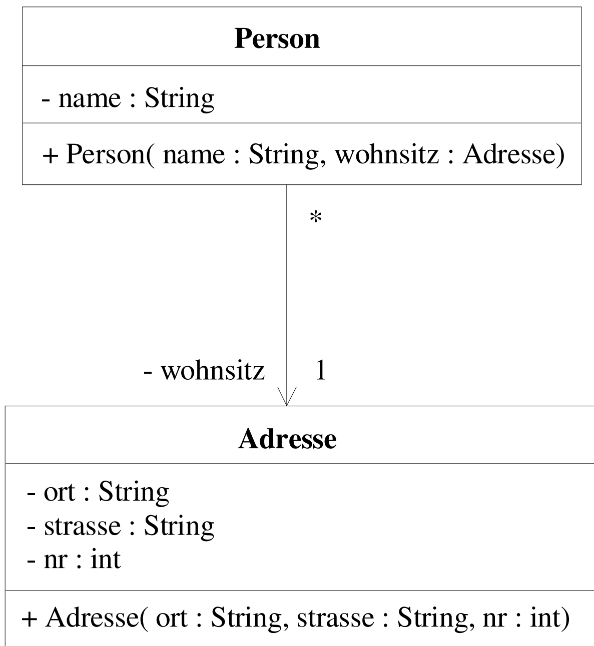
- **Rolle**

  Bezeichnung, unter der ein Objekt der einen Klasse mit einem Objekt der anderen Klasse assoziiert wird

- **Diagramm**

  - ein Objekt der Klasse 2 tritt in der Rolle 2 in der Klasse 1 auf und umgekehrt
  - ein Objekt der Klasse 1 steht mit card2 Objekten der Klasse 2 in Beziehung und umgekehrt

Beispiel



UML: Aggregation und Komposition

- Aggregation und Komposition bilden Spezialfälle der Assoziation
- **Aggregation**
  - Assoziation, bei der Objekte der target-Klasse Teile eines Objektes der source-Klassen bilden
  - wird im Klassendiagramm durch nicht ausgefüllte Raute an der source-Klasse dargestellt

- **Komposition**
  - Assoziation, bei der Objekte der target-Klasse existenzabhängig von einem Objekt der source-Klassen sind
  - wird im Klassendiagramm durch ausgefüllte Raute an der source-Klasse dargestellt

- mögliche Implementationen von **Assoziation/Aggregationen**:
  - unidirektional
    - source-Klasse enthält Referenz auf target-Klasse

```
class SourceClass {
    TargetClass targetObject;
}
```

  - bidirektional
    - Klassen referenzieren sich gegenseitig
    - (beide Klassen sowohl source als auch target)

```
class Class1 {
    Class2  class2Object;
}
class Class2 {
    Class1 class1Object;
}
```

    - Problem: Integritätserhaltung bei Änderungen

- mögliche Implementationen von **Compositionen**:
  - source-Klasse enthält Objekterzeugung der target-Klasse

```
class SourceClass {
    TargetClass targetObject = new TargetClass();
}
```

  - source-Klasse kann auch Definition der Target-Klasse enthalten (innere Klasse)

```
class SourceClass {
    class TargetClass {
    }
}
```

Beispiel 7.1c

```
public class Spieler {
   public String name;
   public int punkte = 0;
   public Spieler(String name) {
```

```
      this.name = name;
   }
   public void gewinn(int punkte) {
      this.punkte+=punkte;
   }
}
public class Programm {
   public static void main(String[] args) {
      Spieler[] runde = new Spieler[3];
      runde[0] = new Spieler("Meier");
      runde[1] = new Spieler("Motzen");
      runde[2] = new Spieler("Schmidt");

      runde[1].gewinn(24);

      for(int i=0; i<3; i++)
         System.out.println(runde[i].name+": "+runde[i].punkte);
   }
}
```

Ausgabe

```
Meier: 0
Motzen: 24
Schmidt: 0
```

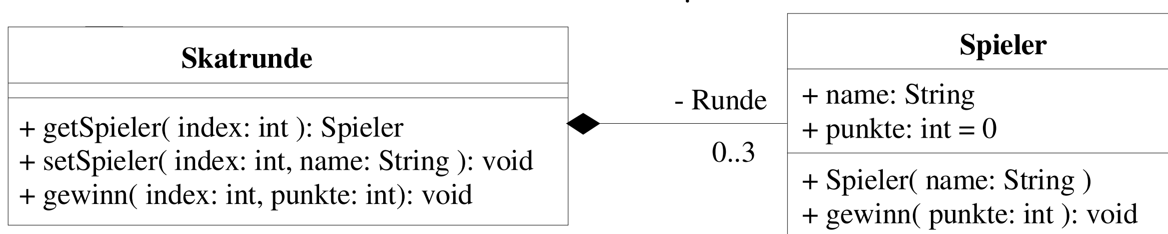
Beispiel 7.1d

```java
public class Spieler {
   public String name;
   public int punkte = 0;
   public Spieler(String name) {
      this.name = name;
   }
   public void gewinn(int punkte) {
      this.punkte+=punkte;
   }
}
public class Skatrunde{
   private Spieler[] runde = new Spieler[3];
   public Spieler getSpieler(int index){
      return runde[index];
   }
   public void setSpieler(int index, String name){
      runde[index] = new Spieler(name);
   }
   public void gewinn(int index, int punkte){
      runde[index].gewinn(punkte);
   }
}
public class Programm {
   public static void main(String[] args) {
      Skatrunde runde = new Skatrunde();
      runde.setSpieler(0, "Meier");
      runde.setSpieler(1, "Motzen");
      runde.setSpieler(2, "Schmidt");

      runde.gewinn(1, 24);

      for (int i = 0; i < 3; i++)
         System.out.println(runde.getSpieler(i).name+
                                 ": "+runde.getSpieler(i).punkte);
   }
}
```

Ausgabe

```
   Meier: 0
   Motzen: 24
   Schmidt: 0
```

- UML-Klassendiagramm



7.2 Iteratoren

- Die foreach-Anweisung erwartet rechts vom Doppelpunkt ein Objekt einer Klasse, die das generische Interface **Iterable** implementiert und damit die Methode **iterator()** besitzt, die einen **Iterator** erzeugt

Interface Iterable<T>

Methode

```
public Iterator<T> iterator()
```

Liefert einen Iterator, der über alle Elemente vom Typ T iteriert.

- Ein Iterator ist ein Objekt einer Klasse, die das generische Interface **Iterator** implementiert

Interface Iterator<T>

Methoden

```
public boolean hasNext()
```

Liefert true, falls noch weitere Elemente verfügbar sind, sonst false.

```
public Object next()
```

Liefert das nächste Element zurück.

```
public void remove()
```

Löscht das zuletzt mit next() abgerufene Element aus der Kollektion (optional). Diese Methode darf nur einmal nach jedem next()-Aufruf aufgerufen werden.

Vorgehensweise

Um eine eigene Klasse MyClass, die eine Collection (im einfachsten Fall ein Array) von Objekten besitzt in einer foreach-Anweisung zu verwenden, d.h. über diese Objekt-Collection zu iterieren, muß:

1. MyClass das Interface Iterable implementieren und damit über eine Methode iterator() verfügen
2. die Methode iterator() ein Iterator-Objekt vom Typ MyIterator zurückgeben
3. Eine Klasse MyIterator existieren, die das Interface Iterator implementiert und über MyClass-spezifische Methoden hasNext(), next() und remove() verfügt.

Beispiel 7.2a

```
public class Spieler{
  public String name;
  public int punkte = 0;
```

```
  public Spieler(String name){
    this.name = name;
  }
  public void gewinn(int punkte){
    this.punkte+=punkte;
  }
}

public class Skatrunde implements Iterable<Spieler>{
  private Spieler[] runde = new Spieler[3];
  public Spieler getSpieler(int index){
    return runde[index];
  }
  public void setSpieler(int index, String name){
    runde[index] = new Spieler(name);
  }
  public void gewinn(int index, int punkte){
    runde[index].gewinn(punkte);
  }
  public SkatrundeIterator iterator() {
    SkatrundeIterator iter = new SkatrundeIterator();
    iter.setData(this.runde);
    return iter;
  }
}

import java.util.Iterator;
public class SkatrundeIterator implements Iterator<Spieler>{
  private int pos = -1;
  private Spieler[] arr = null;
  public void setData(Spieler[] elements){
    this.arr = elements;
  }
  public boolean hasNext() {
    return this.pos+1 < this.arr.length;
  }
  public Spieler next() {
    return this.arr[++this.pos];
  }
  public void remove() {} // nicht implementiert
}

public class Programm {
  public static void main(String[] args){
    Skatrunde runde = new Skatrunde();
    runde.setSpieler(0, "Meier");
    runde.setSpieler(1, "Motzen");
    runde.setSpieler(2, "Schmidt");

    runde.gewinn(1, 24);

    for(Spieler sp : runde)
      System.out.println(sp.name+": "+sp.punkte);
  }
}
```

8 Exceptions

- klassische Fehlerbehandlung: return-Code

Beispiel

```
int Modulo(int x, int y)
{
    if (x>=0)&&(y != 0)
        return x % y;
    else
        return -1;
}
main()
{
    ...
    if (erg=Modulo(a,b) != -1)
        printf("%f",erg)
    else
        printf("Fehler: Division durch 0 oder x ist negativ");
    ...
}
```

Beispiel

```
int Modulo(int x, int y)
{
    if (y != 0)
        return x % y;
    else
        return ???;
}
```

Exception-Mechanismus

- Exception (dt. Ausnahme):
    a. Laufzeitfehler oder
    b. vom Entwickler gewollt ausgelöste Ausnahme

  der/die ein definiertes Ausnahmeobjekt (der Klasse Throwable) erzeugt, das die speziellen Ausnahmeinformationen kapselt

- Ablauf nach Auslösen ("Werfen der Exception") einer Exception (z.B. Division durch 0):
  - das System sucht in der Methode, die die Exception ausgelöst hat, nach einer Exception-Behandlungsroutine
  - existiert diese, so wir sie ausgeführt (Behandelte Ausnahme, Fangen einer Exception) und danach das Programm weiter ausgeführt
  - existiert diese nicht, so wird die Methode beendet und beim Aufrufer der Methode nach einer Behandlungroutine gesucht
  - wird in der gesamten Aufrufhierarchie keine Behandlungsroutine gefunden, so wird die Exception vom System gefangen und das Programm mit einer Fehlernachricht beendet (Unbehandelte Ausnahme)

8.1 Unbehandelte Ausnahmen

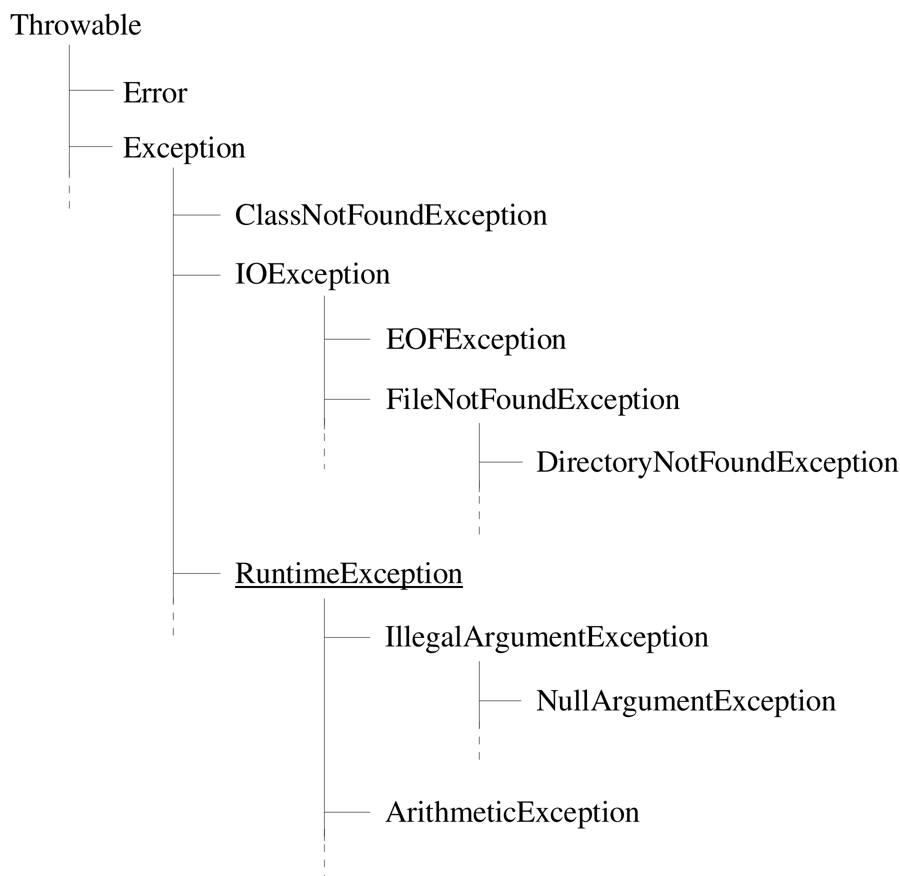
Beispiel 8.1a

```
1   package Beispiel;
2   public class Programm {
3      public static int modulo(int x, int y) {
4         return x%y;
5      }
6      public static void main(String[] args) {
7         int a = 22;
8         int b = 0;
9         modulo(a,b);
10     }
11  }
```

Fehlernachricht

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at Beispiel.Programm.modulo(Programm.java:4)
   at Beispiel.Programm.main(Programm.java:9)
```

Ausnahme-Hierarchie



- Klasse **Throwable**
  Basisklasse für Objekte, die Ausnahmesituationen signalisieren, den Errors und Exceptions

- Klasse **Error**
  Repräsentiert die schwersten Fehler im System. Subklassen dieser Kategorie sollten nicht selber ausgelöst werden. In der Regel wird bei diesen Fehlern das Programm vollständig beendet.

- Klasse **Exception**
  Stellt eine Basis für Exception-Objekte dar, die in Ausnahmesituationen erzeugt werden. Exceptions werden in Fehlerfällen ausgelöst, wobei ein Programm die Möglichkeit hat, über ein try/catch-Statement auf diese Exceptions zu reagieren. Alle benutzerdefinierten Ausnahmen sollten von dieser Klasse ableiten.

- Klasse **RuntimeException**
  Laufzeitfehler, die während des Programmverlaufs auftreten. Die Kategorie vereint alle Fehler, die auf Grund eines Programmierfehlers auftreten können, wie unter anderem:
  - fehlerhafte Typkonvertierung
  - Zugriff über die Arraygrenze hinaus
  - Zugriff auf einen leeren Zeiger

- Alle nicht von RuntimeException abgeleiteten Exception-Klassen repräsentieren Fehler, die sich durch ungünstige Umstände und nicht etwa durch fehlerhaften Code ergeben, wie unter anderem:
  - Fehler beim Lesen einer Datei
  - ungenügend Systemressourcen
  - Fehlfunktion externer Dienste alle anderen Exception-Klassen

Weitergabe von Exceptions mit throws

- Alle Exceptions, die nicht von RuntimeException abgeleitet sind, müssen an der Stelle, an der sie auftreten behandelt werden oder explizit an den Aufrufer weitergegeben werden

Syntax (vervollständigt)

MethodDeclaration ::=

  MethodHeader MethodBlock

MethodHead ::=

  MethodModifiers$_{opt}$ *ResultType Identifier* (*FormalParameterList*$_{opt}$) *Throws*$_{opt}$

Throws ::=

  throws *ExceptionTypeList*

MethodBlock ::=

  { *StatementsOrDeclarations*$_{opt}$ }

- mit **throws** *ExceptionTypeList* werden Exceptions, die in der Methode auftreten können, an den Aufrufer der Methode weitergeleitet

Beispiel 8.1b

```
import java.io.*;
public class Programm {
   public static void main(String[] args) throws IOException {
      FileWriter fw = new FileWriter("c:\\test\\Datei.txt");
      fw.write("Hello World");
      fw.close();
   }
}
```

8.2 Behandelte Ausnahmen

- 2 Schritte bei Ausnahmebehandlung
  1. Markierung des Codeabschnitts der eine Exception werfen kann (try-Block)
  2. Fangen und Behandlung der Exception (catch-Block)

Syntax (vereinfacht)

TryStatement ::=

TryClause CatchClauses FinallyClause$_{opt}$

TryClause ::=

try { StatementsOrDeclarations$_{opt}$ }

CatchClauses ::=

CatchClause
| *CatchClause CatchClause*

CatchClause ::=

catch (ExceptionClass) { StatementsOrDeclarations$_{opt}$ }

FinallyClause ::=

finally { StatementsOrDeclarations$_{opt}$ }

.

- Abarbeitung der try-catch-Anweisung:
  1. **Werfen**
     wirft eine Anweisung im try-Block eine Exception eines bestimmten Typs, so wird die Abarbeitung des try-Blockes sofort beendet, nach einem catch-Block für diesen Exception-Typ gesucht
  2. **Fangen**
     existiert ein solcher Block, so wird dieser abgearbeitet
  3. **Aufräumen**
     falls finally-Block existiert, wird dieser danach abgearbeitet

- wurde die Ausnahme gefangen wird nach der try-catch-Anweisung weitergemacht
- wurde sie nicht gefangen so wird wie bei unbehandelten Ausnahmen verfahren

Beispiel 8.2a

```
import java.io.*;
public class Programm {
  public static void ausgabe(String out, String s) throws IOException {
->  FileWriter fw = new FileWriter(out);
    fw.write(s);
    fw.close();
  }
  public static void main(String[] args) throws IOException {
      String datei = "c:\\teste\\Datei.txt"; // Fehler: ungültiges Verzeichnis
->    ausgabe(datei, "Hello World");
  }
}
```

Laufzeit-Fehler

```
Exception in thread "main" java.io.FileNotFoundException: c:\teste\Datei.txt
(Das System kann den angegebenen Pfad nicht finden)
  at java.io.FileOutputStream.open(Native Method)
  at java.io.FileOutputStream.<init>(Unknown Source)
  at java.io.FileOutputStream.<init>(Unknown Source)
  at java.io.FileWriter.<init>(Unknown Source)
  at Beispiel.Programm.ausgabe(Programm.java:6)
  at Beispiel.Programm.main(Programm.java:12)
```

Klasse Throwable

Basisklasse für Objekte, die Ausnahmesituationen signalisieren, den Errors und Exceptions

Konstruktoren (Auszug)

`public Throwable()`
  Erzeugt ein neues Throwable-Objekt ohne Angabe einer Fehlermeldung.

`public Throwable(String s)`
  Erzeugt ein neues Throwable-Objekt mit der Fehlermeldung s, die nähere Informationen über den speziellen Fehlerfall enthält.

Methoden

`public String getMessage()`
Liefert Informationen über den speziellen Fehlerfall.

`public StackTraceElement[] getStackTrace()`
Liefert den derzeitigen Aufruf-Stack zurück.
Die Klasse StackTraceElement enthält Methoden zur Ausgabe der ausnahmeerzeugenden Klasse, Methode, Zeilennummer im Code ...

`public void printStackTrace()`
Gibt die Fehlermeldung und die dynamische Aufrufhierarchie auf der Standardfehlerausgabe aus.

Beispiel 8.2b

```
import java.io.*;
public class Programm {
->public static void ausgabe(String out, String s) throws IOException {
    FileWriter fw = new FileWriter(out);
    fw.write(s);
    fw.close();
  }
  public static void main(String[] args) {
    try {
      String datei = "c:\\teste\\Datei.txt"; // Fehler: ungültiges Verzeichnis
->    ausgabe(datei, "Hello World");
    }
    catch (IOException e) {
      System.out.println("Fehlerursache");
      System.out.println("  "+e.getMessage());
      System.out.println("Aufrufreihenfolge");
      for(StackTraceElement st: e.getStackTrace())
        System.out.println("  "+st.getMethodName()+
                           "  bei Zeile  "+st.getLineNumber());
    }
    finally {
      System.out.println("DAS WAR'S");
    }
    System.out.println("UND HIER GEHT ES SCHON WEITER");
  }
}
```

Laufzeit-Fehler (Ausgabe)

```
Fehlerursache
  c:\teste\Datei.txt (Das System kann den angegebenen Pfad nicht finden)
Aufrufreihenfolge
  open  bei Zeile  -2
  <init>  bei Zeile  -1
  <init>  bei Zeile  -1
  <init>  bei Zeile  -1
  ausgabe  bei Zeile  6
```

```
  main  bei Zeile  14
DAS WAR'S
UND HIER GEHT ES SCHON WEITER
```

8.3 Ausnahmen werfen

- soll ein Zustand, der vom System nicht als Ausnahme interpretiert wird, als Ausnahme behandelt werden, so muß die Exception explizit mit der **throw**-Anweisung geworfen werden

Syntax

ThrowStatement ::=

throw *Expression*

- *Expression*: mit new erzeugtes Objekt einer existierenden Exception-Klasse
- es existieren mehrere überladene Konstruktoren der Exception-Klassen, u.a. für die Basisklasse und entsprechend für die abgeleiteten Klassen:

Beispiel 8.3a

```
import java.io.*;
public class Programm {
  public static void ausgabe(String out, String s) throws IOException,
                                                           EOFException {
    if(s=="")
      throw new EOFException();    // Exception wird geworfen
    FileWriter fw = new FileWriter(out);
    fw.write(s);
    fw.close();
  }
  public static void main(String[] args) {
    try {
      String datei = "c:\\test\\Datei.txt";
      ausgabe(datei, "");          // Fehler: leerer String
    }
    catch (EOFException e) {
      System.out.println("Fehlerursache");
      System.out.println("  Leere Datei");
    }
    catch (IOException e) {
      System.out.println("Fehlerursache");
      System.out.println("  "+e.getMessage());
    }
    finally {
      System.out.println("DAS WAR'S");
    }
    System.out.println("UND HIER GEHT ES SCHON WEITER");
  }
}
```

Laufzeit-Fehler (Ausgabe)

```
Fehlerursache
  Leere Datei
DAS WAR'S
UND HIER GEHT ES SCHON WEITER
```

8.4 Eigene Ausnahmeklassen definieren

- Für anwendungsspezifische Ausnahmesituationen sollten eigene Exception-Klassen von der Klasse RuntimeException abgeleitet werden

Beispiel 8.4a

```
public class EmptyFileException extends RuntimeException{
   public EmptyFileException(){
      super();
   }
   public String getMessage(){
      return "Schwerer Fehler: Erzeugung leerer Datei";
   }
}
import java.io.*;
public class Programm {
   public static void ausgabe(String out, String s) throws IOException,
                                                            EmptyFileException {
      if(s=="")   // leerer String
         throw new EmptyFileException();
      FileWriter fw = new FileWriter(out);
      fw.write(s);
      fw.close();
   }
   public static void main(String[] args) {
      try {
         String datei = "c:\\test\\Datei.txt";
         ausgabe(datei, "");
      }
      catch (EmptyFileException e) {
         System.out.println("Fehlerursache");
         System.out.println("  "+e.getMessage());
      }
      catch (IOException e) {
         System.out.println("Fehlerursache");
         System.out.println("  "+e.getMessage());
      }
      finally {
         System.out.println("DAS WAR'S");
      }
      System.out.println("UND HIER GEHT ES SCHON WEITER");
   }
}
```

Laufzeit-Fehler (Ausgabe)

```
Fehlerursache
  Schwerer Fehler: Erzeugung leerer Datei
```

```
DAS WAR'S
UND HIER GEHT ES SCHON WEITER
```

- Hauptprobleme beim Einsatz von Exceptions
  - Herausfinden der Stellen im Programm, an denen Exceptions auftreten können
  - Feststellen des speziellen Exception-Typs

- Exceptions sinnvoll verwenden, da Performanceverlust

9 Streams

- ein **Stream** ist eine geordnete Folge von Bytes (Bytestrom) von einer Datenquelle (Spring) in eine Datensenke (Sink)

- zur Realisierung des Stream-Konzeptes stellt Java im Paket java.io Stream-Klassen bereit, die Methoden zur Datenübertragung bereit, deshalb

  import java.io.*

  am Anfang eines Programms

- Methoden der Stream-Klassen erzeugen bei Fehlern Ausnahmen (IOExceptions). Methodenimplementationen, die peripere Geräte über Streams ansprechen und diese Ausnahmen nicht selbst behandeln, erfordern eine throws-Klausel:

  method (...) **throws IOException** {
  }

9.1 Stream-Klassen: Übersicht

- Ein **Inputstream** ist Bytestrom, der aus einer Datenquelle kommt. Klassen, die einen Stream aus einer Datenquelle einlesen, heißen **Inputstream-Klassen**

- Ein **Outputstream** ist Bytestrom, der in eine Datensenke geht. Klassen, die einen Stream in eine Datensenke schreiben, heißen **Outputstream-Klassen**

Aufteilung von Stream-Klassen nach der Art der übertragenen Daten

1. **Bytestream-Klassen**
   - Objekte der Bytestream-Klassen verarbeiten "rohe" Byte-Ströme, d.h. Werte vom Typ byte
   - alle Byte-Inputstream-Klassen sind von der **abstrakten Basisklasse InputStream** abgeleitet
   - alle Byte-Outputstream-Klassen sind von der **abstrakten Basisklasse OutputStream** abgeleitet

2. **Characterstream-Klassen**
   - Objekte der Characterstream-Klassen verarbeiten Zeichen-Ströme, also Werte vom Typ char
   - alle Character-Inputstream-Klassen sind von der **abstrakten Basisklasse Reader** abgeleitet
   - alle Character-Outputstream-Klassen sind von der **abstrakten Basisklasse Writer** abgeleitet

Aufteilung von Stream-Klassen nach ihrer Funktionalität

1. **Springstream-Klassen**
   - Objekte der Springstream-Klassen können Daten direkt aus einer Datenquelle lesen

2. Sinkstream-Klassen
– Objekte der Sinkstream-Klassen können Daten direkt in eine Datensenke schreiben

3. Processingstream-Klassen
– Übernehmen ein Objekt einer Spring- oder Sinkstream-Klasse und erweitern deren Funktionalität

Stream-Anwendung

StreamClass s_object = **new** *StreamClass* (*parameters*)
s_object.Method()

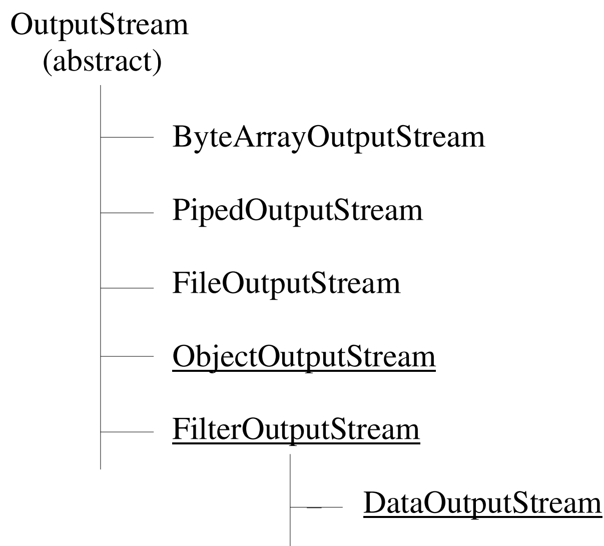
oder

StreamClass s_object = **new** *StreamClass* (*parameters*)
ProcessingStreamClass p_object = **new** *ProcessingStreamClass* (*s_object, parameters*)
p_object.Method()

9.2 Bytestream-Klassen

9.2.1 Outputstream-Klassen

- Realisieren das Schreiben einzelner Bytes oder Bytefolgen in einen Stream



(Processingstream-Klassen unterstrichen)

Abstrakte Klasse OutputStream

Die abtrakte Klasse OutputStream stellt eine Basis für Klassen dar, die Daten ausgeben. Sie ist Oberklasse aller Ausgabe-Streams der Standardbibliothek.

Methoden

```
public abstract void write(int b)
```
Schreibt das niederwertigste Byte von b in den Stream.

```
public void write(byte[] b)
```
Schreibt den gesamten Inhalt von b in den Stream.

```
public void close()
```
Schließt den Stream.

```
public void flush()
```
Bewirkt, dass noch gepufferte Daten in den Stream geschrieben werden.

Klasse ByteArrayOutputStream

Mit der Klasse ByteArrayOutputStream können Schreibzugriffe auf einen internen Byte-Array-Puffer (genauso wie auf einen Ausgabe-Stream) erfolgen.

Klasse PipedOutputStream

Mit der Klasse PipedOutputStream und der zugehörigen Inputstream-Klasse PipedInputStream können zwei Threads Daten austauschen, indem Exemplare dieser Klassen verkettet werden.

Klasse ObjectOutputStream

Bietet somit die Möglichkeit, Objekte zu serialisieren und in den darunterliegenden Stream zu schreiben.

Klasse FileOutputStream

Diese Klasse stellt einen Stream zur Verfügung, mit dem Ausgaben in eine Datei gemacht werden können.

Konstruktoren (Auszug)

```
public FileOutputStream(String name)
```
Erzeugt einen neuen Stream zum Schreiben in die durch name bezeichnete Datei. Falls bereits eine Datei unter diesem Namen existiert, wird sie überschrieben.

```
public FileOutputStream(String name, boolean append)
```
Erzeugt einen neuen Stream zum Schreiben in die durch name bezeichnete Datei. Wenn keine Datei unter diesem Namen existiert, wird sie neu angelegt. Falls bereits eine Datei unter diesem Namen existiert, wird sie in Abhängigkeit vom Flag append entweder überschrieben, oder die neuen Daten werdem am Ende angehängt.

Klasse FilterOutputStream

Bietet eine Basis für Ausgabe-Streams, die die zu schreibenden Daten nach bestimmten Kriterien filtern oder eine zusätzliche Funktionalität bieten

Klasse DataOutputStream

DataOutputStream stellt Methoden zur Verfügung, mit denen Werte der Standarddatentypen in einen Stream geschrieben werden können.

⇒ für die Ausgabe eines jeden Standardtyps wird eine eigene Methode definiert

Methoden

```
public void writeBoolean(boolean v)
public void writeInt(int v)
public void writeFloat(float v)
public void writeDouble(double v)
public void writeChars(String s)
...
```

Schreibt den Wert von v dem Datentyp entsprechend in den Stream

```
public void writeUTF(String str)
```

Schreibt den String str in den Stream, wobei die Zeichen von str mit der Unicode-UTF-8-Codierung verschlüsselt werden.

⇒ Zu Unicode-Format später

Beispiel 9.2a

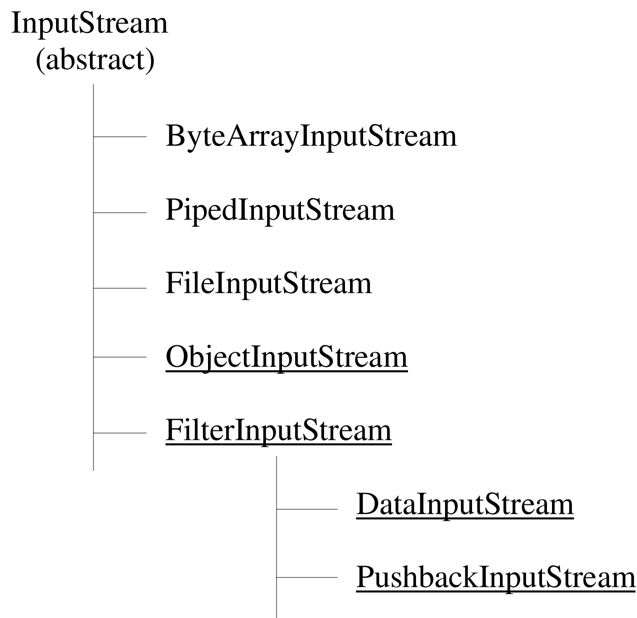
```
import java.io.*;
public class Programm {
  public static void main(String[] args) throws IOException{
    FileOutputStream fos = new FileOutputStream("c:\\test\\Datei.txt");
    DataOutputStream dos = new DataOutputStream(fos);
    dos.writeInt(65);
    dos.writeDouble(Math.PI);
    dos.writeChars("65 € hätte ich ");
    dos.close();
    fos.close();
  }
}
```

Datei.txt

```
 A@   !ûTD- 6 5  ¬  h ä t t e  i c h
```

9.2.2 Inputstream-Klassen

- Realisieren das Lesen einzelner Bytes oder Bytefolgen aus einem Stream



(Processingstream-Klassen unterstrichen)

Abstrakte Klasse InputStream

Die abtrakte Klasse InputStream stellt eine Basis für Klassen dar, die Daten einlesen. Sie ist Oberklasse aller Eingabe-Streams der Standardbibliothek.

Methoden

```
public void close()
```
Schließt den Stream.

```
public void mark(int readlimit)
```
Markiert die momentane Position im Stream. Nachfolgende Aufrufe von reset() springen danach wieder an diese Position.

```
public abstract int read()
```
Liest ein Byte aus dem Stream und liefert es zurück. Der Rückgabewert -1 signalisiert, dass das Ende des Streams erreicht wurde.

```
public int read(byte[] b)
```
Versucht, b.length() Bytes aus dem Stream zu lesen, und speichert sie in b. Liefert die Anzahl der tatsächlich gelesenen Bytes zurück.

```
public void reset()
```
Springt im Stream auf die Position zurück, die mit dem letzten Aufruf von mark() gesetzt wurde.

```
public long skip(long n)
```
Versucht, n Bytes aus dem Stream zu überlesen, und liefert die Anzahl der tatsächlich übersprungenen Bytes.

Klasse ByteArrayInputStream

Mit der Klasse ByteArrayInputStream ist es möglich, auf ein Byte-Array (genauso wie auf einen Eingabe-Stream) lesend zuzugreifen.

Klasse PipedInputStream

Mit der Klasse PipedInputStream und der zugehörigen Outputstream-Klasse PipedOutputStream können zwei Threads Daten austauschen, indem Exemplare dieser Klassen verkettet werden.

Klasse ObjectInputStream

Bietet somit die Möglichkeit, serialisierte Objekte zu lesen und in den darunterliegenden Stream zu schreiben.

Klasse FileInputStream

Diese Klasse stellt einen Stream zur Verfügung, mit dem Dateien ausgelesen werden können.

Konstruktoren (Auszug)

```
public FileInputStream(String name)
```
Erzeugt einen neuen Stream zum Lesen aus einer durch name bezeichnete Datei. Falls die Datei unter diesem Namen nicht existiert, wird die FileNotFoundException geworfen.

Klasse FilterInputStream

Diese Klasse bietet eine Basis für Eingabe-Streams, die die gelesenen Daten nach bestimmten Kriteriten filtern oder eine zusätzliche Funktionalität

Klasse DataInputStream

Stellt Methoden zur Verfügung, mit denen Werte der Standarddatentypen aus einem Stream gelesen werden können.

Methoden

```
public boolean readBoolean()
public int writeInt()
```

```
public float writeFloat()
public double writeDouble()
...
```

Schreibt den Wert von v dem Datentyp entsprechend in den Stream

```
public String readUTF()
```

Liest einen String aus dem Stream, wobei die Zeichen von str mit der Unicode-UTF-8-Codierung verschlüsselt werden.

⇒ Zu Unicode-Format später

Beispiel 9.2b

```
import java.io.*;
public class Programm {
  public static void main(String[] args) throws IOException{
    FileInputStream fis = new FileInputStream("c:\\test\\Datei.txt");
    DataInputStream dis = new DataInputStream(fis);
    System.out.println(dis.readInt());          // 65
    System.out.println(dis.readDouble());       // 3.141592653589793
    for(int i=0; i<14; i++)
      System.out.print(dis.readChar());         // 65 € hätte ich
    dis.close();
    fis.close();
  }
}
```

Textformate

- **ASCII-Zeichensatz**
  - Verschlüsselung der Zeichen durch 7 Bit (0 .. 127)
  - keine Umlaute möglich
  - US-Standard

- **ANSI-Zeichensatz (ISO 8859-1)**
  - Verschlüsselung der Zeichen durch 8 Bit (0 .. 255)
  - Umlaute und andere sprachspezifischer Schriftzeichen möglich
  - Europa-Standard

- **Unicode**
  - Zeichen werden abstrakt definiert und jedem Zeichen eine Nummer zugeordnet
  - derzeit 96.382 unterschiedliche Zeichen
  - abstrakte Zeichen müssen in Bitmuster transformiert werden: Unicode Transformation Format (UTF)
  - verschiedene Transformationen: UTF-7, UTF-8, UTF-16, UTF-32

- **UTF-8**
  - Standard in Java
  - Unicode-Zeichen 0..127 entsprechen dem ASCII-Zeichensatzes und werden in 1 Byte verschlüsselt
  - Unicode-Zeichen 128..2047 werden in 2 Byte verschlüsselt
  - Unicode-Zeichen darüber in 3 Byte

Beispiel 9.2c

```
import java.io.*;
public class Programm {
  public static void main(String[] args) throws IOException{
    FileOutputStream fos = new FileOutputStream("c:\\test\\Datei.txt");
    DataOutputStream dos = new DataOutputStream(fos);
    dos.writeInt(65);
    dos.writeDouble(Math.PI);
    dos.writeUTF("65 € hätte ich");
    dos.writeUTF("gezahlt");
    dos.close();
    fos.close();

    FileInputStream fis = new FileInputStream("c:\\test\\Datei.txt");
    byte[] buf = new byte[100];
    fis.read(buf);
    for(int i=0; i<40; i++)
      System.out.printf("%X ",buf[i]);    // Bildschirmausgabe
    fis.close();
  }
}
```

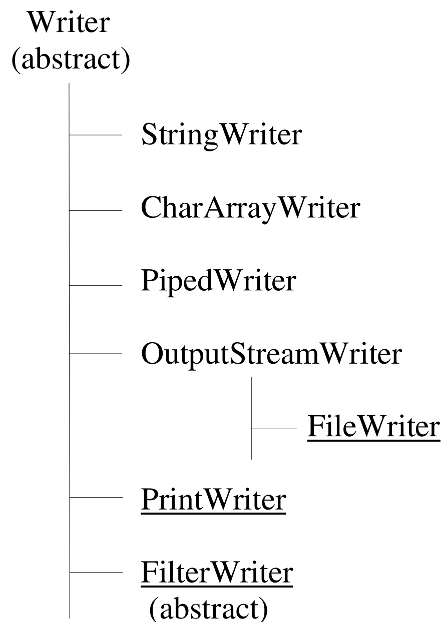
Ausgabe der Datei.txt byteweise hexadezimal auf den Bildschirm

```
0 0 0 41 40 9 21 FB 54 44 2D 18 0 11 36 35 20 E2 82 AC 20 68 C3 A4 74 74 65 20 69 63 68
   65            PI               17  6  5     €        h    ä    t  t  e     i  c  h

0 7 67 65 7A 61 68 6C 74
 7 g  e  z  a  h  l  t
```

9.3 Characterstream-Klassen

9.3.1 Writer-Klassen

- Realisieren das Schreiben in einen Stream



(Processingstream-Klassen unterstrichen)

Abstrakte Klasse Writer

Konstruktoren (Auszug)

```
protected Writer()
```
Erzeugt einen neuen Writer.

Methoden

```
public abstract void close()
```
Schließt den Stream.

```
public abstract void flush()
```
Bewirkt, dass noch gepufferte Daten in den Stream geschrieben werden.

```
public void write(String str)
```
Schreibt die in str enthaltenen Zeichen in den Stream.

```
public void write(int c)
```
Schreibt die niederwertigen 2 Bytes von c als `char`-Wert in dem Stream.

Klasse StringWriter

Bietet die Möglichkeit, in einen internen String-Puffer (genauso wie in einen Stream) zu schreiben. Hierbei kann wahlweise eine Anfangslänge für den String-Puffer gesetzt werden. Falls die Länge nicht mehr ausreicht, um die Daten eines Schreibzugriffs vollständig aufzunehmen, wird sie automatisch angepasst.

Klasse CharArrayWriter

Bietet die Möglichkeit, in einen internen Char-Array-Puffer (genauso wie in einen Stream) zu schreiben. Hierbei kann wahlweise eine Anfangsgröße für den Array-Puffer gesetzt werden. Wenn die Größe des Array-Puffers nicht mehr ausreicht, um die Daten eines Schreibzugriffs vollständig aufzunehmen, wird seine Größe automatisch angepasst.

Klasse PipedWriter

Writer zur Ausgabe in einen PipedReader zum Datenaustausch zwischen Threads.

Klasse OutputStreamWriter

Stellt eine Basis für Klassen dar, die Daten ausgeben. Sie ist Oberklasse aller Ausgabe-Streams der Standardbibliothek.

Klasse FileWriter

Stellt einen Stream zur Verfügung, mit dem Unicode-Zeichen in Dateien geschrieben werden können, wobei die Unicode-Zeichen in den Zeichensatz der Plattform gewandelt. Wenn eine andere Codierung oder eine bestimmte Puffergröße benutzt werden soll, muß ein Exemplar von OutputStreamWriter benutzt werden, das auf einem FileOutputStream arbeitet.

Konstruktoren (Auszug)

`public FileWriter(String fileName)`

Erzeugt einen neuen Stream zum Schreiben in die durch fileName bezeichnete Datei. Falls bereits eine Datei unter diesem Namen existiert, wird sie überschrieben.

`public FileWriter(String fileName, boolean append)`

Erzeugt einen neuen Stream zum Schreiben in die durch fileName bezeichnete Datei. Falls bereits eine Datei unter diesem Namen existiert und append true ist, werden die neuen Daten angehängt, andernfalls wird sie überschrieben.

Methoden

FileWriter besitzt keine eigenen Methoden, sondern nur die von Writer und OutputStreamWriter geerbten

Beispiel 9.3a

```
import java.io.*;
public class Programm {
  public static void main(String[] args) throws IOException{
    FileWriter fw = new FileWriter("c:\\test\\Datei.txt");
    fw.write("65 € hätte ich");
    fw.close();
  }
}
```

-->Datei (mit Editor)
65 € hätte ich

Klasse PrintWriter

Mit PrintWriter können Werte der einfachen Datentypen im Klartext in einen Stream ausgegeben werden. Hierzu werden die Methoden print() und println() zur zeilenorientierten Ausgabe der Standardtypen zur Verfügung gestellt. Die Unicode-Zeichen werden in den Zeichensatz der Plattform umcodiert. Seit Version 5 des SDK existiert die Methode printf() zur formatierten Ausgabe in Anlehnung an C/C++

Abstrakte Klasse FilterWriter

Diese Klasse bietet eine Basis für Ausgabe-Streams, die Unicode-Zeichen schreiben und dabei nach bestimmten Kriteriten filtern oder eine zusätzliche Funktionalität bieten. Ein FilterWriter arbeitet grundsätzlich auf einem anderen Writer. Zur Implementierung der gewünschten Funktionalität müssen die 3 write()-Methoden überschrieben werden, die standardmäßig die Aufrufe direkt an den angeschlossenen Writer weiterleiten.

Konstruktoren

```
protected FilterWriter(Writer out)
```

Erzeugt einen neuen FilterWriter für den Stream out.

Methoden

Überschreibt die Methoden close(), flusch() und write() aus der Klasse Writer.

Vorgehen bei der Konstruktion eigener Filter-Klassen

1. Ableitung der eigenen Filter-Klasse von FilterWriter
2. Aufruf des FilterWriter-Konstruktors im Konstruktor der der eigenen Filter-Klasse um out auszuführen
3. Überschreiben der 3 write()-Methoden von FilterWriter. Ausführung des eigenen Filteralgorithmus vor Übergabe des auszugebenden Zeichens an den FilterWriter

Beispiel 9.3b

```
import java.io.*;
public class ChangeSZ extends FilterWriter{
  public ChangeSZ(Writer out)
  {
```

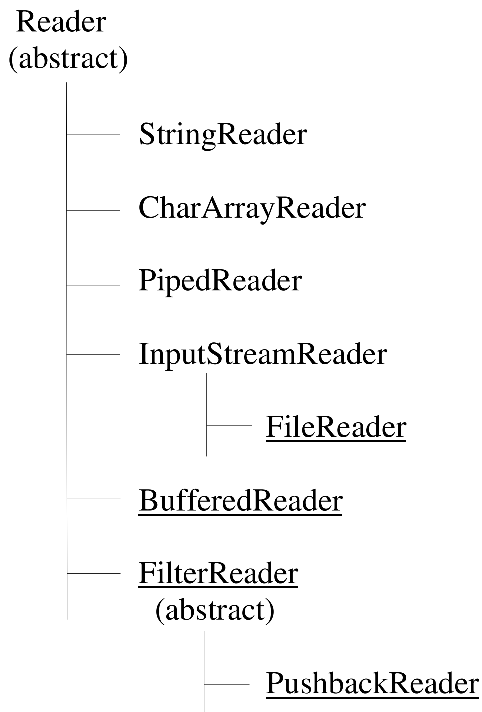
```
      super(out);                         // Konstruktor aus FilterWriter
   }
   public void write(int c) throws IOException {
      if(c=='ß')
         super.write("ss");               // write(String) aus Writer
      else
         super.write(c);                  // write(int) aus FilterWriter
   }
   public void write(String s) throws IOException{
      for(char c: s.toCharArray())
         write(c);                        // write(int) aus ChangeSZ
   }
}
import java.io.*;
public class Programm {
   public static void main(String[] args) throws IOException{
      String s = "Laßt den Maßkrug bloß stehen!";
      FileWriter fw = new FileWriter("c:\\test\\Datei.txt");
      ChangeSZ out = new ChangeSZ(fw);
      out.write(s);                       // write(String) aus ChangeSZ
      out.close();
      fw.close();
   }
}
```

-->Datei.txt

Lasst den Masskrug bloss stehen!

9.3.2 Reader-Klassen

- Realisieren das Lesen aus einem Stream



(Processingstream-Klassen unterstrichen)

Abstrakte Klasse Reader

Konstruktoren (Auszug)

`protected Reader()`
Erzeugt einen neuen Reader.

Methoden

`public abstract void close()`
Schließt den Stream.

`public void mark(int readAheadLimit)`
Markiert die momentane Position im Stream. Nachfolgende Aufrufe von reset() springen danach wieder an diese Position.

`public int read()`
Liest ein Zeichen und liefert es in der Unicode-Codierung zurück. Der Rückgabewert ist `-1`, falls das Ende des Streams erreicht ist.

```
public int read(char[] b)
```

Versucht, b.length Zeichen aus dem Stream zu lesen und speichert sie in b. Wenn beim Versuch, das erste Zeichen zu lesen, das Dateiende bereits erreicht ist, ist der Rückgabewert -1. Ansonsten wird die Anzahl der tatsächlich gelesenen Bytes zurückgeliefert.

```
public void reset()
```

Springt im Stream auf die Position zurück, die mit dem letzten Aufruf von mark() markiert wurde.

Klasse StringReader

Bietet die Möglichkeit, auf einen String (genauso wie auf einen Eingabe-Stream) lesend zuzugreifen. Es werden beim Lesen alle 16 Bit der Unicode-Zeichen des Strings berücksichtigt.

Klasse CharArrayReader

Möglichkeit, auf ein char-Array (genauso wie auf einen Eingabe-Stream) lesend zuzugreifen.

Klasse PipedReader

Reader zur Eingabe aus einem PipedWriter zum Datenaustausch zwischen Threads.

Klasse InputStreamReader

Basis für Reader-Klassen, die die Daten aus InputStreams lesen.

Klasse FileReader

Stellt einen Stream zur Verfügung, der eine Datei liest. Hierbei werden die gelesenen Bytes aus der Zeichencodierung der Plattform in Unicode-Zeichen umgesetzt. Zur Umcodierung wird die voreingestellte Zeichencodierung der Plattform verwendet. Wenn eine andere Codierung benutzt werden soll, muß ein Exemplar von InputStreamReader eingesetzt werden, das auf einem FileInputStream arbeitet.

Konstruktoren (Auszug)

```
public FileReader(String fileName)
```

Erzeugt einen neuen FileReader zum Lesen aus der durch fileName bezeichneten Datei.

Methoden

FileReader besitzt keine eigenen Methoden, sondern nur die von Reader und InputStreamReader geerbten

Beispiel 9.3c

-->Datei
65 € hätte ich

```
public class Programm {
  public static void main(String[] args) throws IOException{
    FileReader fr = new FileReader("c:\\test\\Datei.txt");
    int c;
    while(((c=fr.read()) != -1))
      System.out.print((char)c);  // 65 € hätte ich
    fr.close();
  }
}
```

Klasse BufferedReader

Gepufferter Eingabe-Stream für Unicode-Zeichen. Er arbeitet stets auf einem anderen Reader-Objekt, aus dem er die Daten liest.

Abstrakte Klasse FilterReader

Diese Klasse bietet eine Basis für Eingabe-Streams, die Unicode-Zeichen lesen und nach bestimmten Kriterien filtern oder eine zusätzliche Funktionalität bieten. Ein FilterReader arbeitet grundsätzlich auf einem anderen Reader. Zur Implementierung der gewünschten Funktionalität müssen die read()-Methoden überschrieben werden, die standardmäßig die Aufrufe direkt an den angeschlossenen Reader weiterleiten.

Klasse PushbackReader

Bietet die Möglichkeit, die zuletzt gelesenen Zeichen wieder in den Puffer zurückzulegen, so daß sie erneut gelesen werden können.

Konstruktoren

`public PushbackInputStream(InputStream in)`
Erzeugt einen neuen PushbackInputStream, der Daten aus dem Stream in liest. Er erlaubt das Zurücklegen von einem Zeichen.

Methoden

enthält unter anderem neben den Methoden der Basisklassen die Methoden

`public void unread(int b)`
Speichert das niederwertigste Byte von b in den Stream zurück.

`public void unread(byte[] b)`
Speichert den gesamten Inhalt von b in den Stream zurück.

Beispiel 9.3d

-->Datei.txt

DIE HAELFTE DER AENDERUNGEN

```
import java.io.*;
public class ChangeAE extends PushbackReader {
  public ChangeAE(Reader in)
  {
    super(in);
  }
  public int read() throws IOException {
    int erstesZeichen = super.read();     // read() aus PushbackReader
    int zweitesZeichen = super.read();
    if((erstesZeichen == 'A') && (zweitesZeichen == 'E')){
      return 'Ä';
    }
    else {
      super.unread(zweitesZeichen);
      return erstesZeichen;
    }
  }
}
import java.io.*;
public class Programm {
  public static void main(String[] args) throws IOException{
    FileReader fr = new FileReader("c:\\test\\Datei.txt");
    ChangeAE in = new ChangeAE(fr);
    int c;
    while((c = in.read()) != -1)
      System.out.print((char)c); // DIE HÄLFTE DER ÄNDERUNGEN
    in.close();
    fr.close();
    System.out.println("Ende");
  }
}
```

9.4 Objektserialisierung

- **Serialisierung/Deserialisierung**
  Fähigkeit, ein Objekt aus dem Hauptspeicher einer Anwendung in ein Format zu konvertieren, das eine Übertragung des Objektes in eine Datei oder Netzwerkverbindung erlaubt bzw. ein konvertiertes Objekt wieder zu rekonstruieren.

- Klassen, deren Objekte serialisiert werden sollen, müssen die Schnittstelle **Serializable** implementieren

Klasse ObjectOutputStream

Bietet somit die Möglichkeit, Objekte zu serialisieren und in den darunterliegenden Stream zu schreiben.

Konstruktoren (Auszug)

```
public ObjectOutputStream(OutputStream out)
```
Erzeugt einen neuen ObjectOutputStream auf dem Stream out.

Methoden (Auszug)

Neben Methoden zum Serialisieren primitiver Datentypen und weiteren Methoden zur Serialisierung bietet die Klasse eine Methode zur Serialisierung kompletter Objekte:

```
public void writeObject(Object obj)
```
Schreibt das Objekt obj in den Stream.

⇒ wir wollen uns im Rahmen dieser Vorlesung auf diese eine Methode beschränken

Beispiel 9.4a

```
import java.io.*;
public class Person implements Serializable{
   private String name;
   private String vorname;
   public Person(String name, String vorname) {
      this.name = name;
      this.vorname = vorname;
   }
   public String toString(){
      return vorname+" "+name;
   }
}
import java.io.*;
import java.io.*;
public class Programm {
   public static void main(String[] args) throws IOException{
      Person p1 = new Person("Müller","Max");
      Person p2 = new Person("Meier","Anne");
      System.out.println(p1.toString());    // Max Müller
```

```
        System.out.println(p2.toString());    // Anne Meier

        FileOutputStream fos = new FileOutputStream("c:\\test\\Person.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        oos.writeObject(p1);
        oos.writeObject(p2);
        oos.close();
        fos.close();
    }
}
```

- statische Felder und mit dem Schlüsselwort transient gekennzeichneten Felder werden nicht serialisiert

- Bei der Serialisierung werden folgende Daten in den Output-Stream geschrieben

    1. die Klasse des als Argument übergebenen Objektes
    2. die Signatur der Klasse
    3. alle nichtstatischen und nichttransienten Felder einschließlich der aus den Basisklassen geerbten Felder

Klasse ObjectInputStream

Bietet somit die Möglichkeit, Objekte aus dem darunterliegenden Stream zu lesen und zu deserialisieren.

Konstruktoren (Auszug)

`public ObjectInputStream(IntputStream in)`
Erzeugt einen neuen ObjectIntputStream auf dem Stream in.

Methoden (Auszug)

Neben Methoden zum Deserialisieren primitiver Datentypen und weiteren Methoden zur Deserialisierung bietet die Klasse eine Methode zur Deserialisierung kompletter Objekte:

`public Object readObject()`
Liest ein serialisiertes Objekt aus dem Stream und liefert es zurück.

Beispiel 9.4b

```
import java.io.*;
public class Person implements Serializable{..}  // wie Beispiel 9.4a
import java.io.*;
public class Programm {
    public static void main(String[] args) throws IOException,
                                                  ClassNotFoundException {
        FileInputStream fis = new FileInputStream("c:\\test\\Person.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
```

```
      Person p1 = (Person) ois.readObject();
      Person p2 = (Person) ois.readObject();
      System.out.println(p1.toString());    // Max Müller
      System.out.println(p2.toString());    // Anne Meier

      ois.close();
      fis.close();
   }
}
```

Beispiel 9.4c

```
import java.io.*;
public class Person implements Serializable{
   protected String name;
   protected String vorname;
   public Person(String name, String vorname) {
      this.name = name;
      this.vorname = vorname;
   }
   public String toString(){
      return vorname+" "+name;
   }
}
import java.io.*;
public class Student extends Person implements Serializable {
   private float[] noten = new float[3];
   public Student(String name, String vorname){
      super(name, vorname);
   }
   public void notenEingabe(float... values){
      for(int i=0; i<values.length; i++)
         noten[i] = values[i];
   }
   public String toString(){
      String out = super.toString()+":  ";
      for(int i=0; i<3; i++)
         out = out + noten[i] + "   ";
      return out;
   }
}
import java.io.*;
public class Programm {
   public static void main(String[] args) throws IOException {
      Student s1 = new Student("Müller","Max");
      s1.notenEingabe(2.3F,1.7F,3.0F);
      System.out.println(s1.toString()); // Max Müller:  2.3  1.7  3.0

      FileOutputStream fos = new FileOutputStream("c:\\test\\Person.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fos);

      oos.writeObject(s1);
      oos.close();
```

```
      fos.close();
   }
}
```

Beispiel 9.4d

```
import java.io.*;
public class Person implements Serializable {..}
import java.io.*;
public class Student extends Person implements Serializable {..}
import java.io.*;
public class Programm {
   public static void main(String[] args) throws IOException,
ClassNotFoundException {
      FileInputStream fis = new FileInputStream("c:\\test\\Person.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);

      Student s1 = (Student) ois.readObject();
      System.out.println(s1.toString());    // Max Müller:  2.3  1.7  3.0

      ois.close();
      fis.close();
   }
}
```

10 Collections*

- **Collections**
  Klassen, die Datenstrukturen zur Speicherung und Verwaltung von Mengen von Objekten bereitstellen.

- "traditionelle" Collections seit JDK 1.0: Vector, Stack, Dictionary, Hashtable und BitSet.

10.1 Überblick

- Collections können beliebige Objekte verwalten

- alle Collections sind durch Hinzufügen von Elementen in ihrer Größe beliebig erweiterbar

- Elemente von Collections sind immer Referenzen

- ab JDK 5.0 sind alle Collections generisch

- 4 Collection-Familien (Typen)
  - List (Listen)
  - Queue (Warteschlangen)
  - Set (Mengen)
  - Map (Vezeichnisse)

- **Listen**
  Collections geordneter Elemente, auf die über Indizes zugegriffen werden kann. Elemente können an beliebigen Stellen innerhalb der Collection eingefügt werden

- **Warteschlangen**
  Collections geordneter Elemente, auf die nach dem FIFO-Prinzip (First In First Out) zugegriffen werden kann.

- **Mengen**
  Collections ungeordneter Elemente, wobei jedes nur einmal vorkommen darf

- **Verzeichnisse**
  Collections ungeordneter Elemente, die aus Schlüssel-Wert-Paaren bestehen. Der Zugriff auf die Objekt-Werte erfolgt über den Schlüssel

---

* Grafiken nach Heinisch/Hoffmann/Goll "Java als erste Programmiersprache"

Design-Prinzipien der Collection-Klassen

- **Schnittstellen** legen Gruppen von Operationen für die verschiedenen Collectiontypen fest.

- **Abstrakte Basisklassen** führen die Operationen der Schnittstellen auf eine minimale Zahl von als abstrakt deklarierten Grundoperationen zurück

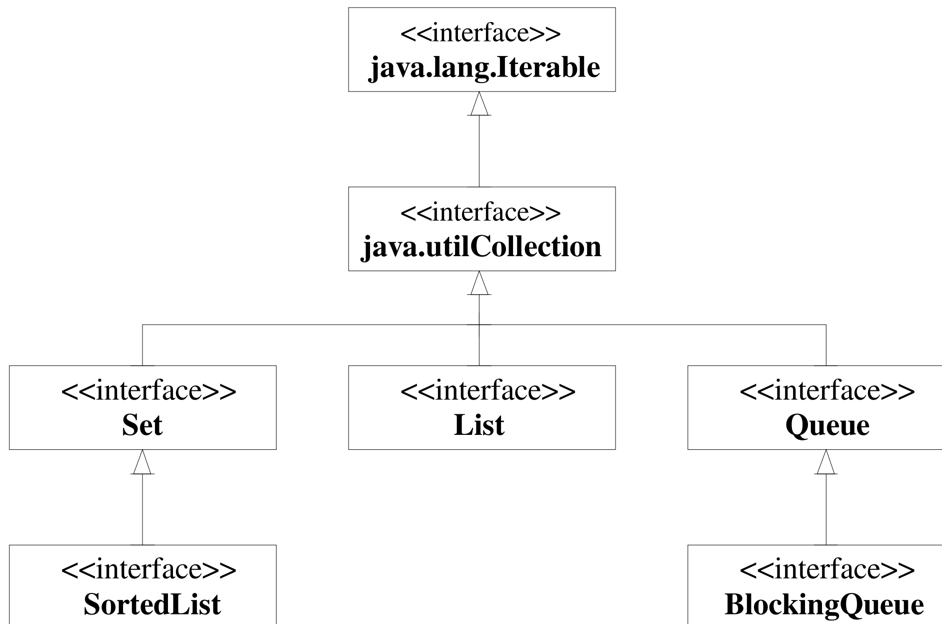
- **Konkrete Klassen** für einen bestimmten Collectiontyp beerben die entsprechende abstrakte Basisklasse und ergänzen die unbedingt erforderlichen Grundoperationen

- Methoden (Algorithmen), wie z.B. die Suche nach einem Element, gehören zum Teil zur Schnittstelle der einzelnen Collection-Typen. Zusätzlich gibt es mit der Klasse **Collections** eine Utility-Klasse mit weiteren Methoden (Algorithmen).

Übersicht über Collection-Typen und einige der zugehörigen Klassen

	Klasse	**Organisation**	**Freiheit beim Zugriff**	**Mechanismus beim Zugriff**	**Duplikate zugelassen**	**Besondere Eigenschaften**
List	**ArrayList**	geordnet	wahlfrei	über Index	ja	schneller Zugriff
	Stack	geordnet	sequenziell	letztes Element	ja	LIFO
Queue	**LinkedList**	geordnet	sequenziell	nächstes Element	ja	FIFO
Set	**HashSet**	ungeordnet	wahlfrei	einfacher Test	nein	schneller Zugriff
	TreeSet	sortiert	wahlfrei	einfacher Test	nein	-

Hierarchie der Collection-Schnittstellen



Schnittstelle Collection<E>

Methoden (Auszug)

`public boolean add(E element)`

Fügt das Element vom Typ E zur Collection hinzu. Der Rückgabewert ist true, falls sich die Collection durch den Aufruf verändert hat, sonst false.

`public boolean remove(E element)`

Entfernt das Element vom Typ E aus der Collection. Falls das Element mehrmals in der Collection enthalten ist, wird nur das erste Vorkommen entfernt. Der Rückgabewert ist true, falls das Element gefunden und entfernt wurde, sonst false.

`public int size()`

Liefert die Anzahl der enthaltenen Elemente zurück.

`public boolean contains(E element)`

Liefert true, falls das Element in der Collection enthalten ist, sonst false.

`public boolean isEmpty()`

Liefert true, wenn die Collection leer ist, sonst false.

`public Iterator iterator()`

Liefert einen Iterator, mit dem die in der Collection enthaltenen Elemente nacheinander abgerufen werden können.

Iterator<E>

- Iteratoren erlauben es, über alle Elemente einer Collection zu laufen, unabhängig vom speziellen Typ der Collection

Beispiel 10.1a

```
import java.util.*;
public class Programm {
  public static void main(String[] args) {
    Collection<String> coll = new ArrayList<String>();
    coll.add("Anna");
    coll.add("Hannes");
    coll.add("Willi");
    Iterator<String> iter = coll.iterator();
    while(iter.hasNext())
      System.out.print(iter.next()+" "); // Anna Hannes Willi
  }
}
```

- Ab JDK 5.0 können auch Collections mit der mit der foreach-Schleife iteriert werden

Beispiel 10.1b

```
import java.util.*;
public class Programm {
  public static void main(String[] args) {
    Collection<String> coll = new ArrayList<String>();
    coll.add("Anna");
    coll.add("Hannes");
    coll.add("Willi");
    for(String s : coll)
      System.out.print(s+" ");  // Anna Hannes Willi
  }
}
```

10.2 Listen

Klassen ArrayList<E> und LinkedList<E>

- **ArrayList**
  Container, der dem List-Interface entspricht und seine Elemente in einem Array hält, wodurch auch der direkte Zugriff über einen Index möglich ist

Methoden (Auszug)

```
public E get(int index)
```
Liefert das Listenelement an der Position index zurück.

```
public E set(int index, E element)
```
Ersetzt das Listenelement an der Position index durch element.

```
public void add(int index, E element)
```
Fügt das Element element an der Position index in die Liste ein.

```
public E remove(int index)
```
Entfernt das Listenelement an der Position index.

Beispiel 10.2a

```
import java.util.*;
public class Programm {
   public static void main(String[] args) {
      List<String> liste = new ArrayList<String>();
      liste.add("Anna");
      liste.add("Hannes");
      liste.add("Willi");
      System.out.println(liste);  // [Anna, Hannes, Willi]

      liste.add(2,"Paul");
      System.out.println(liste);  // [Anna, Hannes, Paul, Willi]

      liste.remove(1);
      System.out.println(liste);  // [Anna, Paul, Willi]

      liste.set(0, "Anna-Maria");
      System.out.println(liste);  // [Anna-Maria, Paul, Willi]
   }
}
```

Klasse Stack<E>

- **Stack**
  Container stellt einen Stack nach dem LIFO-Prinzip dar, in dem Objekte abgelegt werden können.

Methoden (Auszug)

```
public E peek()
```
Liefert das Element, das auf der Spitze des Stacks liegt. Das Element bleibt hierbei auf dem Stack liegen.

```
public E pop()
```
Entfernt das Element, das auf der Spitze des Stacks liegt, und liefert es zurück.

```
public E push(E element)
```
Legt das Element auf der Spitze des Stacks ab.

Beispiel 10.2b

```
import java.util.*;
public class Programm {
  public static void main(String[] args) {
    Stack<Integer> stapel = new Stack<Integer>();

    stapel.push(1);
    stapel.push(2);
    stapel.push(3);
    stapel.push(4);
    System.out.println(stapel);                      // [1, 2, 3, 4]

    System.out.println(stapel.pop()+" entnommen"); // 4 entnommen
    System.out.println(stapel);                      // [1, 2, 3]

    stapel.push(5);
    stapel.push(6);
    System.out.println(stapel);                      // [1, 2, 3, 5, 6]

    System.out.println(stapel.peek()+" belassen"); // 6 belassen
    System.out.println(stapel);                      // [1, 2, 3, 5, 6]
  }
}
```

10.3 Queues

Klasse LinkedList<E>

- **LinkedList**
  Diese Klasse realisiert eine doppelt verkettete Liste. Sie implementiert hier aber die Schnittstelle Queue

Methoden (Auszug)

```
public E peek()
```
Liefert das nächste Element in der Warteschlange. Das Element bleibt hierbei in der Warteschlange liegen.

```
public E poll()
```
Entfernt das nächste Element aus der Warteschlange und liefert es zurück.

```
public boolean offer(E element)
```
Fügt das Element in die Warteschlange ein. Ist sie voll, wird false zurück geleifert.

Beispiel 10.3a

```
import java.util.*;
public class Programm {
  public static void main(String[] args) {
    Queue<Integer> schlange = new LinkedList<Integer>();
```

```
        schlange.offer(1);
        schlange.offer(2);
        schlange.offer(3);
        schlange.offer(4);
        System.out.println(schlange);                         // [1, 2, 3, 4]

        System.out.println(schlange.poll()+" entnommen");// 1 entnommen
        System.out.println(schlange);                         // [2, 3, 4]

        schlange.offer(5);
        schlange.offer(6);
        System.out.println(schlange);                         // [2, 3, 4, 5, 6]

        System.out.println(schlange.peek()+" belassen"); // 2 belassen
        System.out.println(schlange);                         // [2, 3, 4, 5, 6]
    }
}
```

10.4 Sets

- Sets sind Collections, deren Elemente Unikate sind

Klassen HashSet<E> und TreeSet<E>

- **HashSet**
  Menge, in der die Elemente intern mittels Streuspeicherverfahren (Hashing) gespeichert werden. Die Elemente des HashSet sind unsortiert

- **TreeSet**
  Menge, in der die Elemente intern in einem Baum organisiert sind. Die Elemente des TreeSet sind damit sortiert

Methoden (Auszug)

`public boolean add(E element)`
Fügt das Element zur Menge hinzu und liefert true, falls es dort noch nicht enthalten war.

.

`public boolean contains(E element)`
Liefert true, wenn Element in der Menge enthalten ist.

`public boolean remove(E element)`
Entfernt das Element aus der Menge und liefert true, falls es in der Menge enthalten war.

`public int size()`
Liefert die Anzahl der Elemente in der Menge zurück.

Beispiel 10.4a

```
import java.util.*;
public class Programm {
   public static void main(String[] args) {
      Collection<Integer> c1 = new ArrayList<Integer>();
      Collection<Integer> c2 = new HashSet<Integer>();
      Collection<Integer> c3 = new TreeSet<Integer>();

      while(c1.size()<5)
         c1.add((int)(Math.random()*50));
      System.out.println(c1);              // [43, 38, 43, 22, 5]

      while(c2.size()<5)
         c2.add((int)(Math.random()*50));
      System.out.println(c2);              // [16, 19, 4, 9, 13]

      while(c3.size()<5)
         c3.add((int)(Math.random()*50));
      System.out.println(c3);              // [5, 28, 32, 35, 48]
   }
}
```

10.5 Klasse Collections

- Die Utility-Klasse **Collections** stellt eine Reihe von statischen Hilfsmethode für Collections zur Verfügung, insbesondere zum Suchen und Sortieren

Beispiel 10.5a

```
import java.util.*;
public class Programm {
   public static void main(String[] args) {
      int[] array = {3,1,5,4,2};
      for(int i : array)
         System.out.print(i+" ");  // 3 1 5 4 2
      System.out.println();

      ArrayList<Integer> list = new ArrayList<Integer>();
      for(int i=0; i<array.length; i++)
         list.add(array[i]);       // Boxing int --> Integer

      Collections.sort(list);

      for(int i=0; i<list.size(); i++)
         array[i] = list.get(i);   // Unboxing Integer --> int

      for(int i : array)
         System.out.print(i+" ");  // 1 2 3 4 5
   }
}
```

- Klassen, deren Objekte

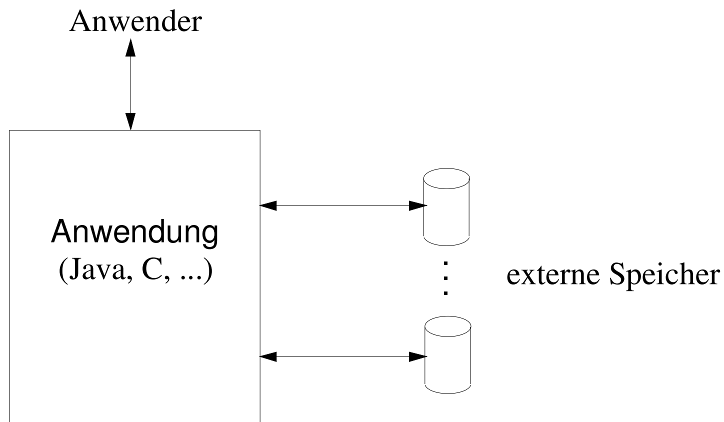
    a) in Collections sortiert werden sollen oder
    b) deren Objekte in sortierten Collections (TreeSet, TreeMap, ...) gespeichert werden sollen

  müssen die Schnittstelle Comparable oder Comparator implementieren.

11 Datenbankzugriff mit JDBC

11.1 Dateisysteme

- **Persistierung der Daten einer Anwendung in Dateien**



Beispiel

Objekte des Typs **Person**

PNR	Name	Vorname	Ort	Vater	Mutter
21	Meier	Udo	HRO	-	-
45	Weber	Klaus	HGW	21	62
62	Müller	Uta	HRO	-	-
37	Meier	Nora	HST	21	55
55	Meier	Elke	HRO	-	-
42	Schulz	Björn	HST	-	62

Beispiel 11.1a

```
import java.io.*;
public class Person implements Serializable{
   private int pnr;
   private String name;
   private String vorname;
   private String ort;
   private int vater;
```

```
  private int mutter;
  public int getPNR(){
    return this.pnr;
  }
  public String getName(){
    return this.name;
  }
  public String getVorname(){
    return this.vorname;
  }
  public String getOrt(){
    return this.ort;
  }
  public int getVater(){
    return this.vater;
  }
  public int getMutter(){
    return this.mutter;
  }
  public Person(int pnr, String name, String vorname,
                String ort, int vater, int mutter) {
    this.pnr = pnr;
    this.name = name;
    this.vorname = vorname;
    this.ort = ort;
    this.vater = vater;
    this.mutter = mutter;
  }
}
import java.io.*;
public class Programm {
  public static void main(String[] args) throws IOException{
    FileOutputStream fos = new FileOutputStream("c:\\test\\Person.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    Person p;

    p = new Person(21,"Meier","Udo","HRO",0,0);
    oos.writeObject(p);
    p = new Person(45,"Weber","Klaus","HGW",21,62);
    oos.writeObject(p);
    p = new Person(62,"Müller","Uta","HRO",0,0);
    oos.writeObject(p);
    p = new Person(37,"Meier","Nora","HST",21,55);
    oos.writeObject(p);
    p = new Person(55,"Meier","Elke","HRO",0,0);
    oos.writeObject(p);
    p = new Person(42,"Schulz","Björn","HST",0,62);
    oos.writeObject(p);

    oos.close();
    fos.close();
  }
}
```

Beispiel 11.1b

```
// Gesucht sind PNR, Vornamen und Name aller Rostocker Personen
import java.io.*;
public class Programm {
  public static void main(String[] args) throws IOException,
                                                ClassNotFoundException {
    FileInputStream fis = new FileInputStream("c:\\test\\Person.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    Person p;

    for(int i=0;i<6;i++){                   //6 Objekte wurden serialisiert
      p = (Person) ois.readObject();
      if(p.getOrt().equals("HRO"))
        System.out.println(p.getPNR()+" "+p.getVorname()+" "+p.getName());
    }

    ois.close();
    fis.close();
  }
}
```

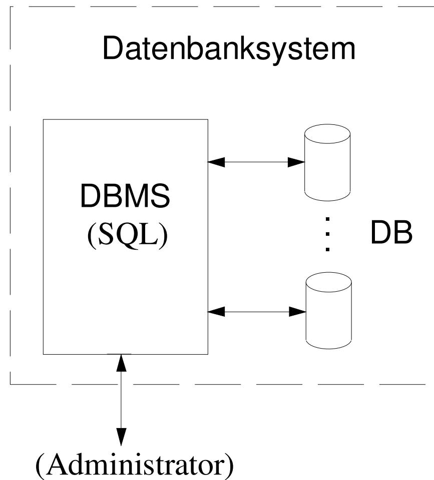
Ausgabe

```
21  Udo Meier
62  Uta Müller
55  Elke Meier
```

- Probleme bei der Persistierung in Dateien
  - Anzahl, Größe, und Komplexität führen zu inakzeptablen Zugriffszeiten auf Informationen
  - Komplexität der Gesamtdatenstruktur erfordert erheblichen Programmieraufwand bei der Informationsrecherche

11.2 Datenbanksysteme

Komponenten eines relationalen Datenbanksystems



- **Datenbank (DB)**
  - Speichermedium mit Daten
  - physische Speicherung der Daten erfolgt in spezielle Strukturen (z.B. B-Bäume, indexsequentiell, Hash-Verfahren ...)
  - diese Strukturen können anwendungsbezogen ausgewählt werden (z.B. für schnelles Einfügen von Daten, schnelle Recherche nach bestimmten Daten ...)
  - Verwaltung der Daten durch Datenbankadministrator
  - für den Anwender werden diese Daten (logisch) durch Tabellen (Relationen) repräsentiert

- **Datenenbankbetriebssystem (DBMS)**
  - Softwaresystem zur Strukturierung, Manipulation und Recherche von Daten
  - eigene Datenbanksprache: SQL (Structured Query Language)

11.3 SQL

- 3 Bestandteile von SQL
  - DDL (Data Description Language): Erzeugung von Tabellenstrukturen
  - DML (Data Manipulation Language): Hinzufügen/Ändern/Löschen von Daten in Tabelen
  - QL (Query Language): Recherche nach Daten in Tabellen

Erzeugung von Tabellenstrukturen

Syntax

CreateTableStatement ::=

CREATE TABLE *TableName* **(** *ColumnDefinitionList* **)**

ColumnDefinition ::=

ColumnName SQLDataType

- *ColumnDefinitionList*: Kommagetrennte Liste von Spaltennamen und den zugehörigen Datentypen

- **SQL Datentypen** (Auswahl)

Bezeichnung	**Bedeutung**
VARCHAR(n)	Zeichenkette variabler Länge mit max. n Zeichen
SMALLINT	16-Bit Integer
INTEGER	32-Bit Integer
FLOAT	Gleitkommazahl mit 15 Stellen
DATE	Datum

Beispiel

```
CREATE TABLE Person
    (PNR        SMALLINT,
     Name       VARCHAR(10),
     Ort        VARCHAR(3))
```

erzeugt eine leere Tabelle 'Person' mit den Spalten PNR, Name und Ort

Hinzufügen von Datensätzen in eine Tabelle

Syntax

InsertStatement ::=

INSERT INTO *TableName* **VALUES (** *ValueList* **)**

- *ValueList*: kommagetrennte Liste von Werten
- es wird eine neue Zeile mit den angegebenen Werten in die Tabelle eingefügt
- SQL besitzt im Gegensatz zu Programmiersprachen eine dreiwertige Logik: neben "normalen" Datenwerten existiert der Wert NULL mit der Bedeutung "unbekannt"

Beispiel

INSERT INTO Person VALUES (21, 'Meier', 'HRO')
INSERT INTO Person VALUES (45, 'Weber', NULL)
INSERT INTO Person VALUES (62, 'Müller', 'HRO')

erzeugen 3 neue Zeilen in der Tabelle 'Person':

PNR	Name	Ort
21	Meier	HRO
45	Weber	-
62	Müller	HRO

- jede Zeile in der Tabelle repräsentiert ein Objekt vom Typ Person

Recherche in Tabellen

Syntax

SelectStatement ::=

SELECT *SelectList*
FROM *TableReference*
WHERE *SeachCondition*

- Eine Select-Anfrage erzeugt als Antwort wiederum eine Tabelle
- *SelectList*: Kommagetrennte Liste von Spaltennamen der Ergebnistabelle
- *TableReference*: Relation(en), auf die sich die Anfrage bezieht. Die Spaltennamen der *SelectList* kommen in dieser/diesen Relation(en) vor
- *SearchContition*: im einfachsten Fall Vergleich eines Spaltenwertes mit einem Wert

Beispiel

SELECT Name, Ort
FROM Person
WHERE PNR > 30

Anfrage: "Gesucht sind Namen und Wohnort der Personen mit einer PNR größer 30

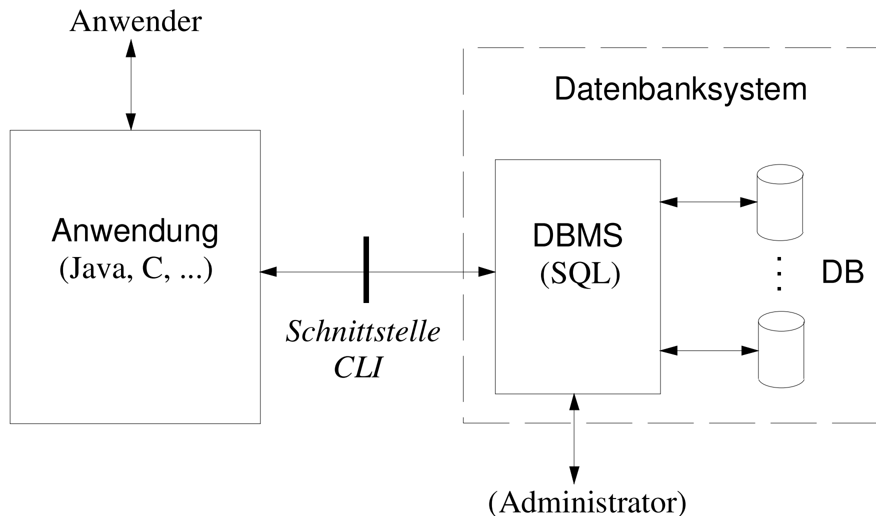
Ergbnistabelle:

Name	Ort
Weber	-
Müller	HRO

11.4 JDBC

⇒ wollen wir nun in einer Anwendung Daten in einer Datenbank speicher oder auf eine bereits existierende DB zugreifen haben wir es mit einem Problem zu tun

- **Persistierung der Daten in Datenbanken**



- Zugriff aus der Anwendung auf die Datenbank nur über SQL

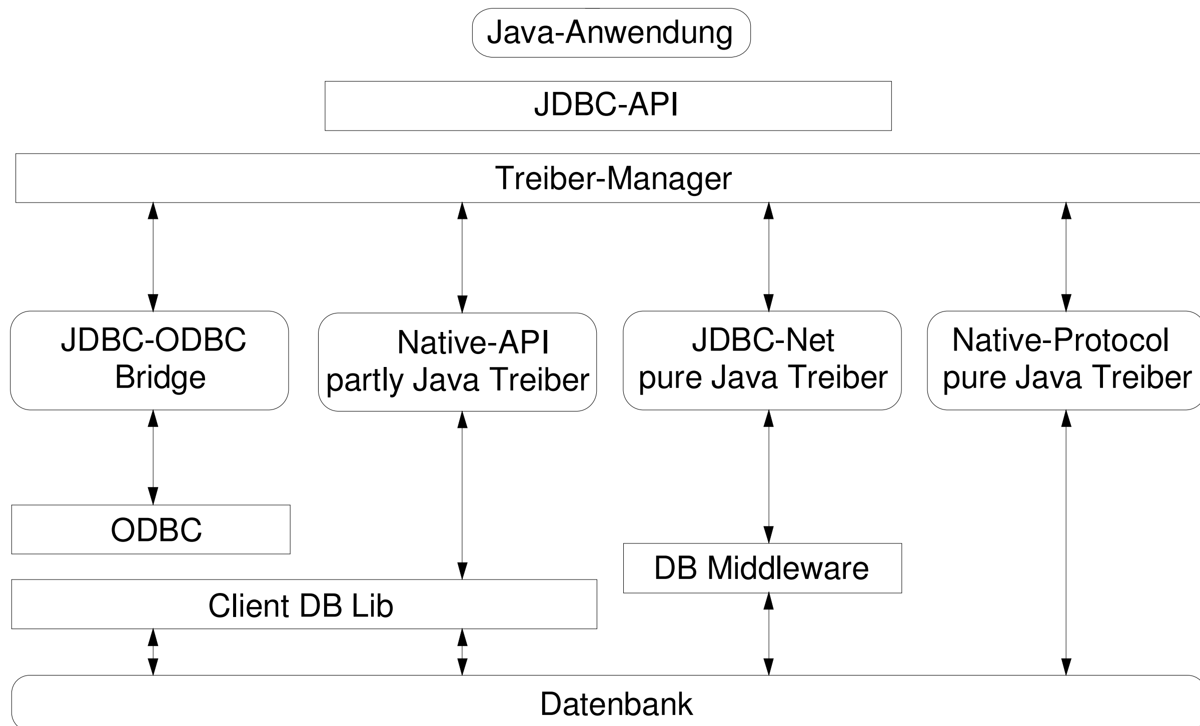
- Problem:
  - algorithmische Problemlösung in der Anwendung
    - Programmmiersprache C/C++/C#, Java ...
    - Bearbeitung einzelner Werte)
    - 2-wertige Logik
  - algebraische Problemlösung in der Datenbank
    - Datenbanksprache: SQL)
    - Bearbeitung von Wertemengen
    - 3-wertige Logik (NULL-Werte)

- Zwei Lösungsansätze
  1. Einbettung von SQL-Anweisungen in Programm (Embedded SQL, Präcompilation)
  2. Call Level Interface (ODBC, JDBC)

- **JDBC (Java Data Base Connectivity)**
  - JDBC-Treiber: Schnittstelle zwischen einer relationalen Datenbank und einer Java-Anwendung
  - JDBC-API: Menge von Java-Klassen und -Schnittstellen zur Ausführung von SQL-Anweisungen

11.4.1 JDBC-Treiber

- **4 Treibertypen**



- **Typ 1-Treiber**
  Treiber greift auf eine auf dem Client installierte ODBC-Schnittstelle der Datenbank zu. Damit können alle Datenquelle, die einen ODBC-Treiber besitzen, genutzt werden.

- **Typ 2-Treiber**
  Treiber greift direkt auf die einen speziellen proprietären Treiber des Datenbanksystems zu (datenbankspezifische API), der auf dem Client installiert werden muß.

- **Typ 3-Treiber**
  Komplett in Java geschrieben. JDBC-API-Befehle werden in SQL-Anweisungen übersetzt und über ein Netzwerkprotokoll an eine Middleware (CLI-Schnittstelle) des Datenbanksystems übertragen

- **Typ 4-Treiber**
  Ebenfalls komplett in Java geschrieben. JDBC-Aufrufe werden direkt in das datenbankspezifische Protokoll übersetzt

Java-Treiber des Datenbanksystems DB2

- DB2 von IBM, 2014
- Klassenbibliothek unter c:\Programme\IBM\SQLLIB\java\db2java.zip

Klasse DB2DataSource

- DB2DataSource-Klasse in db2java.zip unter dem package COM\ibm\db2\jdbc

Konstruktoren (Auszug)

`public DB2DataSource()`
Erzeugt ein Treiberobjekt DB2DataSource.

Methoden (Auszug)
dienen der Konfiguration des Treiber-Objektes

`public void setServerName(String serverName)`
Name des Datenbankservers (bei lokaler Datenbank auf dem Rechner: localhost).

`public void setPortNumber(int portNumber)`
TCP-Port für IBM DB2: 50 000.

`public void setUser(String user)`
Name eines für die Datenbank eingetragenen Nutzers

`public void setPassword(String password)`
Passwort des für die Datenbank eingetragenen Nutzers

`public void setDatabaseName(int databaseName)`
Name einer existierenden DB2-Datenbank.

`public Connection getConnection()`
Erzeugt ein Verbindungsobjekt der Klasse Connection als Basis für den Datenbankzugriff

Beispiel 11.4a

```
// Aufbau einer Verbindung zur Datenbank 'Java_DB'
import java.sql.*;
import COM.ibm.db2.jdbc.*;
public class Programm {
  public static void main(String[] args) throws Exception{
    DB2DataSource ds = new DB2DataSource();
    ds.setServerName("localhost");      // kann auch weggelassen werden
    ds.setUser("MyName");
    ds.setPassword("MyPassword");
    ds.setPortNumber(50000);            // kann auch weggelassen werden
    ds.setDatabaseName("Java_DB");

    Connection con = ds.getConnection();

    con.close();
  }
}
```

- Um das package *COM.ibm.db2.jdbc* importieren zu können, muß die Bibliothek *db2java* in Eclipse als Referenced Library eingebunden werden:
  - Rechter Mausklick auf das Anwendungsprojekt Package-Explorer
  - Build Path --> Add External Archives...
  - dort c:\Programme\IBM\SQLLIB\java\db2java.zip öffnen

11.4.2 JDBC-API

- die JDBC-API ist Bestandteil der JDK in den Paketen
  - ***java.sql***
    enthält die grundlegenden Klassen und Schnittstellen zur Datenbankarbeit
  - ***javax.sql***
    erweitert die Funktionalität

- wichtigste Schnittstellen aus java.sql
  - ***java.sql.Connection***
    repräsentiert eine Datenbank-Verbindung
  - ***javax.sql.Statement***
    ermöglicht Ausführung einer SQL-Anweisung
  - ***javax.sql.ResultSet***
    zeilenweise Auswertung der Ergebnistabelle einer SELECT-Anweisung

- Ablauf eines Datenbankzugriffs
  1. Treiber laden und Verbindungsaufbau zur Datenbank (Erzeugen eines Verbindungs-Objektes)
  2. Erzeugen eines Anweisungsobjektes für das Verbindungsobjekt
  3. Absetzen einer SQL-Anweisung für dieses Anweisungsobjekt
  4. bei SELECT-Anweisung: Auswertung der Ergebnistabelle
  5. Schließen des Anweisungsobjektes
  6. Schließen des Verbindunhgsobjektes

Interface Connection

```
public Statement createStatement()
```

Erzeugt ein Statement-Objekt, mit dem SQL-Anweisungen an die Datenbank geschickt werden können

Interface Statement

```
public int executeUpdate(String sql)
```

Führt eine INSERT-, UPDATE-, DELETE- oder CREATE TABLE-Anweisung aus
liefert die Anzahl der eingefügten, geänderten oder gelöschten Zeilen zurück
Bei einer CREATE TABLE-Anweisung wird 0 zurück geliefert

`public ResultSet executeQuery(String sql)`

Führt eine SELECT-Anweisung aus und liefert eine Ergebnistabelle als Objekt vom Typ ResultSet

Beispiel 11.4b

```
// Erzeugen eine (leeren) Tabelle 'Person' in der existierenden Datenbank
Java_DB
import java.sql.*;
import COM.ibm.db2.jdbc.*;
public class Programm {
  public static void main(String[] args) throws Exception{
    DB2DataSource ds = new DB2DataSource();
    ds.setUser("MyName");
    ds.setPassword("MyPassword");
    ds.setDatabaseName("Java_DB");

    Connection con = ds.getConnection();

    Statement stmt = con.createStatement();

    stmt.executeUpdate(
      "CREATE TABLE Person "+
      "(PNR      SMALLINT, "+
      " Name    VARCHAR(10), "+
      " Vorname  VARCHAR(10), "+
      " Ort      VARCHAR(3), "+
      " Vater   SMALLINT, "+
      " Mutter  SMALLINT)");

    stmt.close();
    con.close();
  }
}
```

Beispiel 11.4c

```
import java.sql.*;
import COM.ibm.db2.jdbc.*;
public class Programm {
  public static void main(String[] args) throws Exception{
    DB2DataSource ds = new DB2DataSource();
    ds.setUser("MyName");
    ds.setPassword("MyPassword");
    ds.setDatabaseName("Java_DB");

    Connection con = ds.getConnection();

    Statement stmt = con.createStatement();

    int Anzahl = 0;
    Anzahl += stmt.executeUpdate(
```

```
                "INSERT INTO Person VALUES (21,'Meier','Udo','HRO',NULL,NULL)");
      Anzahl += stmt.executeUpdate(
                "INSERT INTO Person VALUES (62,'Müller','Uta','HRO',NULL,NULL)");
      Anzahl += stmt.executeUpdate(
                "INSERT INTO Person VALUES (55,'Meier','Elke','HRO',NULL,NULL)");
      Anzahl += stmt.executeUpdate(
                "INSERT INTO Person VALUES (37,'Meier','Nora','HST',21,55)");
      Anzahl += stmt.executeUpdate(
                "INSERT INTO Person VALUES (42,'Schulz','Björn','HST',NULL,62)");
      Anzahl += stmt.executeUpdate(
                "INSERT INTO Person VALUES (45,'Weber','Klaus','HGW',21,62)");

      System.out.println(Anzahl+" Zeilen hinzugefügt");

      stmt.close();
      con.close();
   }
}
```

<u>Ausgabe</u>

```
6 Zeilen hinzugefügt
```

Interface ResultSet

`public boolean next()`

Verschiebt einen Cursor auf die nächste Zeile einer Ergebnistabelle. Nach Absenden einer SELECT-Anweisung steht der Cursor vor der ersten Zeile der Ergebnistabelle. Ist keine Zeile mehr vorhanden, so wird false zurückgegeben, sonst true

`public String getString(int columnIndex)`
`public String getString(String columnName)`

liefert den Wert der Spalte mit dem angegebenen Index (Zählung von 1) oder dem Spaltennamen für SQL-Typen VARCHAR und CHAR

`public int getInt(int columnIndex)`
`public int getInt(String columnName)`

liefert den Wert der Spalte mit dem angegebenen Index (Zählung von 1) oder dem Spaltennamen für SQL-Typen SMALLINT und INTEGER

`public boolean wasNull()`

Liefert true, wenn der als letztes durch eine get-Methode aus der Datenbank übertragene Wert NULL war

Beispiel 11.4d

```
// Gesucht sind PNR, Name und Vorname der Rostocker Personen
import java.sql.*;
import COM.ibm.db2.jdbc.*;
public class Programm {
   public static void main(String[] args) throws Exception{
```

```
      DB2DataSource ds = new DB2DataSource();
      ds.setUser("MyName");
      ds.setPassword("MyPassword");
      ds.setDatabaseName("Java_DB");

      Connection con = ds.getConnection();

      Statement stmt = con.createStatement();

      ResultSet rs = stmt.executeQuery(
           "SELECT PNR, Name, Vorname FROM Person WHERE Ort='HRO'");

      while(rs.next())
        System.out.printf("%d  %s %s \n",rs.getInt(1),
                            rs.getString("Vorname"), rs.getString("Name") );

      rs.close();
      stmt.close();
      con.close();
   }
}
```

Ausgabe

```
21  Udo Meier
62  Uta Müller
55  Elke Meier
```

Beispiel 11.4e(1)

```
// Gesucht sind Name und Vorname der Stralsunder Personen und die ihrer
Mütter und Väter
import java.sql.*;
import COM.ibm.db2.jdbc.*;
public class Programm {
   public static void main(String[] args) throws Exception{
      DB2DataSource ds = new DB2DataSource();
      ds.setUser("MyName");
      ds.setPassword("MyPassword");
      ds.setDatabaseName("Java_DB");

      Connection con = ds.getConnection();

      Statement stmt1 = con.createStatement();
      Statement stmt2 = con.createStatement();

      ResultSet rs1 = stmt1.executeQuery(
             "SELECT Name, Vorname, Vater, Mutter FROM Person WHERE Ort='HST'");
      while(rs1.next()){
        System.out.printf("%s %s\n",rs1.getString(2), rs1.getString(1));

        int vater=rs1.getInt(3);
        int mutter=rs1.getInt(4);

        ResultSet rs2;
```

```
        rs2 = stmt2.executeQuery(
            "SELECT Name, Vorname FROM Person WHERE PNR="+vater);
        if(rs2.next())
          System.out.printf("   Vater: %s %s\n",rs2.getString(2),rs2.getString(1));
        else
          System.out.printf("   Vater: unbekannt\n");

        rs2 = stmt2.executeQuery(
            "SELECT Name, Vorname FROM Person WHERE PNR="+mutter);
        if(rs2.next())
          System.out.printf("   Mutter: %s %s\n",rs2.getString(2),rs2.getString(1));
        else
          System.out.printf("   Mutter: unbekannt\n");
        rs2.close();
      }
      rs1.close();
      stmt1.close();
      stmt2.close();
      con.close();
    }
  }
```

<u>Ausgabe</u>

```
Nora Meier
   Vater: Udo Meier
   Mutter: Elke Meier
Björn Schulz
   Vater: unbekannt
   Mutter: Uta Müller
```

Beispiel 11.4e(2)

```
// Gesucht sind Name und Vorname der Stralsunder Personen und die ihrer
Mütter und Väter
import java.sql.*;
import COM.ibm.db2.jdbc.*;
public class Programm {
  public static void main(String[] args) throws Exception{
    DB2DataSource ds = new DB2DataSource();
    ds.setUser("MyName");
    ds.setPassword("MyPassword");
    ds.setDatabaseName("Java_DB");

    Connection con = ds.getConnection();

    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(
      "SELECT x.Vorname, x.Name, y.Vorname, y.Name, z.Vorname, z.Name "+
      "FROM Person x LEFT JOIN Person y ON x.Vater=y.PNR "+
                     LEFT JOIN Person z ON x.Mutter=z.PNR "+
      "WHERE x.Ort='HST'");

    while(rs.next()){
      System.out.printf("%s %s\n",rs.getString(1),rs.getString(2));
```

```
        rs.getString(3);
        if(rs.wasNull())
          System.out.printf("    Vater: unbekannt\n");
        else
          System.out.printf("    Vater: %s %s\n",rs.getString(3), rs.getString(4));

        rs.getString(5);
        if(rs.wasNull())
          System.out.printf("    Mutter: unbekannt\n");
        else
          System.out.printf("    Mutter: %s %s\n",rs.getString(5), rs.getString(6));
      }
      rs.close();
      stmt.close();
      con.close();
    }
  }
```

--