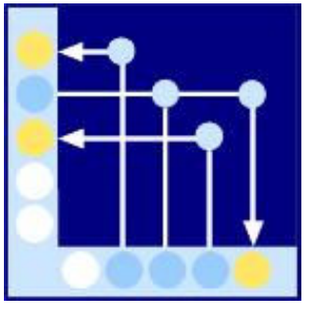
Fakultät Elektronik und Informatik
Studiengang Informatik



Programmieren 2

Objektorientierte Programmierung mit Java

Vorlesung im Wintersemester 2014/2015

Prof. Dr. habil. Christian Heinlein

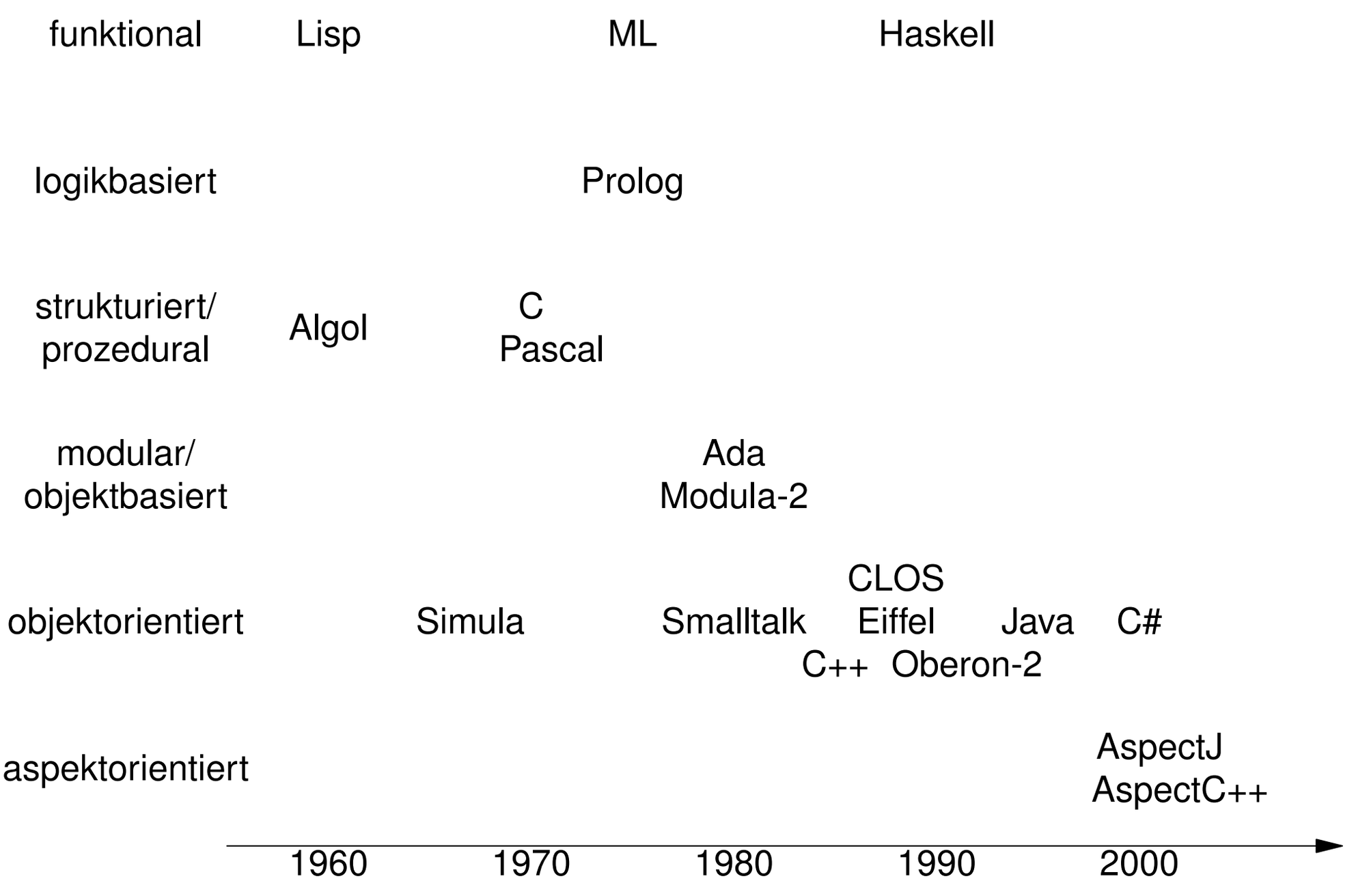
christian.heinleins.net

1 Einleitung und Überblick

1.1 Vorlesungsüberblick

- objektorientierte Programmierung in C
- Gemeinsamkeiten und Unterschiede zwischen C und Java
- objektorientierte Programmierung in Java
  - Klassen mit Feldern, Methoden und Konstruktoren
  - Kapselung, Geheimnisprinzip
  - Unterklassen, Vererbung, Polymorphie, dynamisches Binden
  - abstrakte Klassen und Schnittstellen
  - Pakete
  - Ausnahmen
  - Java-Standardbibliothek
- Unified Modeling Language (UML)

1.2 Bekannte Programmiersprachen und -paradigmen



1.3 Literaturhinweise

- K. Arnold, J. Gosling, D. Holmes: *The Java Programming Language* (Fourth Edition). Addison-Wesley, Amsterdam, 2005.
  (Von den „Vätern“ der Sprache. Auch auf Deutsch: *Die Programmiersprache Java*.)

- J. Gosling, B. Joy, G. Steele, G. Bracha: *The Java Language Specification* (Third Edition). Addison-Wesley, Amsterdam, 2005.
  (Als Nachschlagewerk für erfahrene Programmierer.)

- D. Flanagan: *Java in a Nutshell* (Fifth Edition). O’Reilly, 2005.
  (Für eine „Nutshell“ ziemlich dick.)

- Zahllose weitere Bücher über Java . . .

2 Objektorientierte Programmierung in C

2.1 Beispiel: Bankkonten

2.1.1 Aufgabenstellung

- Definieren Sie eine Datenstruktur `Account` zur Repräsentation von Bankkonten mit Kontonummer, Kontoinhaber und aktuellem Kontostand!
- Implementieren Sie eine Funktion `newAccount`, die ein dynamisches Objekt des Typs `Account` erzeugt, es geeignet mit den als Parameter übergebenen Werten initialisiert und einen Zeiger auf das Objekt zurückliefert!
- Implementieren Sie Funktionen `number`, `holder` und `balance`, um die Kontonummer, den Kontoinhaber und den aktuellen Kontostand eines bestimmten Kontos abzufragen!
  (Benutzer des Typs `Account` sollen auf die entsprechenden Datenfelder nicht direkt zugreifen, weil sie z. B. nicht beliebig geändert werden dürfen; Geheimnisprinzip!)
- Implementieren Sie Funktionen `deposit` und `withdraw`, um den Kontostand eines bestimmten Kontos um einen bestimmten Betrag zu erhöhen bzw. zu erniedrigen, sowie eine Funktion `transfer`, um Geld von einem Konto auf ein anderes zu überweisen!

2.1.2 Datenstruktur

```
/* Zeichenkette variabler Länge. */
/* (»typedef« definiert den Namen »String« */
/* als Abkürzung für den Typ »const char*«.) */
typedef const char* String;

/* Konto. */
struct Account {
  long number;          /* Kontonummer. */
  String holder;        /* Kontoinhaber. */
  long balance;         /* Kontostand in Cent. */
};

/* Verwendung von long, weil int eventuell nur 16 Bit groß ist. */
```

2.1.3 Konten erzeugen und initialisieren

```
#include <stdlib.h>      /* malloc, exit, NULL */
#include <stdio.h>       /* printf */

/* Nächste zu vergebende Kontonummer. */
long nextNumber = 1;

/* Konto mit Inhaber h, eindeutiger Nummer */
/* und Anfangsbetrag 0 erzeugen. */
struct Account* newAccount (String h) {
  struct Account* this = malloc(sizeof(struct Account));
  if (this == NULL) {
    printf("newAccount: out of memory\n");
    exit(1);
  }
  this->number = nextNumber++;
  this->holder = h;
  this->balance = 0;
  return this;
}
```

2.1.4 Kontodaten abfragen

```
/* Nummer von Konto this liefern. */
long number (struct Account* this) {
  return this->number;
}

/* Inhaber von Konto this liefern. */
String holder (struct Account* this) {
  return this->holder;
}

/* Kontostand von Konto this liefern. */
long balance (struct Account* this) {
  return this->balance;
}
```

2.1.5 Geld einzahlen, abheben und überweisen

```
/* Betrag amount auf Konto this einzahlen. */
void deposit (struct Account* this, long amount) {
  this->balance += amount;
}

/* Betrag amount von Konto this abheben. */
void withdraw (struct Account* this, long amount) {
  this->balance -= amount;
}

/* Betrag amount von Konto this auf Konto that überweisen. */
void transfer (struct Account* this, long amount,
                                     struct Account* that) {
  withdraw(this, amount);
  deposit(that, amount);
}
```

2.1.6 Anwendungsbeispiel

```
/* Zeiger auf Kontoobjekt. */
typedef struct Account* Account;

int main () {
  /* Zwei Konten erzeugen. */
  Account a = newAccount("Hans Maier");
  Account b = newAccount("Fritz Müller");

  /* 1000 Cent auf Konto a einzahlen, */
  /* dann 300 Cent auf Konto b überweisen. */
  deposit(a, 1000);
  transfer(a, 300, b);

  /* Kontodaten ausgeben. */
  printf("Konto a: %ld, %s, %ld\n",
   number(a), holder(a), balance(a));
  printf("Konto b: %ld, %s, %ld\n",
   number(b), holder(b), balance(b));

  return 0;
}
```

2.2 Limitierte Konten

2.2.1 Aufgabenstellung

- Definieren Sie eine weitere Datenstruktur `LimitedAccount` zur Repräsentation von limitierten Konten, d. h. Konten, die nur bis zu einer bestimmten Kreditlinie überzogen werden dürfen!
- Implementieren Sie analog zu `newAccount` eine Funktion `newLimitedAccount`, die ein dynamisches Objekt des Typs `LimitedAccount` erzeugt und geeignet initialisiert!
- Implementieren Sie analog zu `number` etc. eine Funktion `limit`, um die Kreditlinie eines limitierten Kontos abzufragen!
- Sorgen Sie dafür, dass ein limitiertes Konto überall verwendet werden kann, wo ein allgemeines Konto erwartet wird! Insbesondere sollen alle für `Account` definierten Funktionen (wie z. B. `number` oder `transfer`) auch für `LimitedAccount`-Objekte aufrufbar sein!
- Redefinieren Sie die Funktionen `withdraw` und `transfer` für limitierte Konten dahingehend, dass die Operation nur dann ausgeführt wird, wenn dadurch die Kreditlinie nicht überschritten wird! (Andernfalls soll eine Fehlermeldung ausgegeben werden.)

2.2.2 Datenstruktur

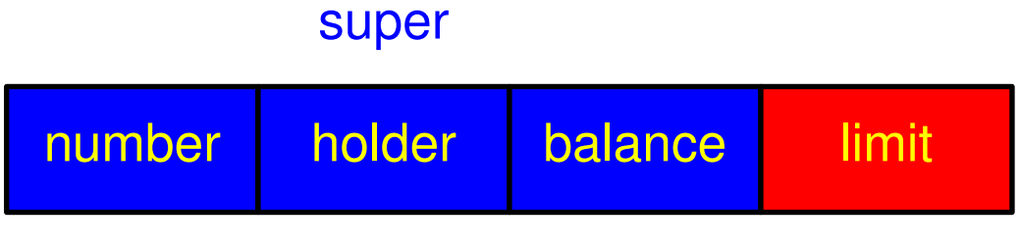
Problem

- `struct LimitedAccount` sollte von `struct Account` „erben“, d. h. alle seine Datenfelder besitzen, ohne dass deren Definition wiederholt werden muss.

Lösung

- `struct LimitedAccount` besitzt als erste Komponente ein „Teilobjekt“ des Typs `struct Account`:

```
/* Limitiertes Konto. */
struct LimitedAccount {
  struct Account super; /* Account-Daten. */
  long limit;           /* Kreditlinie in Cent (positiv). */
};
```



2.2.3 Limitierte Konten erzeugen und initialisieren

Problem

- `newLimitedAccount` muss, ebenso wie `newAccount`:
  - mittels `malloc` ein dynamisches Objekt erzeugen,
  - den Resultatwert von `malloc` überprüfen und ggf. eine Fehlermeldung ausgeben,
  - `number`, `holder` und `balance` initialisieren.
- Andererseits soll das „oberste Gebot der Programmierung“ beachtet werden:
  „Du sollst nicht Code verdoppeln!“

Lösung

- Die wiederverwendbaren Teile von `newAccount` werden in Hilfsfunktionen `newObject` und `initAccount` ausgelagert:

```
/* Dynamisches Objekt der Größe n Byte erzeugen. */
void* newObject (int n) {
  void* this = malloc(n);
  if (this == NULL) {
    printf("newObject: out of memory\n");
    exit(1);
  }
  return this;
}
```

```
/* Nächste zu vergebende Kontonummer. */
long nextNumber = 1;

/* Konto this mit Inhaber h, eindeutiger Nummer */
/* und Anfangsbetrag 0 initialisieren. */
void initAccount (struct Account* this, String h) {
  this->number = nextNumber++;
  this->holder = h;
  this->balance = 0;
}

/* Konto mit Inhaber h, eindeutiger Nummer */
/* und Anfangsbetrag 0 erzeugen. */
struct Account* newAccount (String h) {
  struct Account* this = newObject(sizeof(struct Account));
  initAccount(this, h);
  return this;
}
```

- Die Initialisierungen von `newLimitedAccount` werden ebenfalls in eine Hilfsfunktion `initLimitedAccount` ausgelagert, die ihrerseits `initAccount` (mit Typecast von `struct LimitedAccount*` auf `struct Account*`) aufruft:

```
/* Limitiertes Konto this mit Inhaber h, Kreditlinie l, */
/* eindeutiger Nummer und Anfangsbetrag 0 initialisieren. */
void initLimitedAccount (struct LimitedAccount* this,
                                              String h, long l) {
  initAccount((struct Account*)this, h);
  this->limit = l;
}

/* Limitiertes Konto mit Inhaber h, Kreditlinie l, */
/* eindeutiger Nummer und Anfangsbetrag 0 erzeugen. */
struct LimitedAccount* newLimitedAccount (String h, long l) {
  struct LimitedAccount* this =
                        newObject(sizeof(struct LimitedAccount));
  initLimitedAccount(this, h, l);
  return this;
}
```

2.2.4 Kreditlinie abfragen

```
/* Kreditlinie des limitierten Kontos this liefern. */
long limit (struct LimitedAccount* this) {
  return this->limit;
}
```

2.2.5 Ersetzbarkeit von Konten durch limitierte Konten

Problem

- Obwohl ein Zeiger auf ein `LimitedAccount`-Objekt technisch gleichzeitig ein Zeiger auf das `Account`-Teilobjekt `super` dieses Objekts ist, sind die entsprechenden Zeigertypen logisch verschieden, sodass ein Zeigerwert des Typs `struct LimitedAccount*` nicht an eine Variable oder einen Parameter des Typs `struct Account*` zugewiesen werden kann.

- Daher können die für `Account` definierten Funktionen (wie z. B. `number` oder `transfer`) nicht für `LimitedAccount`-Objekte aufgerufen werden.

Lösung

- Die Zeigertypen `Account` und `LimitedAccount` werden beide als Synonyme des generischen Zeigertyps `void*` definiert:

```
typedef void* Account;
typedef void* LimitedAccount;
```

- Da `void*` mit jedem anderen Zeigertyp kompatibel ist, können nun sowohl `Account`- als auch `LimitedAccount`-Objekte an Parameter des Typs `struct Account*` übergeben werden.

- Daher können die für `Account` definierten Funktionen – die jeweils Parameter des Typ `struct Account*` besitzen – sowohl für `Account`- als auch für `LimitedAccount`-Objekte aufgerufen werden.

2.2.6 Überschreiben und dynamisches Binden von Funktionen

Problem

- Obwohl die Funktionen `withdraw` und `transfer` auch für `LimitedAccount`-Objekte aufrufbar sein sollen – damit ein `LimitedAccount`-Objekt überall verwendet werden kann, wo ein `Account`-Objekt erwartet wird –, soll ihr Verhalten für `LimitedAccount`-Objekte anders sein als für `Account`-Objekte.

- Allgemein gesprochen, soll das Verhalten bestimmter Funktionen davon abhängen, für welche Art von Objekten sie aufgerufen werden.

Lösung

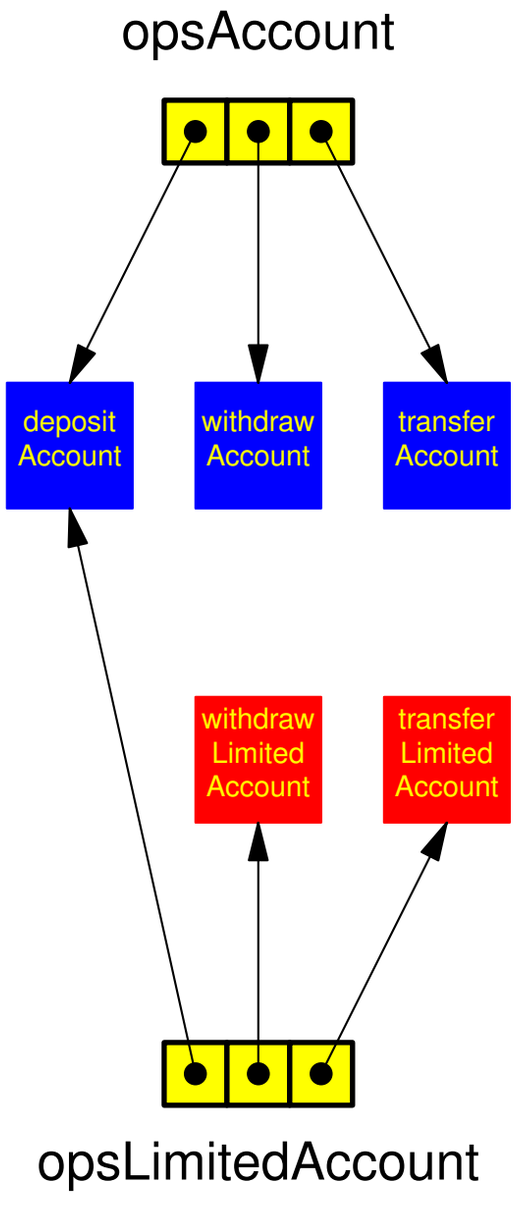
- Es gibt ggf. unterschiedliche Implementierungen der Funktionen `deposit`, `withdraw` und `transfer` für `Account`- und `LimitedAccount`-Objekte:

```
/* Betrag amount auf gewöhnliches Konto this einzahlen. */
void depositAccount (struct Account* this, long amount) {
  this->balance += amount;
}

/* Betrag amount von gewöhnlichem Konto this abheben. */
void withdrawAccount (struct Account* this, long amount) {
  this->balance -= amount;
}

/* Betrag amount von gewöhnlichem Konto this */
/* auf Konto that überweisen. */
void transferAccount (struct Account* this, long amount,
                                            struct Account* that) {
  withdraw(this, amount);
  deposit(that, amount);
}
```

```
/* Überprüfen, ob Betrag amount von limitiertem Konto this */
/* abgezogen werden kann, ohne die Kreditlinie zu überschreiten. */
int check (struct LimitedAccount* this, long amount) {
  if (this->super.balance - amount >= -this->limit) return 1;
  printf("Unzulässige Kontoüberziehung!\n");
  return 0;
}

/* Betrag amount von limitiertem Konto this abheben, falls mgl. */
void withdrawLimitedAccount (struct LimitedAccount* this,
                                                long amount) {
  if (check(this, amount)) {
    withdrawAccount((struct Account*)this, amount);
  }
}
/* Betrag amount von limitiertem Konto this */
/* auf Konto that überweisen, falls möglich. */
void transferLimitedAccount (struct LimitedAccount* this,
                                long amount, struct Account* that) {
  if (check(this, amount)) {
    transferAccount((struct Account*)this, amount, that);
  }
}
```

- Die für `Account`- bzw. `LimitedAccount`-Objekte passenden Implementierungen werden jeweils in einem *Funktionszeigersatz* zusammengefasst:

```
/* Funktionszeigersatz für Konten. */
struct AccountOps {
  void (*deposit) ();   /* Funktion für Einzahlung. */
  void (*withdraw) ();  /* Funktion für Abhebung. */
  void (*transfer) ();  /* Funktion für Überweisung. */
};

/* Funktionszeigersatz für gewöhnliche Konten. */
struct AccountOps opsAccount = {
  depositAccount,
  withdrawAccount,
  transferAccount
};

/* Funktionszeigersatz für limitierte Konten. */
struct AccountOps opsLimitedAccount = {
  depositAccount,
  withdrawLimitedAccount,
  transferLimitedAccount
};
```

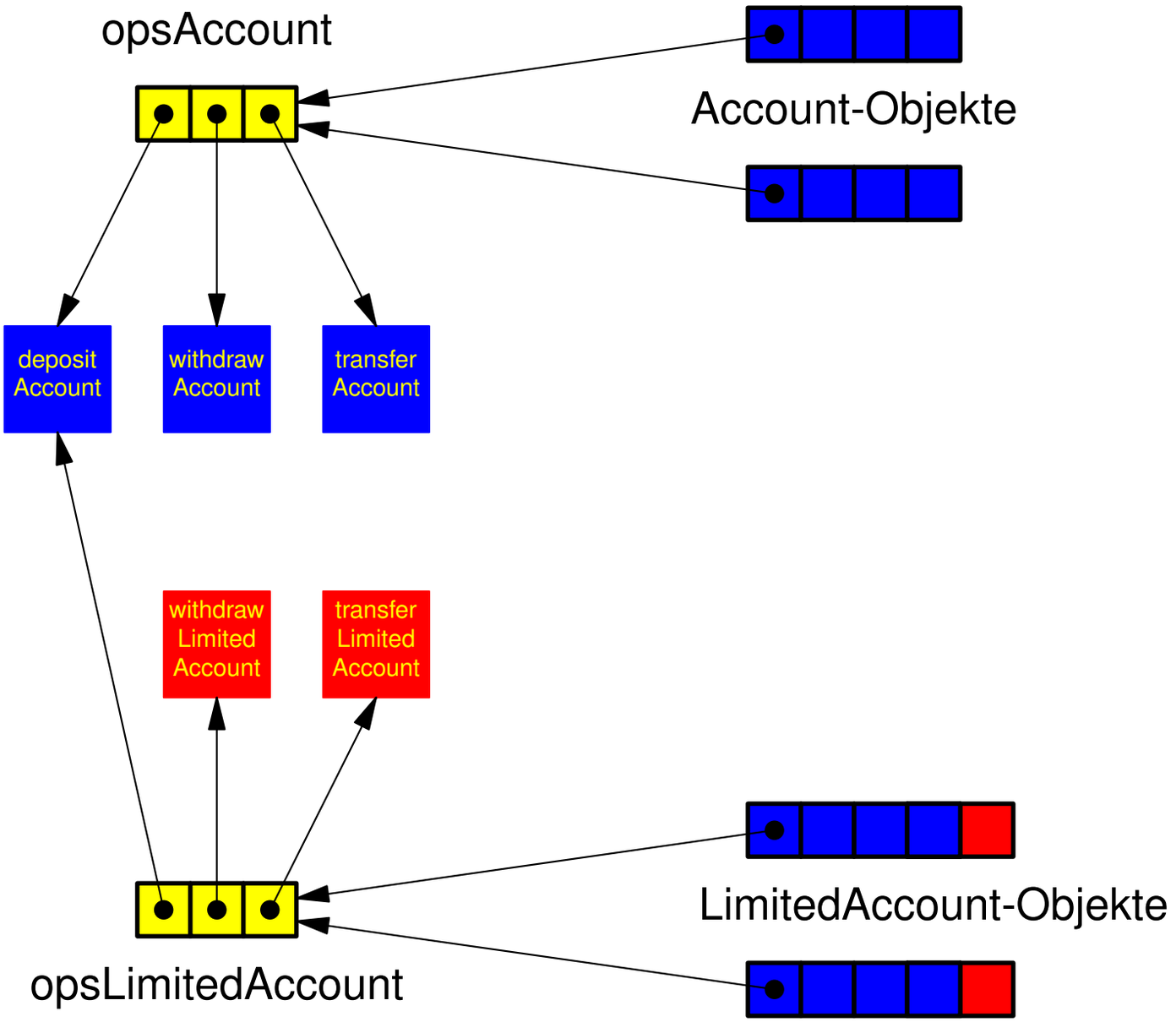

opsAccount
deposit Account
withdraw Account
transfer Account
withdraw Limited Account
transfer Limited Account
opsLimitedAccount

- `Account`-Objekte – und damit auch `LimitedAccount`-Objekte – besitzen als erste Komponente einen Zeiger auf einen Funktionszeigersatz:

```
/* Konto. */
struct Account {
  struct AccountOps* ops; /* Zeiger auf Funktionszeigersatz. */
  long number;            /* Kontonummer. */
  String holder;          /* Kontoinhaber. */
  long balance;           /* Kontostand in Cent. */
};

/* Limitiertes Konto. */
struct LimitedAccount {
  struct Account super;   /* Account-Daten (inkl. ops-Zeiger). */
  long limit;             /* Kreditlinie in Cent. */
};
```

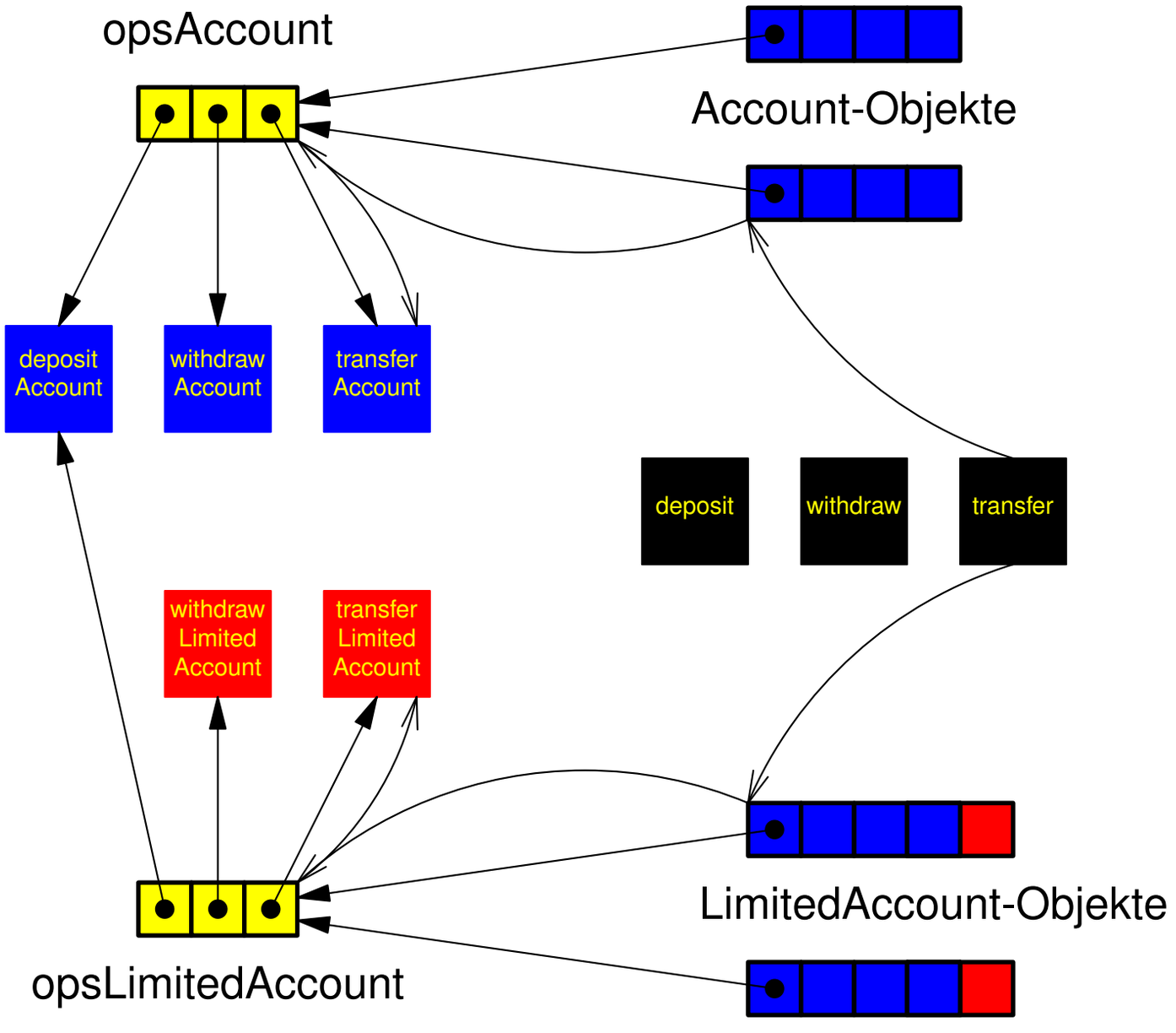
- Die „Konstruktoren“ `newAccount` und `newLimitedAccount` initialisieren den Zeiger auf den Funktionszeigersatz mit der passenden Adresse:

```
/* Konto mit Inhaber h, eindeutiger Nummer */
/* und Anfangsbetrag 0 erzeugen. */
Account newAccount (String h) {
  struct Account* this = newObject(sizeof(struct Account));
  this->ops = &opsAccount;
  initAccount(this, h);
  return this;
}

/* Limitiertes Konto mit Inhaber h, Kreditlinie l, */
/* eindeutiger Nummer und Anfangsbetrag 0 erzeugen. */
LimitedAccount newLimitedAccount (String h, long l) {
  struct LimitedAccount* this =
                        newObject(sizeof(struct LimitedAccount));
  this->super.ops = &opsLimitedAccount;
  initLimitedAccount(this, h, l);
  return this;
}
```


opsAccount
Account-Objekte
deposit Account
withdraw Account
transfer Account
withdraw Limited Account
transfer Limited Account
LimitedAccount-Objekte
opsLimitedAccount

- Die „generischen“ Funktionen `deposit`, `withdraw` und `transfer` rufen über den Funktionszeigersatz ihres Zielobjekts `this` automatisch die jeweils passende Implementierung auf:

```
/* Betrag amount auf irgendein Konto this einzahlen. */
void deposit (struct Account* this, long amount) {
  this->ops->deposit(this, amount);
}

/* Betrag amount von irgendeinem Konto this abheben. */
void withdraw (struct Account* this, long amount) {
  this->ops->withdraw(this, amount);
}

/* Betrag amount von irgendeinem Konto this */
/* auf Konto that überweisen. */
void transfer (struct Account* this, long amount, Account that) {
  this->ops->transfer(this, amount, that);
}
```


opsAccount
Account-Objekte
deposit Account
withdraw Account
transfer Account
deposit
withdraw
transfer
withdraw Limited Account
transfer Limited Account
LimitedAccount-Objekte
opsLimitedAccount

2.2.7 Anwendungsbeispiel

```
int main () {
  /* Ein limitiertes und ein normales Konto erzeugen. */
  LimitedAccount a = newLimitedAccount("Hans Maier", 500);
  Account b = newAccount("Fritz Müller");

  /* 1000 Cent auf Konto a einzahlen, */
  /* dann 300 Cent auf Konto b überweisen. */
  /* (a kann wie ein normales Konto verwendet werden.) */
  deposit(a, 1000);
  transfer(a, 300, b);

  /* Von jedem Konto 2000 Cent abheben. */
  /* Bei a würde das die Kreditlinie überschreiten. */
  withdraw(a, 2000);
  withdraw(b, 2000);
```

```
  /* Kontodaten ausgeben. */
  printf("Konto a: %ld, %s, %ld\n",
   number(a), holder(a), balance(a));
  printf("Konto b: %ld, %s, %ld\n",
   number(b), holder(b), balance(b));

  return 0;
}
```

2.3 Rückblick

- Objektorientierte Programmierung beinhaltet u. a.:
  - Datenkapselung/Geheimnisprinzip
  - Vererbung/Wiederverwendung von Daten und Operationen
  - Überschreiben und dynamisches Binden von Operationen
- Objektorientierte Programmierung mit C ist prinzipiell möglich, aber mühsam:
  - explizite Verwaltung von Funktionszeigersätzen
  - umständlicher Zugriff auf „geerbte“ Datenfelder
  - mechanisch zu generierender Code
- Außerdem sind ein paar Tricks erforderlich, um zu strenge Typprüfungen des Compilers zu umgehen:
  - `void*` als Zeiger auf `Account`- und `LimitedAccount`-Objekte, damit `LimitedAccount`-Objekte als `Account`-Objekte verwendbar sind. Kehrseite: `Account`-Objekte können auch als `LimitedAccount`-Objekte verwendet werden, was jedoch zu undefiniertem Verhalten führt.
  - Bewusst ungenaue Typen in Funktionszeigersätzen, z. B. `void (*withdraw) ()` mit beliebiger Parameterliste statt `void (*withdraw) (struct Account*, long)`.

2.4 Implementierung in Java

Allgemeine Konten

```
// Klasse: Konto.
class Account {
  // Objektvariablen:
  private int number;           // Kontonummer.
  private String holder;        // Kontoinhaber.
  private int balance;          // Kontostand.

  // Klassenvariable: Nächste zu vergebende Kontonummer.
  private static int nextNumber = 1;

  // Konstruktor:
  // Konto mit Inhaber h, eindeutiger Nummer
  // und Anfangsbetrag 0 konstruieren.
  public Account (String h) {
    this.number = nextNumber++;
    this.holder = h;
    this.balance = 0;
  }
```

```
  // Objektmethoden: Kontonummer/-inhaber/-stand abfragen.
  public int number () {
    return this.number;
  }
  public String holder () {
    return this.holder;
  }
  public int balance () {
    return this.balance;
  }

  // Objektmethoden: Betrag amount einzahlen/abheben/überweisen.
  public void deposit (int amount) {
    this.balance += amount;
  }
  public void withdraw (int amount) {
    this.balance -= amount;
  }
  public void transfer (int amount, Account that) {
    this.withdraw(amount);
    that.deposit(amount);
  }
}
```

Limitierte Konten

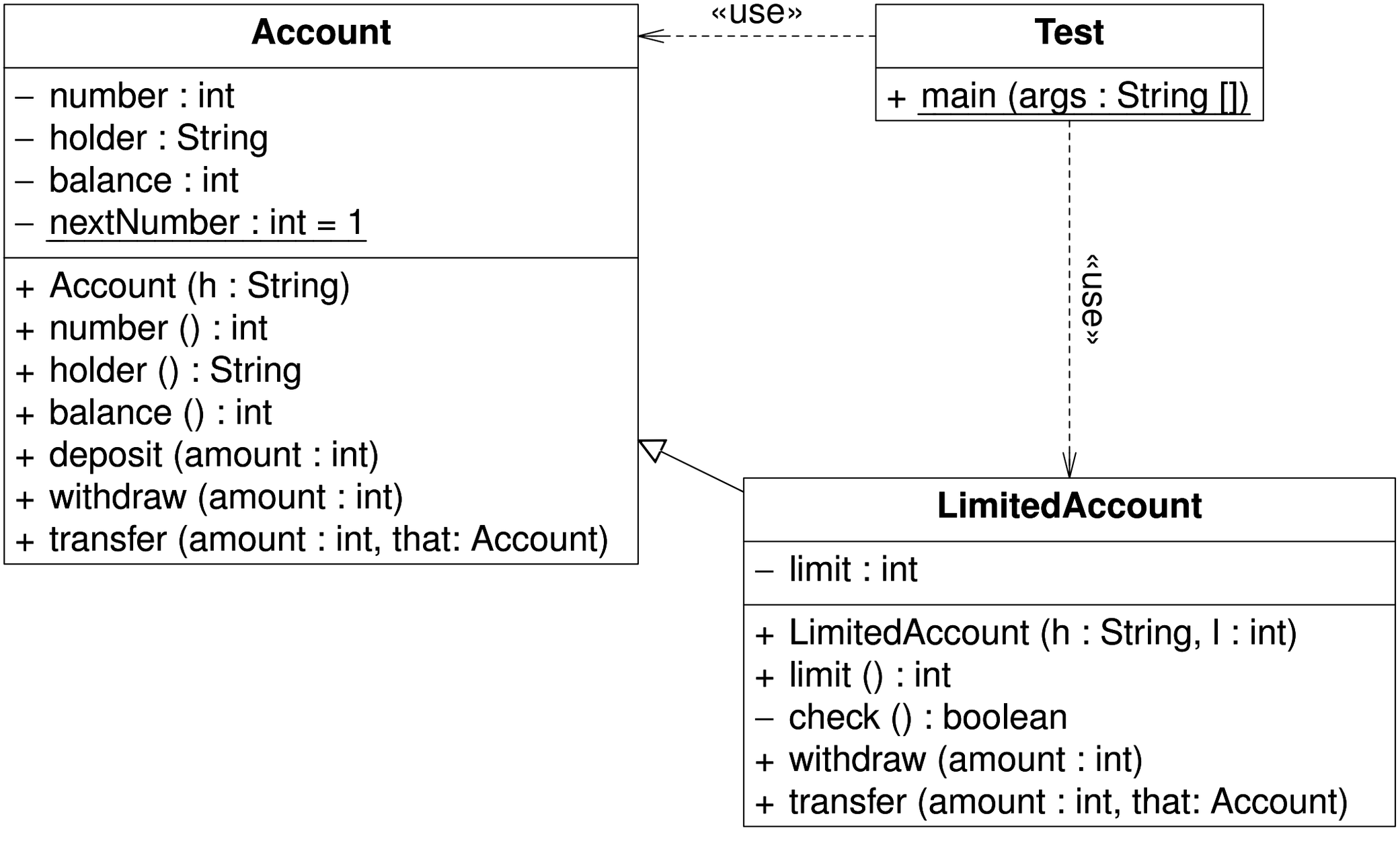
```
// Unterklasse von Account: Limitiertes Konto.
class LimitedAccount extends Account {
  // Zusätzliche Objektvariable:
  private int limit;              // Kreditlinie in Cent.

  // Konstruktor:
  // Limitiertes Konto mit Inhaber h, Kreditlinie l,
  // eindeutiger Nummer und Anfangsbetrag 0 konstruieren.
  public LimitedAccount (String h, int l) {
    // Konstruktor der Oberklasse Account aufrufen,
    // um deren Objektvariablen zu initialisieren.
    super(h);

    limit = l;
  }

  // Zusätzliche Objektmethode: Kreditlinie abfragen.
  public int limit () { return limit; }
```

```
  // Hilfsmethode: Kann Betrag amount abgezogen werden,
  // ohne die Kreditlinie zu überschreiten?
  private boolean check (int amount) {
    if (balance() - amount >= -limit) return true;
    System.out.println("Unzulässige Kontoüberziehung!");
    return false;
  }

  // Überschreiben geerbter Objektmethoden:
  // Betrag amount abheben/überweisen.
  public void withdraw (int amount) {
    if (check(amount)) {
      // Überschriebene Methode aufrufen.
      super.withdraw(amount);
    }
  }
  public void transfer (int amount, Account that) {
    if (check(amount)) {
      // Überschriebene Methode aufrufen.
      super.transfer(amount, that);
    }
  }
}
```

Test

```
class Test {
  // Hauptprogramm.
  public static void main (String [] args) {
    // Objekte erzeugen und durch Konstruktoraufrufe initialisieren.
    LimitedAccount a = new LimitedAccount("Hans Maier", 500);
    Account b = new Account("Fritz Müller");

    // Methoden auf Objekten aufrufen.
    a.deposit(1000);
    a.transfer(300, b);

    a.withdraw(2000);
    b.withdraw(2000);

    // Ausgabe.
    System.out.println("Konto a: " + a.number() + " " +
     a.holder() + " " + a.balance());
    System.out.println("Konto b: " + b.number() + " " +
     b.holder() + " " + b.balance());
  }
}
```

2.5 Darstellung als UML-Klassendiagramm



3 Gemeinsamkeiten und Unterschiede zwischen Java und C

3.1 Äußerlichkeiten

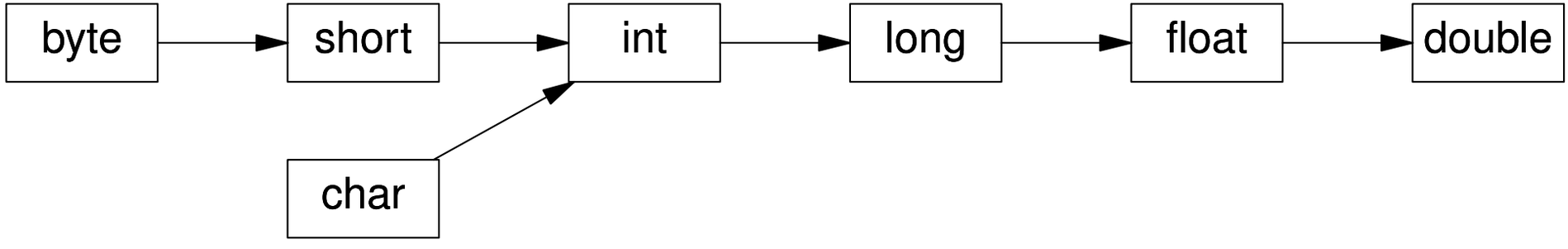
- Kommentare in Java erstrecken sich entweder (wie in C) von `/*` bis `*/` („Blockkommentare“) oder von `//` bis zum Zeilenende („Zeilenkommentare“).
- Blockkommentare können (wie in C) nicht verschachtelt werden, d. h. ein Blockkommentar endet immer beim ersten Auftreten von `*/`.
- Wenn man „echte“ Kommentare als Zeilenkommentare formuliert, kann man Blockkommentare zum „Auskommentieren“ von Programmabschnitten verwenden.
- In Java gibt es keinen Präprozessor, d. h. insbesondere kein `#include` und kein `#define`.
- In C besitzt die Hauptfunktion `main` Resultattyp `int` (der Resultatwert stellt den Exitstatus des Programms dar, der an das Betriebssystem zurückgegeben wird), in Java ist der Resultattyp `void` (der Exitstatus ist standardmäßig 0).
  Auch die Parameterliste von `main` ist unterschiedlich (vgl. § 4.6.3).

3.2 Datentypen

3.2.1 Elementare Typen

Typ	*Java*	*C*
boolean	+	–
char	16 Bit	8 Bit
byte	8 Bit	entspricht `signed char`
short	16 Bit	meist 16 Bit
int	32 Bit	meist 16 oder 32 Bit
long	64 Bit	meist 32 Bit
unsigned short	vgl. `char`	Größe wie `short`
unsigned int	–	Größe wie `int`
unsigned long	–	Größe wie `long`
float	32 Bit	meist 32 Bit
double	64 Bit	meist 64 Bit

- Ganzzahlige Arithmetik wird grundsätzlich mit `int`- oder `long`-Werten ausgeführt; „kürzere“ Werte werden ggf. automatisch expandiert.
- Umwandlungen von „kürzeren“ in „längere“ Typen erfolgen bei Bedarf implizit, während andere Umwandlungen explizit durch Casts erfolgen müssen.



- Beispiel: Umwandlung von Klein- in Großbuchstaben

```
char c = 'x'; // ... oder ein anderer Kleinbuchstabe.

// Fehler: Rechte Seite hat Typ int => Zuweisung an char verboten.
c = c + 'A' - 'a';

// Korrekt: Rechte Seite explizit in char umwandeln.
c = (char)(c + 'A' - 'a');

// Auch korrekt: Bei += etc. wird automatisch umgewandelt!
c += 'A' - 'a';
```

3.2.2 Zeiger und Referenzen

- In Java gibt es keine expliziten Zeigertypen (wie z. B. `int*` oder `int (*) ()`) und somit auch keinen Adress- (`&`) und Inhaltsoperator (`*`).
- Stattdessen sind alle nicht-elementaren Typen (d. h. Klassen, Schnittstellen und Arrays) *Referenztypen*, d. h. implizit Zeigertypen.
- Die *Nullreferenz* `null`, die mit allen Referenztypen kompatibel ist, repräsentiert eine Referenz auf „nichts“.
- Anstelle von Funktionszeigern kann man Referenzen auf Objekte verwenden, die die gewünschte „Funktion“ (d. h. Methode) besitzen (vgl. § 7.2).

3.2.3 Arrays

Eindimensionale Arrays

- Arraytypen in Java legen nur den Typ der Elemente fest, nicht jedoch ihre Anzahl, z. B.:

```
double [] a;                // Array von double-Werten.
```

- Arrayobjekte müssen zur Laufzeit explizit mit `new` erzeugt werden, z. B.:

```
a = new double [10];    // Array mit 10 double-Werten erzeugen.
```

- Anders als in C, muss die Länge bzw. Elementzahl kein konstanter Ausdruck sein, z. B.:

```
// Kommandozeilenargument args[0] in eine ganze Zahl umwandeln
// und ein Array mit entsprechend vielen double-Werten erzeugen.
a = new double [Integer.parseInt(args[0])];
```

- Bei Angabe einer negativen Elementzahl erhält man eine Ausnahme des Typs `NegativeArraySizeException`. (Die Elementzahl 0 ist zulässig.)

- Die Elementzahl eines Arrays `a` kann mittels `a.length` abgefragt werden.

- Bei der Erzeugung eines Arrays werden seine Elemente mit einem typabhängigen Standardwert initialisiert:
  - `false` für Typ `boolean`
  - `'\0'` für Typ `char`
  - `0` bzw. `0.0` für alle numerischen Typen
  - `null` für alle Referenztypen (Klassen, Schnittstellen, Arrays)

- Nach Erzeugung eines Arrays kann seine Elementzahl nicht mehr verändert werden. Um ein Array quasi zu „vergrößern“ (oder zu „verkleinern“), muss man ein neues Array mit der gewünschten Elementzahl erzeugen und die Elemente umkopieren.

- Für einen ganzzahligen Wert `i` zwischen 0 einschließlich und `a.length` ausschließlich liefert `a[i]` das `i`-te Element des Arrays `a`.

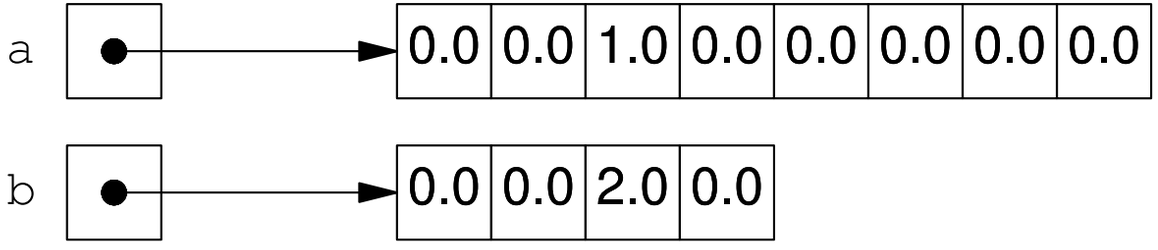
- Für andere Werte von `i` erhält man eine Ausnahme des Typs `ArrayIndexOutOfBoundsException`, d. h. es findet Bereichsprüfung statt.

- Wenn `a` die Nullreferenz `null` ist, produzieren die Ausdrücke `a.length` und `a[i]` eine Ausnahme des Typs `NullPointerException`.

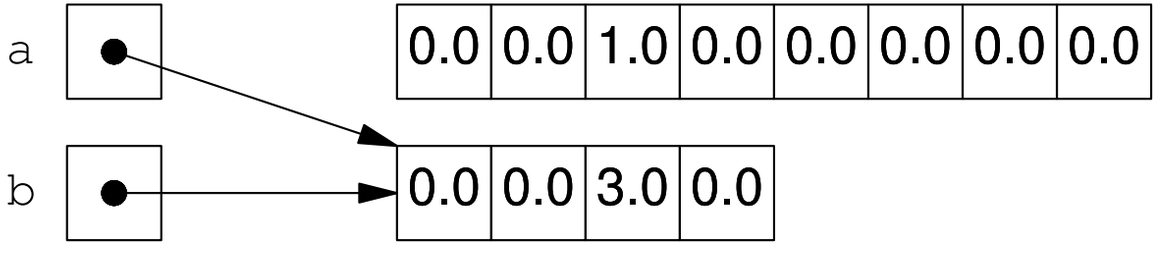
- Wenn `a` und `b` Array-Variablen sind, wird bei einer Zuweisung `a = b` lediglich die *Referenz* `b` kopiert, aber nicht die Elemente des von `b` referenzierten Arrays.

Beispiel

```
double [] a, b;
a = new double [8]; a[2] = 1.0;
b = new double [4]; b[2] = 2.0;
```




```
a = b;
a[2] = 3.0; // Gleichbedeutend mit: b[2] = 3.0;
```



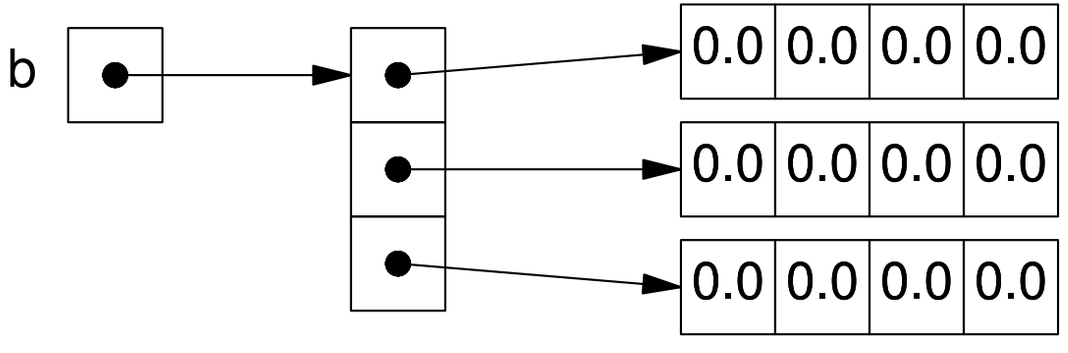

```
// Das nicht mehr referenzierte Array wird (zu einem nicht näher
// spezifizierten Zeitpunkt) vom System automatisch freigegeben.
```

Mehrdimensionale Arrays

- Mehrdimensionale Arrays erhält man indirekt als Arrays von Arrays, z. B.:

```
int m = ..., n = ...;
double [] [] b = new double [m] [n];
for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++) {
    System.out.println(b[i][j]);
  }
}
```

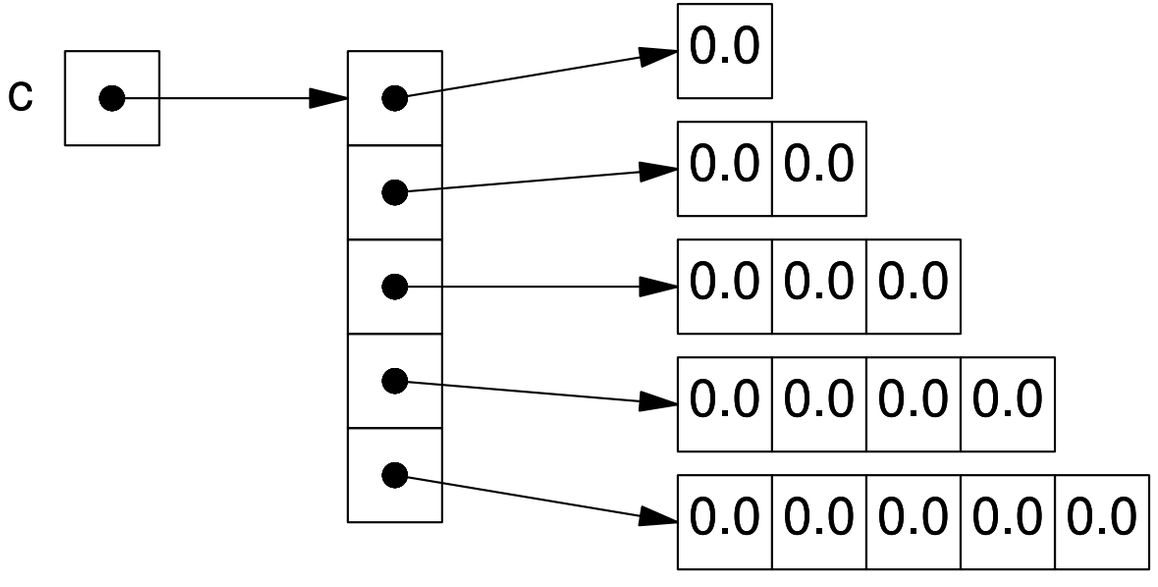
Hier ist `b` (eine Referenz auf) ein Array mit `m` Elementen (d. h. `b.length` liefert `m`); jedes dieser Elemente ist selbst wieder (eine Referenz auf) ein Array mit `n` Elementen des Typs `double` (d. h. `b[i].length` liefert jeweils `n`).



- Arrays von unterschiedlich großen Arrays kann man z. B. wie folgt erzeugen:

```
int n = ...;
double [] [] c = new double [n] [];
for (int i = 0; i < n; i++) {
  c[i] = new double [i+1];
}
```

Hier ist `c` (eine Referenz auf) ein Array mit `n` Elementen, die anfangs alle den Wert `null` besitzen; anschließend wird an jedes Element `c[i]` (eine Referenz auf) ein Array mit `i+1` Elementen des Typs `double` zugewiesen.



Array-Initialisierer

- Arrays können auch durch Array-Initialisierer erzeugt werden, z. B.:

```
double [] a = { 1.0, 2.0, 3.0 }; // double-Array mit 3 Elementen.

a = new double [] { 4.0, 5.0 };  // double-Array mit 2 Elementen.

double [] [] b = { { 1.0 }, { 2.0, 3.0 } };    // Zweidim. Array.
```

- Bei der Deklaration einer Array-Variablen kann der Array-Initialisierer `{ ..... }` direkt verwendet werden, in allgemeinen Ausdrücken muss er mit einem `new`-Ausdruck kombiniert werden.

- Um mehrdimensionale Arrays zu erzeugen und zu initialisieren, können Array-Initialisierer verschachtelt werden.

- Die Elemente eines Array-Initialisierers können beliebige Ausdrücke sein.
  Bei mehrdimensionalen Arrays können sie selbst wieder Arrays sein.

3.2.4 Zeichenketten (Strings)

- Zeichenkettenkonstanten wie z. B. `"abc"` besitzen in Java den vordefinierten Typ `String`.
- Die Länge eines `String`-Objekts `s` kann mit `s.length()` abgefragt werden. (Achtung: Die Länge eines Arrays `a` erhält man mittels `a.length`, ohne Klammern.)
- Für `i` zwischen 0 einschließlich und `s.length()` ausschließlich liefert `s.charAt(i)` das `i`-te Zeichen des Strings `s`. Für andere Werte von `i` erhält man eine Ausnahme des Typs `StringIndexOutOfBoundsException`.
- Wenn `s` die Nullreferenz `null` ist, produzieren die Ausdrücke `s.length()` und `s.charAt(i)` (sowie alle anderen Methodenaufrufe auf `s`) eine Ausnahme des Typs `NullPointerException`.
- Strings können mit dem Verkettungsoperator `+` verkettet werden.
- Wenn nur ein Operand des Verkettungsoperators ein String ist, wird der andere Operand automatisch in einen String umgewandelt.

- Beispiel:

```
int x = 2, y = 3;
System.out.println(
 "Die Summe von " + x + " und " + y + " ist " + (x + y) + ".");
```

- `s.equals(t)` überprüft, ob die Strings `s` und `t` (d. h. eigentlich die von `s` und `t` referenzierten `String`-Objekte) „gleich“ sind, d. h. die gleichen Zeichen in der gleichen Reihenfolge enthalten.

- Der Vergleich `s == t` überprüft lediglich, ob die Referenzen `s` und `t` gleich sind, d. h. ob `s` und `t` *dasselbe* `String`-Objekt referenzieren.

- Die Klasse `String` bietet zahlreiche weitere Operationen an, z. B. Suchen einzelner Zeichen oder Teilstrings in einem String, Bilden von Teilstrings usw.

- Anders als in C, sind Strings keine `char`-Arrays, und sie besitzen kein abschließendes Nullzeichen.

- Außerdem sind Strings *unveränderbar*, d. h. der Inhalt eines `String`-Objekts kann nach seiner Erzeugung nicht mehr geändert werden.
  Um einen String quasi zu „ändern“, muss man einen neuen String mit geändertem Inhalt erzeugen.

3.3 Operatoren

3.3.1 Übersicht

Operator	*Anwendung*	*Bedeutung*
`new`	präfix	Objekterzeugung
`.`*`name`*	postfix	Feldzugriff
`.`*`name`*`(`*`args`*`)`	postfix	Methodenaufruf
`[`*`index`*`]`	postfix	Array-Elementzugriff
`++`	postfix	Inkrement
`--`	postfix	Dekrement
`++`	präfix	Inkrement
`--`	präfix	Dekrement
`+`	präfix	unäres Plus (identische Funktion)
`-`	präfix	unäres Minus (arithmetische Negation)
`~`	präfix	bitweises Komplement
`!`	präfix	logische Negation
`(`*`type`*`)`	präfix	Cast (Typumwandlung)

Operator	*Anwendung*	*Bedeutung*
`*`	infix	Multiplikation
`/`	infix	Division
`%`	infix	Rest bei Division
`+`	infix	Addition oder String-Verkettung
`-`	infix	Subtraktion
`<<`	infix	Bitverschiebung nach links
`>>`	infix	arithmetische Bitverschiebung nach rechts
`>>>`	infix	logische Bitverschiebung nach rechts
`< > <= >=`	infix	numerische Vergleiche
`instanceof`	infix	dynamischer Typtest
`==`	infix	Test auf Gleichheit
`!=`	infix	Test auf Ungleichheit
`&`	infix	bitweise oder Boolesche Und-Verknüpfung
`^`	infix	bitweise oder Boolesche Exklusiv-Oder-Verknüpfung
`|`	infix	bitweise oder Boolesche (Inklusiv-)Oder-Verknüpfung
`&&`	infix	logische Und-Verknüpfung
`||`	infix	logische Oder-Verknüpfung
`? :`	infix	Verzweigung
`=`	infix	Zuweisung
`+= -= ...`	infix	kombinierte Zuweisung

3.3.2 Erläuterungen

- In den Tabellen besitzen alle Operatoren einer Gruppe denselben Vorrang, der höher ist als der Vorrang der nächsten Gruppe.

- Daher ist der Ausdruck `-a++ * b < c + d` beispielsweise äquivalent zu `((-(a++)) * b) < (c + d)`.

- Die Zuweisungsoperatoren und der Verzweigungsoperator sind rechtsassoziativ, alle anderen binären Operatoren linksassoziativ.

- Daher ist der Ausdruck `a - b - c` beispielsweise äquivalent zu `(a - b) - c` (und nicht zu `a - (b - c)`), während `a = b = 1` äquivalent zu `a = (b = 1)` ist.

3.3.3 Anmerkungen

- Anders als in C, wird der linke Operand eines Operators garantiert vor dem rechten ausgewertet.
  (Zum Beispiel liefert der Ausdruck `a[i++] - a[i++]` für `i = 0` garantiert den Wert von `a[0] - a[1]`; in C wäre auch `a[1] - a[0]` zulässig.)

- Ebenso werden die Argumente eines Methodenaufrufs garantiert von links nach rechts ausgewertet.
  (Zum Beispiel wird beim Ausdruck `f(i++, i++)` für `i = 0` garantiert der Methodenaufruf `f(0, 1)` ausgeführt; in C wäre auch `f(1, 0)` zulässig.)

- Die logischen Operatoren `!`, `&&` und `||` können nur auf `boolean`-Werte angewandt werden und liefern als Resultat wieder einen `boolean`-Wert (`true` oder `false`).
  (In C können sie auf Werte aller elementaren Typen angewandt werden und liefern als Resultat einen `int`-Wert, 1 oder 0.)

- Bei den Operatoren `&&` und `||` wird der rechte Operand (wie in C) nur ausgewertet, wenn dies zur Ermittlung des Ergebnisses notwendig ist.
  (Zum Beispiel produziert der Ausdruck `a != null && a.length > 0` für `a` gleich `null` keine `NullPointerException`, weil `a.length` in diesem Fall gar nicht ausgewertet wird.)

- Bei einer Verzweigung `x ? y : z` wird (wie in C), abhängig vom Wert des ersten Operanden (`x`), entweder nur der zweite (`y`) oder nur der dritte Operand (`z`) ausgewertet.
- Anders als in C, wird bei ganzzahliger Division garantiert „in Richtung 0“ gerundet.
- Für den Rest bei ganzzahliger Division gilt für alle möglichen Werte von `x` und `y`: `x % y == x - x/y * y`.
- Zum Beispiel:

```
 14 /  3  →   4        14 %  3  →  2
-14 /  3  →  -4       -14 %  3  → -2
 14 / -3  →  -4        14 % -3  →  2
-14 / -3  →   4       -14 % -3  → -2
```

- Für positives `y` erhält man mittels `(x % y + y) % y` für beliebige `x` den mathematisch korrekten Wert `x` mod `y`, der immer zwischen 0 und `y - 1` liegt.
- In C führt der Operator `>>` entweder eine logische oder eine arithmetische Verschiebung nach rechts durch (was nur bei vorzeichenlosen Werten gleichbedeutend ist); der Operator `>>>` existiert in C nicht.
- Der Komma-Operator, mit dem man in C die sequentielle Ausführung von Teilausdrücken beschreiben kann, existiert nicht.

3.3.4 Fehler und Ausnahmen

- Wenn `new` wegen Speicherplatzmangels kein Objekt erzeugen kann, erhält man einen Fehler des Typs `OutOfMemoryError`.
- Wenn man mit `new` ein Array mit negativer Elementzahl erzeugen will, erhält man eine Ausnahme des Typs `NegativeArraySizeException`.
- Ein Feldzugriff, ein Methodenaufruf oder ein Array-Elementzugriff über eine Nullreferenz produziert eine Ausnahme des Typs `NullPointerException`.
- Ein Array-Elementzugriff mit einem unzulässigen Index produziert eine Ausnahme des Typs `ArrayIndexOutOfBoundsException`.
- Eine unzulässige Typumwandlung produziert eine Ausnahme des Typs `ClassCastException` (sofern sie nicht schon vom Compiler als unzulässig erkannt wird; vgl. § 5.7).
- Eine ganzzahlige Division durch 0 (bzw. eine entsprechende Modulo-Operation) produziert eine Ausnahme des Typs `ArithmeticException`.
  Eine Gleitkommadivision wie z. B. `1.0/0.0` oder `-1.0/0.0` liefert jedoch einen unendlichen Wert mit entsprechendem Vorzeichen, während `0.0/0.0` den Sonderwert „not a number“ liefert.

3.4 Anweisungen

3.4.1 Atomare Anweisungen

- leere Anweisung (z. B. als Schleifenrumpf: `while ((x /= 2) > 1) ;`)
- Ausdrucks-Anweisung, bestehend aus einem Anweisungs-Ausdruck:
  - Zuweisung (z. B. `x = 1`)
  - Präfix- oder Postfix-Inkrement- oder -Dekrement-Ausdruck (z. B. `++x` oder `x--`)
  - Methodenaufruf (z. B. `System.out.println(5)`)
  - Objekterzeugung (z. B. `new int [10]` oder `new Account("Heinlein")`)

  In C kann eine Ausdrucks-Anweisung aus einem beliebigen Ausdruck bestehen.
- `break [label]`
  - zum vorzeitigen Beenden einer Anweisung
  - ohne Marke nur direkt oder indirekt in einer Schleife oder `switch`-Anweisung
  - mit Marke direkt oder indirekt in einer passend markierten beliebigen Anweisung

- `continue [label]`
  - zum vorzeitigen Fortsetzen einer Schleife
  - grundsätzlich nur direkt oder indirekt in einer Schleife
  - mit Marke nur direkt oder indirekt in einer passend markierten Schleife
- `return [expression]`
  - zum vorzeitigen Beenden einer Methode oder eines Konstruktors
  - und/oder zum Zurückgeben eines Resultatwerts
- `throw expression`
  - zum Werfen einer Ausnahme (vgl. Kapitel 8)
- Alle atomaren Anweisungen enden mit einem Semikolon.
- Im Gegensatz zu C gibt es keine `goto`-Anweisung!
  `break`- und `continue`-Anweisungen mit Marke (die es in C nicht gibt) erlauben nur Sprünge „von innen nach außen“ (d. h. keinen „Spaghetti-Code“).

3.4.2 Zusammengesetzte Anweisungen mit beliebigen Teilanweisungen

- markierte Anweisung (für `break` und `continue` mit Marke):
  - `label: statement`
- Verzweigung:
  - `if (condition) statement`
  - `if (condition) statement else statement`
  - Anders als in C, muss die Bedingung Typ `boolean` besitzen.
- Schleifen:
  - `while (condition) statement`
  - `do statement while (condition);`
  - `for (init; condition; update) statement`
  - Der Initialisierungs- und Aktualisierungsteil einer `for`-Schleife kann aus mehreren Anweisungs-Ausdrücken bestehen, die durch Komma getrennt sind.
  - Der Initialisierungsteil kann auch eine lokale Variablendeklaration sein. (Die Variablen sind dann nur innerhalb der Schleife gültig.)
  - Jeder Teil innerhalb der Klammern kann leer sein, wobei eine leere Bedingung immer erfüllt ist.

- Anweisungsblock:
  - `{ { declaration | statement } }`
  - Ein Anweisungsblock kann neben Anweisungen auch lokale Variablen- und Klassendeklarationen in beliebiger Reihenfolge enthalten.

Erläuterungen

- Schwarz geschriebene Zeichen oder Zeichenfolgen wie z. B. `for`, `{` und `}` sind Terminalsymbole, d. h. sie müssen „wörtlich“ verwendet werden.
- Blau geschriebene Namen wie z. B. `label` und `statement` sind Nicht-Terminalsymbole, d. h. Platzhalter.
- Rot geschriebene Zeichen wie z. B. `{` und `}` sind EBNF-Metazeichen (EBNF = Extended Backus-Naur Form):
  - `x | y` bedeutet: `x` oder `y`
  - `[ x ]` bedeutet: null- oder einmal `x`
  - `{ x }` bedeutet: beliebig viele `x` nacheinander (d. h. auch nullmal `x` ist zulässig)

3.4.3 Zusammengesetzte Anweisungen mit Blöcken als Teilanweisungen

- Fallunterscheidung:
  - `switch (expression) block`
  - Anders als in C, müssen `case`- und `default`-Marken direkt im abhängigen Anweisungsblock stehen.
  - Ohne `break`, `continue`, `return` oder `throw` am Ende eines Falls „fällt“ man direkt in den nächsten Fall.
  - Der Typ des kontrollierenden Ausdrucks muss `char`, `byte`, `short` oder `int` sein, die `case`-Marken müssen dazu passende konstante Werte (oder konstante Ausdrücke) sein.
- Synchronisation paralleler Threads:
  - `synchronized (expression) block`
- Auffangen von Ausnahmen und/oder Ausführen von Terminierungsanweisungen:
  - `try block { catch (type var) block } [ finally block ]`
  - Es muss mindestens ein `catch`-Block oder der `finally`-Block vorhanden sein.

3.4.4 Beispiel: markierte Anweisungen

```
// Überprüfe möglichst effizient, ob jede Zeile der Matrix a
// mindestens eine positive Zahl enthält.
double [] [] a = ...;
boolean result = true;

outer:                          // Äußere for-Schleife mit Marke »outer«.
for (int i = 0; i < a.length; i++) {  // Lokale Deklaration von i.
  for (int j = 0; j < a[i].length; j++) {   // Lokale Dekl. von j.
    if (a[i][j] > 0) {     // Positive Zahl in Zeile i gefunden.
      continue outer;      // Abbruch der inneren und
                           // Fortsetzung der äußeren for-Schleife.
    }
  }
  result = false;          // Keine pos. Zahl in Zeile i gefunden.
  break;                   // Abbruch der äußeren for-Schleife.
}
```

3.4.5 Beispiel: try/catch/finally

```
// Auffangen unterschiedlicher Ausnahmen.
int p = -1, q = 0, r = 1000*1000*1000;
double [] a = null;
while (true) {
  try {
    a = new double [p/q + r];
    a[0] = 1;
    break;
  }
  catch (ArithmeticException e) {
    System.out.println("Division durch 0");
    q = 1;
  }
  catch (NegativeArraySizeException e) {
    System.out.println("negative Arraygröße");
    p = -p;
  }
  catch (OutOfMemoryError e) {
    System.out.println("Speichermangel");
    r = -1;
  }
```

```
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Indexfehler");
      break;
    }
    finally {
      System.out.println("Ende der try-Anweisung");
    }
  }
```

- Der obige Code produziert folgende Ausgabe:

```
Division durch 0
Ende der try-Anweisung
Speichermangel
Ende der try-Anweisung
negative Arraygröße
Ende der try-Anweisung
Indexfehler
Ende der try-Anweisung
```

3.5 Überladen von Namen

- In C müssen alle Funktionen eines Programms (oder zumindest einer Quelldatei) eindeutige Namen besitzen.
- In Java können Methodennamen *überladen* werden, d. h. innerhalb einer Klasse kann es mehrere Methoden mit dem gleichen Namen geben, sofern sie unterschiedliche Parameterlisten besitzen. (Unterschiedliche Resultattypen genügen nicht.)
- Beim Aufruf einer überladenen Methode vergleicht der Compiler die Typen der aktuellen Aufrufparameter jeweils mit den Typen der formalen Methodenparameter und wählt die *am besten passende* Methode aus (d. h. die Reihenfolge der Methodendeklarationen spielt keine Rolle).
- Eine Methode passt exakt, wenn die Typen aller Aufrufparameter mit den Typen der entsprechenden Methodenparameter übereinstimmen.
- Eine Methode passt (mit Umwandlungen), wenn sich die Typen aller Aufrufparameter implizit in die Typen der entsprechenden Methodenparameter umwandeln lassen.
- Eine Methode passt besser als eine andere, wenn sich alle Parametertypen der ersten implizit in die entsprechenden Parametertypen der zweiten umwandeln lassen.
- Wenn es keine passende Methode gibt oder mehrere Methoden „gleich gut“ passen, erhält man eine entsprechende Fehlermeldung.

Beispiel 1

```
// Methodendefinitionen.
void print (boolean x) { ..... }        // 1
void print (int x) { ..... }            // 2
void print (double x) { ..... }         // 3

// b/i/d/s/l/c/S seien Ausdrücke mit Typ
// boolean/int/double/short/long/char/String.

// Welche der Methoden 1/2/3 passt exakt (++),
// passt mit Umwandlung (+), passt nicht (-)?
// Welche Methode wird ausgewählt?
                    //  Parametertyp     Methode          Auswahl
                    //                  1   2   3
print(b);           //  boolean         ++  -   -        1
print(i);           //  int             -   ++  +        2
print(d);           //  double          -   -   ++       3
print(s);           //  short           -   +   +        2
print(l);           //  long            -   -   +        3
print(c);           //  char            -   +   +        2
print(S);           //  String          -   -   -        keine
```

Beispiel 2

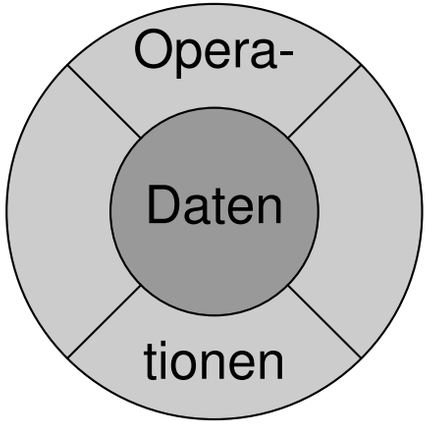
```
// Methodendefinitionen.
void test (long x, double y) { ..... }          // 1
void test (double x, long y) { ..... }          // 2
void test (int x, int y) { ..... }              // 3


// Welche der Methoden 1/2/3 passt exakt (++),
// passt mit Umwandlung (+), passt nicht (-)?
// Welche Methode wird ausgewählt?
                      //  Parametertypen   Methode       Auswahl
                      //                   1   2   3
test(i, i);           //  int, int         +   +   ++     3
test(i, d);           //  int, double      +   -   -      1
test(d, i);           //  double, int      -   +   -      2
test(d, d);           //  double, double   -   -   -      keine
test(c, c);           //  char, char       +   +   +      3
test(l, l);           //  long, long       +   +   -      mehrdeutig

// Auflösung der Mehrdeutigkeit durch Casts.
test(l, (double)l);   // long, double     ++  -   -      1
test((double)l, l);   // double, long     -   ++  -      2
```

4 Klassen in Java

4.1 Grundsätzliches

- Eine Klasse definiert einerseits eine Datenstruktur (in Form von *Feldern*) und andererseits die zugehörigen Operationen (in Form von *Methoden*).

- Häufig sind die Datenfelder *privat*, d. h. „von außen“ nicht direkt zugänglich, während die Methoden *öffentlich* sind, d. h. von „Klienten“ aufgerufen werden können (*Kapselung*, *Geheimnisprinzip*). Dadurch können Implementierungsdetails der Klasse geändert werden, ohne dass dies Auswirkungen auf Klientencode hat.



- Klassen sind *Referenztypen* (vgl. § 3.2.2), d. h. Variablen (und Ausdrücke), deren Typ eine Klasse ist, besitzen als Werte *Referenzen auf Objekte* dieser Klasse (oder die Nullreferenz `null`).

4.2 Aufbau einer Klasse

❒ Eine Klasse kann folgende Elemente in beliebiger Reihenfolge enthalten:

	statisch	nicht-statisch	
Felder	Klassenvariablen	Objektvariablen	§ 4.5
Methoden	Klassenmethoden	Objektmethoden	§ 4.6
Konstruktoren	–	+	§ 4.7
Initialisierungsblöcke	Klasseninitialisierer	Objektinitialisierer	§ 4.8
geschachtelte Klassen	+	innere Klassen	
geschachtelte Schnittstellen	+	–	

❒ Anders als in C, wo striktes „declare before use“ gilt, sind alle Elemente einer Klasse an jeder Stelle der Klasse sichtbar und (mit gewissen Einschränkungen bei Initialisierungen) verwendbar.

❒ Für die Lesbarkeit einer Klasse ist es aber trotzdem ratsam, Elemente erst nach ihrer Deklaration zu verwenden.

4.3 Zugriffskontrolle

- Elemente einer Klasse (mit Ausnahme von Initialisierungsblöcken) können mit folgenden Zugriffsbeschränkungen deklariert werden:
  - `private` (privat)
    Zugriff nur innerhalb der Klasse selbst erlaubt
  - *keine Angabe* (paketöffentlich)
    Zugriff im gesamten Paket erlaubt, zu dem die Klasse gehört (vgl. Kap. 9)
  - `protected` (unterklassenöffentlich)
    Zugriff im gesamten Paket sowie in allen Unterklassen der Klasse erlaubt (vgl. Kap. 5)
  - `public` (öffentlich)
    Zugriff überall erlaubt
- Eine Klasse selbst kann entweder öffentlich (Schlüsselwort `public`) oder paketöffentlich (keine Angabe) sein (vgl. Kap. 9).

4.4 Beispiel: Bankkonten (vgl. § 2.4)

```
// Klasse: Konto.
class Account {
  // Private Klassenvariable:
  // Nächste zu vergebende Kontonummer.
  private static int nextNumber = 1;

  // Private Objektvariablen:
  private final int number = nextNumber++;
                                  // Kontonummer (unveränderlich).
  private String holder;          // Kontoinhaber.
  private int balance = 0;        // Kontostand.

  // Öffentliche Konstruktoren: Konto mit Inhaber h, ggf.
  // Anfangsbetrag b und eindeutiger Nummer konstruieren.
  public Account (String h) {
    holder = h;
  }
  public Account (String h, int b) {
    this(h);                      // Den anderen Konstruktor aufrufen.
    balance = b;
  }
```

```
  // Öffentliche Objektmethoden:
  // Kontonummer/-inhaber/-stand abfragen.
  public int number () { return number; }
  public String holder () { return holder; }
  public int balance () { return balance; }

  // Öffentliche Objektmethoden:
  // Betrag amount einzahlen/abheben/überweisen.
  public void deposit (int amount) {
    balance += amount;
  }
  public void withdraw (int amount) {
    balance -= amount;
  }
  public void transfer (int amount, Account that) {
    withdraw(amount);
    that.deposit(amount);
  }

  // Öffentliche Klassenmethode:
  // Anzahl bereits erzeugter Konten abfragen.
  public static int numberOfAccounts () { return nextNumber - 1; }
}
```

4.5 Felder

4.5.1 Objektvariablen (nicht-statische Felder)

- Objektvariablen entsprechen Strukturkomponenten in C.
- Jedes Objekt einer Klasse besitzt eigene Ausprägungen der Objektvariablen der Klasse.
- Objektvariablen können Initialisierungsausdrücke besitzen, die bei jeder Erzeugung eines Objekts der Klasse ausgewertet werden (vgl. § 4.9).
- Nicht explizit initialisierte Objektvariablen erhalten bei der Erzeugung eines Objekts der Klasse einen typabhängigen Standardwert (vgl. § 3.2.3).
- Beispiele:

```
private final int number = nextNumber++;
private String holder;
private int balance = 0;
```

- Der Zugriff auf Objektvariablen erfolgt mit dem Punktoperator: `object.objvar`
  Dabei kann `object` ein beliebiger (ggf. geklammerter) Ausdruck sein, dessen Resultat ein Objekt der Klasse darstellt.

- Beispiele:

```
a.number
(a = new Account("Heinlein")).balance
```

- Die Objektvariablen des aktuellen Objekts (vgl. § 4.6) können auch direkt über ihren Namen angesprochen werden, d. h. `objvar` ist äquivalent zu `this.objvar`.

4.5.2 Klassenvariablen (statische Felder)

- Klassenvariablen entsprechen globalen Variablen in C, deren Zugriff jedoch beschränkt werden kann (vgl. § 4.3).

- Alle Objekte einer Klasse „teilen sich“ die Klassenvariablen der Klasse, d. h. verwenden sie gemeinsam.

- Klassenvariablen können ebenfalls Initialisierungsausdrücke besitzen, die einmalig beim Laden/Initialisieren der Klasse ausgewertet werden.

- Nicht explizit initialisierte Klassenvariablen besitzen analog zu Objektvariablen typabhängige Standardwerte.

- Beispiel:

```
private static int nextNumber = 1;
```

- Der Zugriff auf Klassenvariablen erfolgt normalerweise über den Klassennamen: `classname.classvar`

- Die Klassenvariablen der aktuellen Klasse können auch direkt über ihren Namen angesprochen werden.

- Prinzipiell können Klassenvariablen auch über irgendein Objekt der Klasse angesprochen werden: `object.classvar`
  Das ist allerdings nicht üblich und wird von manchen Compilern mit einer Warnung quittiert. Das Objekt `object` wird hierbei gar nicht verwendet.

- Beispiele:

```
Account.nextNumber      // Zugriff über Klassenname
nextNumber              // Direkter Zugriff über Variablenname
a.nextNumber            // Zugriff über ein Objekt der Klasse
```

4.5.3 Unveränderliche Felder

- Objekt- bzw. Klassenvariablen, die `final` deklariert sind, müssen während der Initialisierung eines Objekts bzw. der Klasse (und vor ihrer ersten Verwendung) initialisiert werden und dürfen anschließend nicht mehr verändert werden.
  (Die Initialisierung eines Objekts besteht aus den in § 4.9 beschriebenen Schritten 4 und 5.)

- Beispiel:

```
private final int number = nextNumber++;
```

  (Alternativ kann die Initialisierung `number = nextNumber++` in einem Konstruktor oder Objektinitialisierer der Klasse erfolgen.)

- Unveränderliche Klassenvariablen, deren Wert bereits vom Compiler bestimmt werden kann, belegen zur Laufzeit des Programms keinen Speicherplatz und werden an jeder Verwendungsstelle direkt durch ihren Wert ersetzt (ähnlich wie Makros in C).

- Beispiel:

```
public static final int FIRST_NUMBER = 1;
```

  Entspricht in C:

```
#define FIRST_NUMBER 1
```

4.6 Methoden

4.6.1 Objektmethoden (nicht-statische Methoden)

- Objektmethoden entsprechen Funktionen in C, die einen zusätzlichen impliziten Parameter `this` besitzen, dessen Typ die umgebende Klasse ist und der das *aktuelle Objekt* der Klasse bezeichnet.
- Beispiele:

```
public int number () { return this.number; }

public void transfer (int amount, Account that) {
  withdraw(amount);
  that.deposit(amount);
}
```

- Objektmethoden werden – analog zu Objektvariablen – immer für ein bestimmtes Objekt aufgerufen: `object.objmeth([arguments])`
- Innerhalb einer Objektmethode steht das Aufrufobjekt `object` dann als aktuelles Objekt `this` zur Verfügung.
- Die Objektmethoden des aktuellen Objekts können auch direkt über ihren Namen angesprochen werden, d. h. `objmeth` ist äquivalent zu `this.objmeth`.

4.6.2 Klassenmethoden (statische Methoden)

- Klassenmethoden entsprechen globalen Funktionen in C, deren Zugriff jedoch beschränkt werden kann (vgl. § 4.3).
- Beispiel:

```
public static int numberOfAccounts () { return nextNumber - 1; }
```

- Klassenmethoden werden normalerweise über den Klassennamen aufgerufen:
  `classname.classmeth([arguments])`
- Die Klassenmethoden der aktuellen Klasse können auch direkt über ihren Namen angesprochen und aufgerufen werden.
- Prinzipiell können Klassenmethoden auch über irgendein Objekt der Klasse aufgerufen werden: `object.classmeth([arguments])`
  Das ist allerdings, ebenso wie bei Klassenvariablen, nicht üblich und wird von manchen Compilern ebenfalls mit einer Warnung quittiert. Auch hier wird das Objekt `object` gar nicht verwendet.

- Beispiele:

```
Account.numberOfAccounts()      // Aufruf über Klasssenname
a.numberOfAccounts()            // Aufruf über Objekt der Klasse
```

- Innerhalb von Klassenmethoden gibt es kein aktuelles Objekt `this` (selbst wenn eine Klassenmethode formal über ein Objekt der Klasse aufgerufen wird), d. h. die explizite oder implizite Verwendung von `this` in Klassenmethoden führt zu einem Fehler.

4.6.3 Hauptmethode `main`

- Wenn eine Klasse eine öffentliche statische Methode mit dem Namen `main`, einem einzelnen Parameter des Typs `String []` und Resultattyp `void` besitzt, kann diese Klasse als Programm ausgeführt werden.

- Beim Aufruf des Programms wird die Klasse zunächst geladen und initialisiert (vgl. § 4.8) und anschließend die o. g. Methode `main` aufgerufen.

- Die beim Aufruf des Programms eventuell angegebenen Kommandoargumente sind über den Parameter der o. g. Methode `main` verfügbar.

- Der Name `main` kann wie jeder andere Methodenname verwendet und ggf. auch überladen werden, d. h. eine Klasse kann auch Methoden mit dem Namen `main` und anderen Signaturen besitzen.

4.6.4 Resultatwert von Methoden

- Eine Methode, deren Resultattyp nicht `void` ist, muss unter allen Umständen mit `return` einen passenden Resultatwert zurückliefern (oder eine Ausnahme werfen).
- Das heißt, eine solche Methode darf unter keinen Umständen das normale Ende ihres Rumpfs erreichen.
- Wenn der Compiler dies (mit seinem begrenzten Wissen) nicht zweifelsfrei beweisen kann, signalisiert er einen Fehler.
- Beispiel:

```
public static String signAsString (double x) {
  // Math.signum liefert -1.0/0.0/+1.0, wenn x>0/x==0/x<0 ist.
  switch ((int)Math.signum(x)) {
  case -1: return "-";
  case  0: return "";
  case +1: return "+";
  }
}
```

Da der kontrollierende Ausdruck der `switch`-Anweisung aus Sicht des Compilers nicht nur die Werte `-1`, `0` und `+1` besitzen kann, ist die Methode fehlerhaft, weil sie nur in diesen drei Fällen eine `return`-Anweisung ausführt.

4.7 Konstruktoren

- Konstruktoren sind spezielle Objektmethoden, die zur Initialisierung neuer Objekte aufgerufen werden (vgl. § 4.9).
- Der Name eines Konstruktors muss mit dem Namen der Klasse übereinstimmen, zu der er gehört.
- Konstruktoren besitzen keinen Resultattyp (nicht einmal `void`).
- Innerhalb eines Konstruktors kann das zu initialisierende Objekt wie in einer gewöhnlichen Objektmethode über das Schlüsselwort `this` angesprochen werden. (Das heißt, das Objekt ist bereits erzeugt und muss nur noch initialisiert werden.)
- Konstruktoren können – wie andere Methoden – überladen werden.
- Die erste Anweisung eines Konstruktorrumpfs kann ein expliziter Aufruf eines anderen Konstruktors der Klasse sein, wobei hier als „Methodenname“ nicht der Klassenname, sondern das Schlüsselwort `this` verwendet wird:
  `this([arguments])`
  In der Argumentliste eines solchen Konstruktoraufrufs darf das aktuelle Objekt `this` weder direkt noch indirekt verwendet werden, da es noch nicht initialisiert ist.

- Ansonsten können Konstruktoren nur im Zusammenhang mit Objekterzeugungsausdrücken aufgerufen werden (vgl. § 4.9).

- Beispiele:

```
public Account (String h) {
  holder = h;
}

public Account (String h, int b) {
  this(h);
  balance = b;
}
```

- Ein Konstruktorrumpf kann `return`-Anweisungen ohne Ausdruck enthalten, um die Ausführung vorzeitig zu beenden.

- Wenn eine Klasse keinen expliziten Konstruktor enthält, besitzt sie implizit einen öffentlichen parameterlosen Konstruktor mit leerem Rumpf.

- Um dies zu verhindern, kann man einen privaten parameterlosen Dummy-Konstruktor definieren.

4.8 Initialisierungsblöcke

4.8.1 Objektinitialisierer (nicht-statische Initialisierungsblöcke)

- Objektinitialisierer sind Anweisungsblöcke innerhalb einer Klasse, die bei jeder Initialisierung eines neuen Objekts der Klasse ausgeführt werden (vgl. § 4.9).

- Sie sind nützlich, um Initialisierungscode zu formulieren, der von allen Konstruktoren einer Klasse ausgeführt werden soll, nicht von Konstruktorparametern abhängt und nicht (vernünftig) mit Initialisierungsausdrücken von Objektvariablen ausgedrückt werden kann.

- Wie in einem Konstruktor, kann das zu initialisierende Objekt über das Schlüsselwort `this` angesprochen werden.

- Beispiel:

```
{
  System.out.println("Initialisierung eines neuen Kontos");
}
```

4.8.2 Klasseninitialisierer (statische Initialisierungsblöcke)

- Klasseninitialisierer sind Anweisungsblöcke innerhalb einer Klasse, die einmalig beim Laden/Initialisieren der Klasse ausgeführt werden.

- Sie sind nützlich, um Initialisierungscode zu formulieren, der nicht (vernünftig) mit Initialisierungsausdrücken von Klassenvariablen ausgedrückt werden kann.

- Beim Laden/Initialisieren einer Klasse werden die Klasseninitialisierer und die Initialisierungsausdrücke von Klassenvariablen der Reihe nach ausgeführt bzw. ausgewertet.

- Beispiel:

```
static {
  System.out.println("Initialisierung der Klasse Account");
}
```

4.9 Erzeugung und Initialisierung von Objekten einer Klasse

- Objekte einer Klasse werden durch Ausführung eines Objekterzeugungsausdrucks erzeugt, der aus dem Schlüsselwort `new`, dem Namen der Klasse und einer (eventuell leeren) Konstruktorargumentliste besteht: `new classname ([arguments])`

- Zum Beispiel:

```
Account a = new Account("Heinlein");
a = new Account("Heinlein", 1000);
```

- Ein Objekterzeugungsausdruck wird wie folgt ausgewertet:

1. Es wird Platz für das neue Objekt beschafft.
   Falls dies nicht möglich ist, wird eine Ausnahme des Typs `OutOfMemoryError` ausgelöst.
2. Alle Objektvariablen des Objekts werden mit typabhängigen Standardwerten vorinitialisiert (vgl. § 3.2.3).
3. Die Konstruktorargumente werden (von links nach rechts) ausgewertet.
4. Die Objektinitialisierer der Klasse und die Initialisierungsausdrücke von Objektvariablen werden der Reihe nach ausgeführt bzw. ausgewertet.
5. Der passende Konstruktor wird ausgeführt.
   - Wenn er als erstes via `this` einen anderen Konstruktor aufruft, wird dieser ausgeführt, nachdem seine Argumente (von links nach rechts) ausgewertet wurden.
   - Andernfalls bzw. anschließend wird der (verbleibende) Konstruktorrumpf ausgeführt.
6. Als Resultat des Objekterzeugungsausdrucks wird eine Referenz auf das neue Objekt geliefert.

4.10 Freigabe von Objekten

- Mit `new` erzeugte (Klassen- und Array-) Objekte können und müssen nicht explizit freigegeben werden.

- Objekte, die vom Programm nicht mehr benötigt werden, weil sie nicht mehr *erreichbar* sind, werden von Zeit zu Zeit automatisch freigegeben (automatische Speicherbereinigung, garbage collection).

- Ein Objekt ist *erreichbar*, wenn es direkt oder indirekt erreichbar ist.

- Ein Objekt ist *direkt erreichbar*, wenn es von einer Klassenvariablen, einer lokalen Variablen (vgl. § 4.11) oder einem Parameter eines Konstruktors, einer Methode oder eines `catch`-Blocks (vgl. § 8.6.2) referenziert wird.

- Ein Objekt ist *indirekt erreichbar*, wenn es von einer Objektvariablen eines erreichbaren Klassenobjekts oder einem Element eines erreichbaren Arrayobjekts referenziert wird.

4.11 Lokale Variablen und Parameter

- Lokale Variablen sind Variablen, die in einem Anweisungsblock oder im Initialisierungsteil einer `for`-Schleife deklariert werden (vgl. § 3.4.2), d. h. direkt oder indirekt in einer Methode, einem Konstruktor oder einem Initialisierungsblock.

- Eine lokale Variable oder ein Parameter *verdeckt* (engl. *shadows*) eine gleichnamige Objekt- oder Klassenvariable der umgebenden Klasse (die ggf. mittels `this.objvar` bzw. `classname.classvar` angesprochen werden kann).

- Eine lokale Variable (oder ein Parameter eines `catch`-Blocks; vgl. § 8.6.2) darf aber keine weiter außen deklarierten lokalen Variablen oder Parameter verdecken.

- Anders als Objekt- und Klassenvariablen (vgl. § 4.5.1 und § 4.5.2), werden lokale Variablen *nicht* mit Standardwerten vorinitialisiert, sondern müssen unter allen Umständen vor ihrer ersten Verwendung einen Wert erhalten.

- Wenn der Compiler dies (mit seinem begrenzten Wissen) nicht zweifelsfrei beweisen kann, signalisiert er einen Fehler (den man z. B. durch explizite Initialisierung mit einem Dummywert vermeiden kann).

4.12 Namenskonventionen

- In der „Java Community“ gelten die folgenden Namenskonventionen, d. h. Regeln, die üblicherweise befolgt werden, obwohl ihre Einhaltung vom Compiler nicht überprüft wird:
  - Namen von Klassen und Schnittstellen (vgl. § 6.2) beginnen mit einem Großbuchstaben (z. B. `Account`), alle anderen Namen mit einem Kleinbuchstaben (z. B. `number`, `deposit`, `i`).
  - Bei Namen, die aus mehreren Wörtern zusammengesetzt sind, beginnen das zweite und alle folgenden Wörter mit einem Großbuchstaben (z. B. `LimitedAccount`, `nextNumber`).
- Für Konstanten, d. h. unveränderliche Klassenvariablen, gelten abweichend folgende Regeln:
  - Die Namen bestehen komplett aus Großbuchstaben (z. B. `PI`).
  - Bei zusammengesetzten Namen werden die einzelnen Wörter durch Unterstriche getrennt (z. B. `FIRST_NUMBER`).

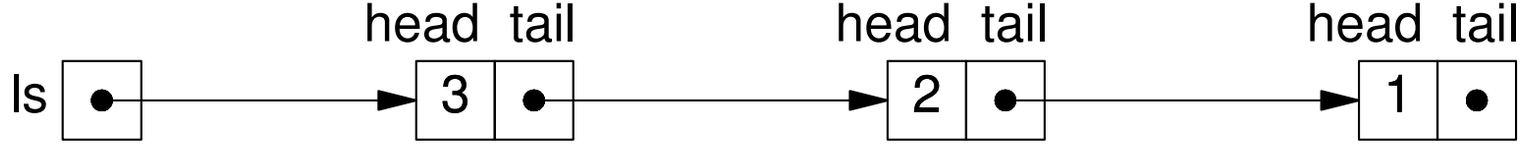
4.13 Beispiel: Lineare Liste

```
class List {
  // Unterklassenöffentliche Objektvariablen:
  // (protected statt private, damit die Variablen später
  // in Unterklassen verwendet werden können.)
  protected int head;   // Erstes Element (Kopf).
  protected List tail;  // Restliste (Schwanz).

  // Öffentliche Konstruktoren:
  // Liste mit erstem Element h und ggf. Restliste t konstruieren.
  public List (int h, List t) {
    head = h;
    tail = t;
  }
  public List (int h) {
    this(h, null);
  }

  // Öffentliche Objektmethoden:
  // Erstes Element und Restliste abfragen.
  public int head () { return head; }
  public List tail () { return tail; }
```

```
  // Öffentliche Objektmethode:
  // Länge (d. h. Anzahl der Elemente) ermitteln.
  public int length () {
    int n = 1;
    for (List p = tail; p != null; p = p.tail) n++;
    return n;
  }

  // Öffentliche Objektmethode: Liste ausgeben.
  public void print () {
    for (List p = this; p != null; p = p.tail) {
      System.out.println(p.head);
    }
  }
}
```

```
class Test {
  // Testprogramm.
  public static void main (String [] args) {
    // Liste ls erzeugen, die als Elemente die Werte der
    // Kommandoargumente in umgekehrter Reihenfolge enthält.
    List ls = null;
    for (int i = 0; i < args.length; i++) {
      ls = new List(Integer.parseInt(args[i]), ls);
    }

    // Liste ausgeben.
    System.out.println(ls.length());
    ls.print();
  }
}
```

- Beispielaufruf: `java Test 1 2 3`

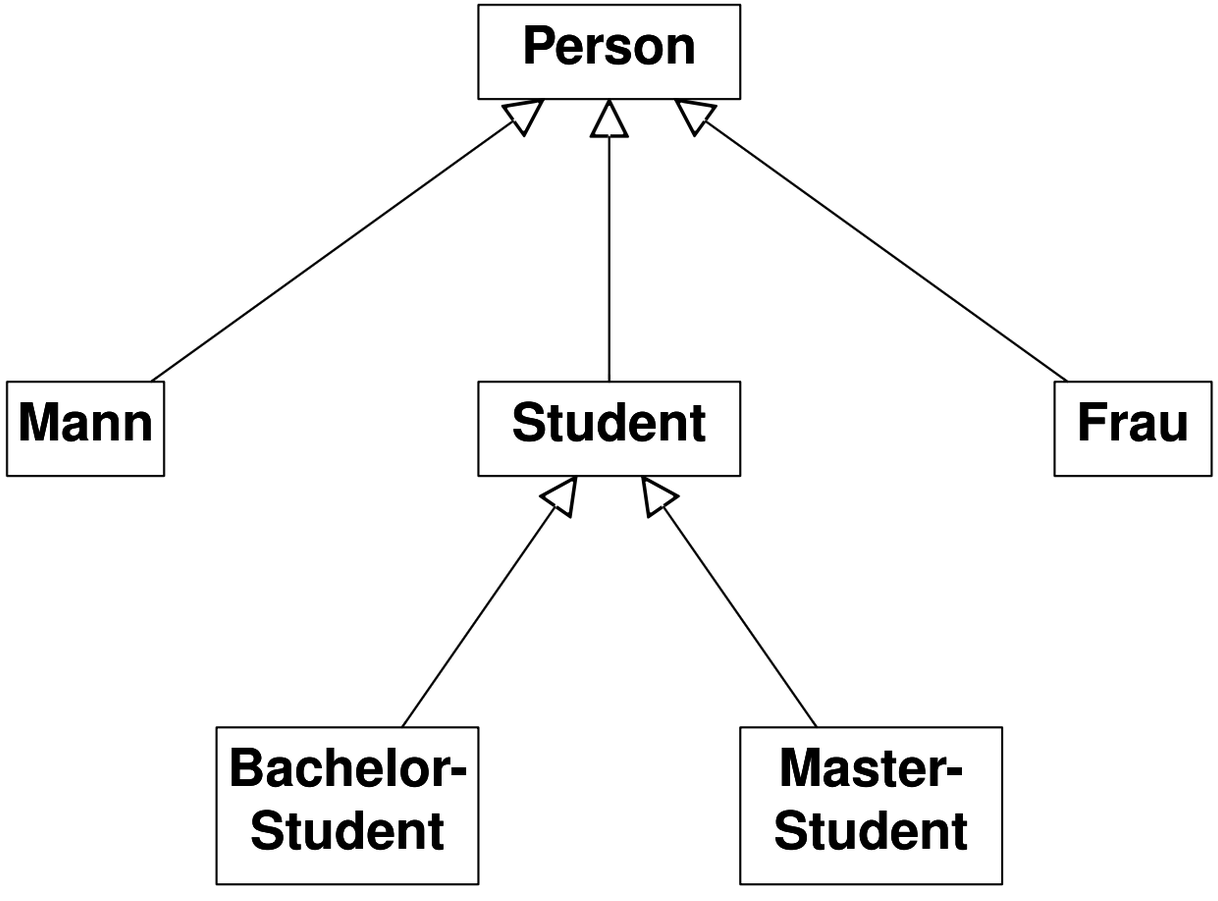


5 Unterklassen

5.1 Begriffe

- Ein Klasse kann eine andere Klasse *erweitern* (Schlüsselwort `extends`), d. h. ihre Felder und Methoden (sowie ihre geschachtelten Klassen und Schnittstellen, nicht jedoch ihre Konstruktoren und Initialisierungsblöcke) *erben*.
- Die erbende Klasse heißt (direkte) *Unterklasse*, die beerbte Klasse (direkte) *Oberklasse*.
- Eine Klasse kann beliebig viele direkte Unterklassen, aber nur eine direkte Oberklasse besitzen (*einfache Vererbung*).
- Eine Unterklasse einer Unterklasse wird auch als *indirekte Unterklasse*, eine Oberklasse einer Oberklasse als *indirekte Oberklasse* bezeichnet.
- Gelegentlich wird eine Klasse auch als *triviale Unter-* und *Oberklasse* ihrer selbst bezeichnet.
- Formal besteht die Menge aller Unter- bzw. Oberklassen einer Klasse daher aus:
  - der Klasse selbst
  - der Menge aller Unter- bzw. Oberklassen ihrer direkten Unter- bzw. Oberklasse(n).

Beispiel



5.2 Zugriffskontrolle

- Eine Unterklasse kann auf folgende Elemente ihrer (direkten und indirekten) Oberklasse(n) zugreifen (vgl. § 4.3):
  - öffentliche Elemente (`public`)
  - unterklassenöffentliche Elemente (`protected`)
  - paketöffentliche Elemente (keine Angabe), wenn sich die Unterklasse im selben Paket wie die Oberklasse befindet (vgl. Kap. 9)
- Auf private Elemente (`private`) einer Oberklasse kann grundsätzlich nicht zugegriffen werden.

5.3 Untertyp-Polymorphie

- Da ein Objekt einer Unterklasse alle Merkmale seiner Oberklasse besitzt, kann es überall verwendet werden, wo ein Objekt der Oberklasse erwartet wird (*Ersetzbarkeit*). Insbesondere kann ein Objekt der Unterklasse an eine Variable der Oberklasse zugewiesen werden (aber nicht umgekehrt).

- Somit können Variablen eines bestimmten Klassentyps zur Laufzeit nicht nur Objekte dieser Klasse, sondern auch Objekte von (direkten und indirekten) Unterklassen referenzieren (*Untertyp-Polymorphie*).

- Daher muss zwischen dem *statischen Typ* einer Variablen (oder eines Ausdrucks) und ihrem *dynamischen Typ* unterschieden werden:

  - Der statische Typ ist der im Programmcode deklarierte Typ einer Variablen (bzw. der aus dem Programmcode ableitbare Typ eines Ausdrucks).
    Er ist dem Compiler bekannt und während der gesamten Programmausführung konstant.

  - Der dynamische Typ einer Variablen (oder eines Ausdrucks) ist der Typ des tatsächlich referenzierten Objekts.
    Er ist dem Compiler nicht bekannt und kann sich im Laufe der Programmausführung ändern.
    Es handelt sich jedoch immer um einen Untertyp des statischen Typs.

Beispiel

```
class Person { ..... }
class Student extends Person { ..... }

class Test {
  public static void main (String [] args) {
    // Der statische und dynamische Typ von p ist Person.
    Person p = new Person();

    // Der statische und dynamische Typ von s ist Student.
    Student s = new Student();

    // Das von s referenzierte Student-Objekt
    // kann an die Person-Variable p zugewiesen werden.
    // Der statische Typ von p bleibt Person,
    // der dynamische Typ wird Student.
    p = s;

    // Die umgekehrte Zuweisung ist nicht möglich.
    s = p;      // Fehler!!
  }
}
```

5.4 Überschreiben und dynamisches Binden von Methoden

- Eine Unterklasse kann von einer Oberklasse geerbte Objektmethoden *überschreiben*, d. h. neu implementieren.

- Beim Aufruf einer Objektmethode wird *zur Laufzeit* anhand des dynamischen Typs des Aufrufobjekts (d. h. unabhängig von seinem statischen Typ) die passende Implementierung der Methode ausgewählt (*dynamisches Binden*).

- Demgegenüber werden Klassenmethoden *statisch gebunden*, d. h. es steht bereits zur *Übersetzungszeit* fest, welche Methode zur Laufzeit ausgeführt wird.

- *Überschreiben* und *Überladen* von Methoden (vgl. § 3.5) sind voneinander unabhängige Konzepte, die bei Bedarf auch kombiniert werden können.
  Die Auswahl der ausgeführten Methode erfolgt dann in mehreren Schritten:
  - Der statische Typ des Aufrufobjekts bestimmt die Klasse, in der der Compiler nach dem Methodennamen sucht.
  - Die statischen Typen der Aufrufparameter bestimmen (zur Übersetzungszeit) die am besten passende Methode.
  - Wenn diese Methode in Unterklassen überschrieben ist, bestimmt der dynamische Typ des Aufrufobjekts zur Laufzeit die passende Implementierung dieser Methode.

5.5 Konstruktoren

- Konstruktoren werden grundsätzlich nicht vererbt, d. h. jede Klasse definiert ihre eigenen Konstruktoren.

- Ein Konstruktor kann als erste Anweisung einen Aufruf eines Konstruktors der Oberklasse enthalten, um die Objektvariablen der Oberklasse zu initialisieren. Als Name dieses Konstruktors wird nicht der Name der Oberklasse, sondern das Schlüsselwort `super` verwendet.

- Wenn die erste Anweisung eines Konstruktors weder ein Aufruf eines anderen Konstruktors der eigenen Klasse (vgl. § 4.7) noch ein Aufruf eines Konstruktors der Oberklasse ist, wird automatisch ein Aufruf `super()` des parameterlosen Konstruktors der Oberklasse eingefügt.

- Dies impliziert, dass die Oberklasse in diesem Fall einen (entsprechend zugreifbaren) parameterlosen Konstruktor besitzen muss.
  (Wenn eine Klasse keine explizite Oberklasse besitzt, besitzt sie implizit die Wurzelklasse `Object` als Oberklasse, die einen solchen Konstruktor besitzt; vgl. § 5.10.)

- Der explizite oder implizite Aufruf des Oberklassenkonstruktors findet zwischen den in § 4.9 beschriebenen Schritten 3 und 4 statt (d. h. nach der Auswertung der eigenen Konstruktorargumente) und führt (nach der Auswertung seiner Argumente) für die Oberklasse die dort genannten Schritte 4 und 5 aus, d. h. die Initialisierungen der Oberklasse finden vor den eigenen Initialisierungen statt.

- Wenn die Oberklasse selbst wieder eine Oberklasse besitzt (d. h. wenn sie verschieden von der Wurzelklasse `Object` ist), wird zuvor der passende `super`-Aufruf ihres Konstruktors ausgeführt, usw.
  Somit werden die Initialisierungsschritte 4 und 5 für alle Oberklassen der Klasse „von oben nach unten“ ausgeführt.

5.6 Beispiel: Limitierte Konten (vgl. § 2.2)

```
// Unterklasse von Account: Limitiertes Konto.
class LimitedAccount extends Account {
  // Zusätzliche Objektvariable:
  private int limit;              // Kreditlinie in Cent.

  // Konstruktoren:
  // Limitiertes Konto mit Inhaber h, ggf. Anfangsbetrag b,
  // Kreditlinie l und eindeutiger Nummer konstruieren.
  public LimitedAccount (String h, int b, int l) {
    super(h, b); // Konstruktor der Oberklasse Account aufrufen,
                 // um deren Objektvariablen zu initialisieren.
    limit = l;   // Zusätzliche Objektvariable limit initialisieren.
  }
  public LimitedAccount (String h, int l) {
    // Entweder:                  // Oder:
    super(h);                     // this(h, 0, l);
    limit = l;                    //
  }

  // Zusätzliche Objektmethode: Kreditlinie abfragen.
  public int limit () { return limit; }
```

```
  // Hilfsmethode: Kann Betrag amount abgezogen werden,
  // ohne die Kreditlinie zu überschreiten?
  private boolean check (int amount) {
    if (balance() - amount >= -limit) return true;
    System.out.println("Unzulässige Kontoüberziehung!");
    return false;
  }

  // Überschreiben geerbter Objektmethoden:
  // Betrag amount abheben/überweisen.
  public void withdraw (int amount) {
    if (check(amount)) {
      // Überschriebene Methode aufrufen.
      super.withdraw(amount);
    }
  }
  public void transfer (int amount, Account that) {
    if (check(amount)) {
      // Überschriebene Methode aufrufen.
      super.transfer(amount, that);
    }
  }
}
```

```
// Testklasse.
class Test {
  public static void main (String [] args) {
    // Objekte erzeugen und durch Konstruktoraufrufe initialisieren.
    Account a = new LimitedAccount("Hans Maier", 500);
    Account b = new Account("Fritz Müller");

    // Da deposit in LimitedAccount nicht überschrieben ist,
    // wird die Account-Implementierung der Methode ausgeführt.
    a.deposit(1000);

    // Da a den dynamischen Typ LimitedAccount besitzt,
    // wird die LimitedAccount-Implementierung von transfer
    // und withdraw ausgeführt (was bei withdraw zu einer
    // unzulässigen Kontoüberziehung führt).
    a.transfer(300, b);
    a.withdraw(2000);

    // Da b den dynamischen Typ Account besitzt,
    // wird die Account-Implementierung von withdraw ausgeführt.
    b.withdraw(2000);
  }
}
```

5.7 Dynamische Typtests und statische Typumwandlungen

- Der Ausdruck `object instanceof type` liefert genau dann `true`, wenn der dynamische Typ des Objekts `object` ein (trivialer, direkter oder indirekter) Untertyp von `type` ist (woraus folgt, dass `object` keine Nullreferenz ist).

- Wenn die Bedingung `object instanceof type` erfüllt ist, liefert der Ausdruck `(type)object` das Objekt `object` mit statischem Typ `type` (und unverändertem dynamischen Typ); wenn die Bedingung nicht erfüllt ist, erhält man eine Ausnahme des Typs `ClassCastException`.

- Wenn `type` ein Obertyp des statischen Typs von `object` ist, bezeichnet man die Typumwandlung `(type)object` als *Aufwärtsumwandlung* (up-cast).
  Da `object` in diesem Fall sowieso überall verwendet werden kann, wo ein Objekt des Typs `type` erwartet wird (vgl. § 5.3), ist eine solche Umwandlung normalerweise nicht erforderlich; sie kann jedoch notwendig sein, um den Aufruf einer überladenen Methode eindeutig zu machen (vgl. § 3.5) oder um auf ein verborgenes Feld zuzugreifen (vgl. § 5.8).

- Wenn `type` ein Untertyp des statischen Typs von `object` ist, bezeichnet man die Typumwandlung `(type)object` als *Abwärtsumwandlung* (down-cast).

- Wenn `type` weder ein Ober- noch ein Untertyp des statischen Typs von `object` ist, wird die Umwandlung vom Compiler zurückgewiesen.

5.8 Verbergen von Feldern

- Wenn ein Feld (Objekt- oder Klassenvariable) einer Klasse denselben Namen wie ein geerbtes Feld besitzt, ist das geerbte Feld in dieser Klasse zwar vorhanden, aber *verborgen* (engl. *hidden*), d. h. nicht direkt über seinen Namen ansprechbar.

- Ein verborgenes Feld `field` des aktuellen Objekts kann über `super.field` angesprochen werden.

- Ein verborgenes Feld eines beliebigen Objekts `object` kann durch eine explizite Typumwandlung in die passende Oberklasse angesprochen werden:
  `((superclass)object).field`

- Allgemein: Feldzugriffe werden *statisch gebunden*, d. h. bei einem Feldzugriff `object.field` bestimmt allein der *statische* Typ von `object`, welches Feld ausgewählt wird.

5.9 Überschreiben und Verbergen von Methoden

5.9.1 Überschreiben von Objektmethoden

- Wenn eine Objektmethode einer Klasse dieselbe Signatur (d. h. denselben Namen und dieselben Parametertypen) wie eine geerbte (nicht-private) Objektmethode besitzt, wird die geerbte Methode *überschrieben* (vgl. § 5.4).

- Für das aktuelle Objekt `this` kann eine überschriebene Methode `objmeth` (nur) mittels `super.objmeth(arguments)` aufgerufen werden.
  (Das heißt, die Verwendung von `super` setzt das dynamische Binden außer Kraft und ruft direkt die Methode der Oberklasse auf.)

- Ein Aufruf der Art `((superclass)object).objmeth(arguments)` ist aufgrund des normalen dynamischen Bindens äquivalent zu `object.objmeth(arguments)`, da sich der *dynamische* Typ von `object` durch die Umwandlung des *statischen* Typs nach `superclass` nicht ändert.

5.9.2 Verbergen von Klassenmethoden

- Wenn eine Klassenmethode einer Klasse dieselbe Signatur wie eine geerbte (nicht-private) Klassenmethode besitzt, wird die geerbte Methode *verborgen* (engl. *hidden*).
- Eine verborgene Methode `classmeth` kann wie folgt aufgerufen werden:
  - `superclass.classmeth(arguments)`
  - `super.classmeth(arguments)` (innerhalb einer Objektmethode)
  - `((superclass)object).classmeth(arguments))` (unüblich)

5.9.3 Allgemeine Regeln

- Eine Objektmethode kann keine geerbte Klassenmethode überschreiben, und eine Klassenmethode kann keine geerbte Objektmethode verbergen.
- Der Resultattyp der überschreibenden bzw. verbergenden Methode muss mit dem Resultattyp der überschriebenen bzw. verborgenen Methode übereinstimmen oder ein Untertyp davon sein.
- Die überschreibende bzw. verbergende Methode muss mindestens so viel Zugriff erlauben wie die überschriebene bzw. verborgene Methode, d. h.:
  - Eine öffentliche Methode kann nur von einer öffentlichen Methode überschrieben bzw. verborgen werden.
  - Eine unterklassenöffentliche Methode kann nur von einer öffentlichen oder unterklassenöffentlichen Methode überschrieben bzw. verborgen werden.
  - Eine paketöffentliche Methode kann nur von einer öffentlichen, unterklassen-öffentlichen oder paketöffentlichen Methode überschrieben bzw. verborgen werden.
- Private Methoden werden grundsätzlich nicht überschrieben oder verborgen, d. h. wenn eine Methode einer Klasse dieselbe Signatur wie eine private Methode einer Oberklasse besitzt, stehen diese Methoden in keinerlei Beziehung zueinander, und die o. g. Regeln sind in diesem Fall bedeutungslos.

5.10 Die Wurzelklasse `Object`

- Wenn eine Klasse keine explizite Oberklasse besitzt, besitzt sie implizit die Wurzelklasse `Object` als Oberklasse.
- Daraus folgt, dass `Object` eine direkte oder indirekte Oberklasse jeder Klasse ist.
- Daher kann eine Variable mit statischem Typ `Object` zur Laufzeit Objekte beliebiger Klassen referenzieren.
- Die Klasse `Object` besitzt einen öffentlichen parameterlosen Konstruktor und (unter anderem) die folgenden öffentlichen Methoden (die von Unterklassen überschrieben werden können):

- `boolean equals (Object other)`
  Dient grundsätzlich zum Vergleich des aktuellen Objekts `this` mit irgendeinem anderen Objekt `other`.
  Wenn die Methode nicht überschrieben ist, liefert sie genau dann `true`, wenn `this` und `other` *dasselbe* Objekt referenzieren.
  Um die Methode zu überschreiben, muss man (wie immer) eine Methode mit derselben Signatur definieren; insbesondere muss als Parametertyp `Object` (und nicht die aktuelle Klasse) verwendet werden.
- `int hashCode ()`
  Liefert grundsätzlich einen Hashwert für das aktuelle Objekt `this`. Für Objekte, die bzgl. `equals` „gleich“ sind, muss derselbe Hashwert geliefert werden (d. h. wenn man `equals` überschreibt, muss man i. d. R. auch `hashCode` überschreiben). „Verschiedene“ Objekte sollten nach Möglichkeit verschiedene Hashwerte besitzen (was aber häufig nicht möglich ist).
  Wenn die Methode nicht überschrieben ist, liefert sie typischerweise die interne Adresse des aktuellen Objekts als ganze Zahl.
- `String toString ()`
  Liefert grundsätzlich eine Zeichenketten-Darstellung des aktuellen Objekts `this` und wird automatisch aufgerufen, um das Objekt in einen String umzuwandeln (vgl. § 3.2.4).
  Wenn die Methode nicht überschrieben ist, liefert sie den Namen der Klasse des Objekts, gefolgt von einem @-Zeichen, gefolgt von einer hexadezimalen Darstellung des Hashwerts des Objekts.

Beispiel

```
// Punkt im zweidimensionalen Raum.
class Point {
  // Koordinaten des Punkts.
  public final double x, y;

  // Punkt mit Koordinaten x und y konstruieren.
  public Point (double x, double y) {
    this.x = x;
    this.y = y;
  }

  // Zeichenketten-Darstellung des aktuellen Objekts liefern.
  public String toString () {
    return "(" + x + ", " + y + ")";
  }
```

```
  // Vergleich des aktuellen Objekts this (mit Typ Point)
  // mit irgendeinem anderen Objekt other (mit beliebigem
  // dynamischem Typ).
  public boolean equals (Object other) {
    // 1. Wenn other kein Point ist, kann es nicht gleich this sein.
    if (!(other instanceof Point)) return false;

    // 2. Andernfalls kann other in Point that umgewandelt werden.
    Point that = (Point)other;

    // 3. Dann können this und that inhaltlich verglichen werden.
    return this.x == that.x && this.y == that.y;
  }

  // Hashwert für das aktuelle Objekt liefern.
  public int hashCode () {
    // Für Punkte, die gemäß equals gleich sind,
    // erhält man den gleichen Hashwert.
    return (int)(x + y);
  }
}
```

Unterschied zwischen Überschreiben und Überladen von `equals`

- Wenn man `a` und `b` jeweils durch eine der folgenden Variablen ersetzt, liefert `a.equals(b)` die in der Tabelle dargestellten Resultatwerte (T bedeutet `true`, F bedeutet `false`):

```
Point p1 = new Point(3, 4);
Point p2 = new Point(2, 2);
Object p3 = new Point(3, 4);

String s1 = "hello";
String s2 = "world";
Object s3 = "hello";
```

a \ b	p1	p2	p3	s1	s2	s3
p1	T	F	***T***	F	F	F
p2	F	T	F	F	F	F
p3	***T***	F	T	F	F	F
s1	F	F	F	T	F	T
s2	F	F	F	F	T	F
s3	F	F	F	T	F	T

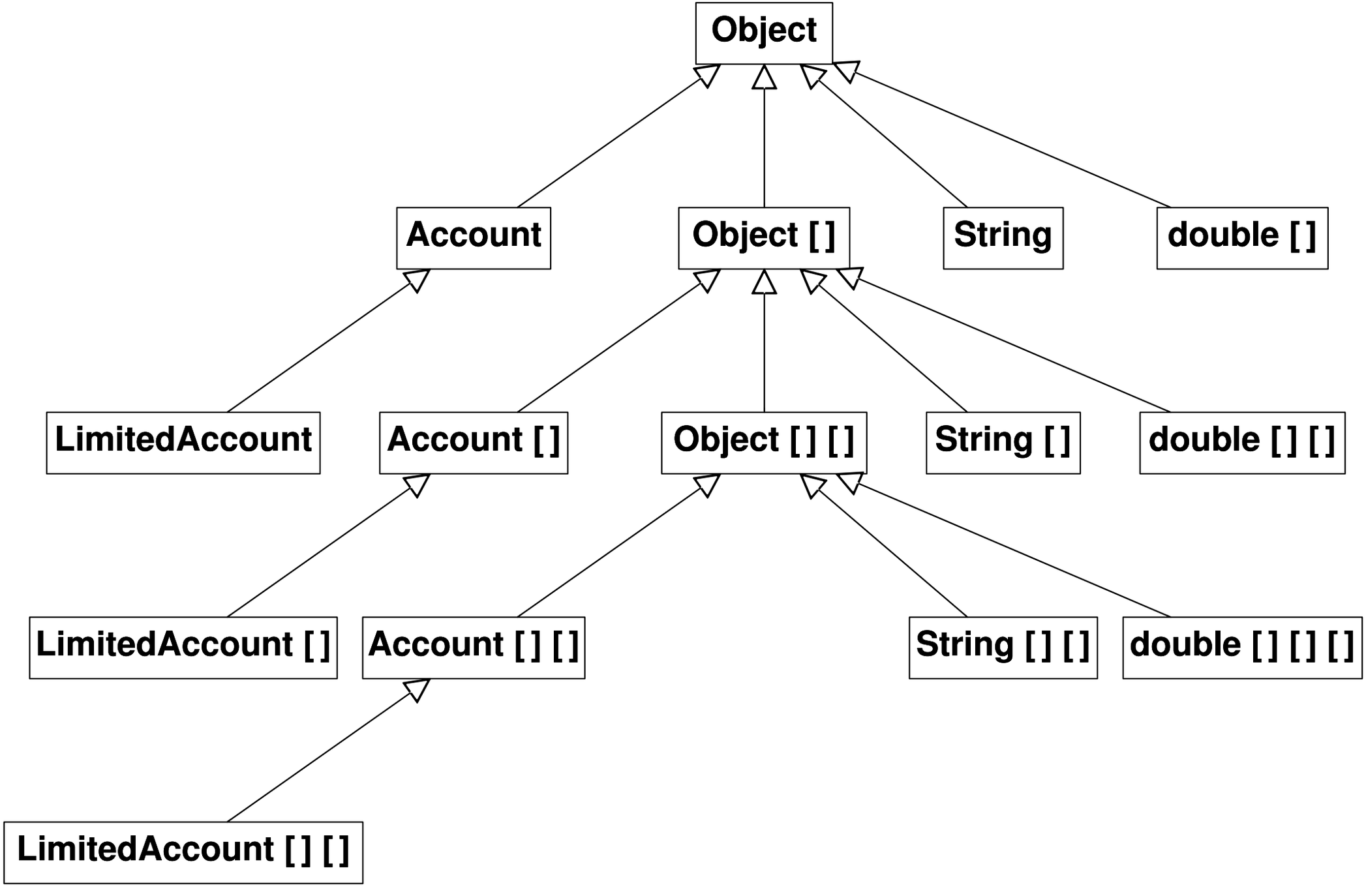
- Wenn man die Methode `equals` in der Klasse `Point` nicht überschreibt, sondern durch die folgende andere Methode mit Parametertyp `Point` *überlädt* – was auf den ersten Blick einfacher und naheliegender erscheint –, dann wird bei `p3.equals(b)` immer die Methode der Klasse `Object` ausgeführt, die für `b` gleich `p1` den unzutreffenden Wert `false` liefert. Ebenso wird bei `p1.equals(p3)` die „falsche“ Methode mit dem falschen Ergebnis ausgeführt. (Das heißt, die beiden in der Tabelle hervorgehobenen Werte stimmen dann nicht mehr.)

```
// Vergleich des aktuellen Objekts this (mit Typ Point)
// mit einem anderen Objekt that des Typs Point.
public boolean equals (Point that) {
  return this.x == that.x && this.y == that.y;
}
```

- Deshalb sollte `equals` immer durch eine Methode mit Parametertyp `Object` *überschrieben* werden, auch wenn dies die Implementierung etwas verkompliziert.

5.11 Arrays

- Arraytypen sind formal ebenfalls Klassen (mit einer öffentlichen, unveränderlichen Objektvariablen `length`) und somit Unterklassen von `Object`.
- Dementsprechend besitzen sie auch die Methoden `equals`, `hashCode` und `toString`, die jedoch nicht überschrieben sind. (Insbesondere vergleicht `a.equals(b)` nur die zwei Referenzen `a` und `b` und führt keinen elementweisen Vergleich durch.)
- Außerdem können Arraytypen genauso wie Klassen in dynamischen Typtests (`instanceof`) und statischen Typumwandlungen (Cast) verwendet werden.
- Wenn `S` ein Untertyp von `T` ist, dann ist auch `S []` ein Untertyp von `T []`, d. h. ein `S`-Array kann überall verwendet werden, wo ein `T`-Array erwartet wird.
- Durch wiederholte Anwendung dieser Regel ergibt sich auch, dass `S [] []` ein Untertyp von `T [] []` ist, usw.
- Außerdem folgt für jede Klasse (oder Schnittstelle) `T`, dass der Arraytyp `T []` ein Untertyp von `Object []` ist.
- Abgesehen von diesen Sonderregeln, können Arraytypen jedoch nicht als Oberklassen anderer Klassen verwendet werden.


Object
Account
Object [ ]
String
double [ ]
LimitedAccount
Account [ ]
Object [ ] [ ]
String [ ]
double [ ] [ ]
LimitedAccount [ ]
Account [ ] [ ]
String [ ] [ ]
double [ ] [ ] [ ]
LimitedAccount [ ] [ ]

- Wenn `a` eine Variable des Typs `T []` ist, akzeptiert der Compiler Zuweisungen der Art `a[i] = t`, sofern `t` den (statischen) Typ `T` besitzt. Wenn `a` zur Laufzeit jedoch ein Array des Typs `S []` referenziert, ist eine solche Zuweisung nur dann semantisch korrekt, wenn `t` den dynamischen Typ `S` besitzt; wenn dies nicht der Fall ist, erhält man eine Ausnahme des Typs `ArrayStoreException`.

```
// LimitedAccount-Array las erzeugen und mit Objekten füllen.
LimitedAccount [] las = new LimitedAccount [10];
las[0] = new LimitedAccount(...);
......

// las an Account-Array-Variable as zuweisen.
Account [] as = las;

// Lesezugriffe über as sind unkritisch.
Account a = as[0];

// Zuweisungen von LimitedAccount-Objekten sind korrekt.
as[0] = new LimitedAccount(...);

// Zuweisungen von Account-Objekten sind nicht zulässig,
// da das von as referenzierte Array ein LimitedAccount-Array ist.
as[0] = new Account(...);        // ArrayStoreException!!
```

5.12 Unveränderliche Methoden und Klassen

- Eine Methode, die `final` deklariert ist, kann in Unterklassen nicht überschrieben oder verborgen werden.
  (Dementsprechend sind private Methoden quasi `final`; vgl. § 5.9.3.)

- Eine Klasse, die `final` deklariert ist, kann keine Unterklassen besitzen (und dementsprechend können ihre Methoden nicht überschrieben oder verborgen werden, d. h. sie sind implizit quasi ebenfalls `final`).

- Beide Arten von `final`-Deklarationen erlauben einem Compiler u. U., effizienteren Code zu generieren, weil Methoden, die nicht überschrieben werden können, statisch gebunden und ggf. inline expandiert werden können.

- Andererseits erschweren oder verhindern derartige `final`-Deklarationen spätere (möglicherweise unvorhergesehene) Erweiterungen eines bestehenden Software-Systems um neue Unterklassen und widersprechen damit eigentlich einem Grundprinzip der Objektorientierung.

- Die Bibliotheksklasse `String` ist `final`.

- Arraytypen sind „semi-`final`“, da sie nicht als Oberklassen anderer Klassen verwendet werden können (vgl. § 5.11).

5.13 Beispiele: Listen (vgl. § 4.13)

5.13.1 Doppelt verkettete Listen

```
// Unterklasse von List: Doppelt verkettete Liste.
class BiList extends List {
  // Zusätzliche Objektvariable:
  protected BiList prev; // Verweis auf voriges Listenelement.

  // Konstruktoren: Doppelt verkettete Liste mit erstem
  // Element h und ggf. Restliste t konstruieren.
  public BiList (int h, BiList t) {
    // Konstruktor der Oberklasse List aufrufen,
    // um deren Objektvariablen head und tail zu initialisieren.
    super(h, t);          // Implizite Aufwärtsumwandlung von t.

    // Rückwärtsverkettung über Objektvariable prev herstellen.
    if (t != null) t.prev = this;
    prev = null;
  }
  public BiList (int h) {
    this(h, null);
  }
```

```
  // Zusätzliche Objektmethoden: Vorgänger und Nachfolger abfragen.
  public BiList next () {
    return (BiList)tail; // Explizite Abwärtsumwandlung notwendig,
                         // da tail vom (statischen) Typ List ist.
  }
  public BiList prev () {
    return prev;
  }

  // Die geerbten Objektmethoden head, tail, length und print
  // werden unverändert übernommen.

}
```

5.13.2 Zirkuläre Listen

```
// Unterklasse von List: Zirkuläre Liste.
class CircList extends List {
  // Keine zusätzlichen Objektvariablen.

  // Konstruktoren: Zirkuläre Liste mit erstem
  // Element h und ggf. Restliste t konstruieren.
  public CircList (int h, CircList t) {
    // Konstruktor der Oberklasse List aufrufen,
    // um deren Objektvariablen head und tail zu initialisieren.
    super(h, t);          // Implizite Aufwärtsumwandlung von t.

    // Zirkuläre Verkettung herstellen.
    if (t == null) {
      tail = this;        // Implizite Aufwärtsumwandlung von this.
    }
    else {
      List p = t;         // Implizite Aufwärtsumwandlung von t.
      while (p.tail != t) p = p.tail;   // Dto.
      p.tail = this;      // Implizite Aufwärtsumwandlung von this.
    }
  }
```

```
  public CircList (int h) { this(h, null); }

  // Überschriebene Objektmethode mit kovarianter Anpassung
  // des Resultattyps (CircList statt List).
  public CircList tail () { return (CircList)tail; }

  // Überschriebene Objektmethode:
  // Länge der zirkulären Liste ermitteln.
  public int length () {
    int n = 1;
    for (List p = tail; p != this; p = p.tail) n++;
    return n;
  }

  // Überschriebene Objektmethode:
  // Zirkuläre Liste ausgeben.
  public void print () {
    System.out.println(head);
    for (List p = tail; p != this; p = p.tail) {
      System.out.println(p.head);
    }
  }
}
```

6 Abstrakte Klassen und Schnittstellen

6.1 Abstrakte Klassen und Methoden

6.1.1 Grundsätzliches

Abstrakte Klassen

- Eine Klasse kann mit dem Schlüsselwort `abstract` deklariert werden, um anzuzeigen, dass sie (technisch oder logisch) unvollständig ist.
- Von einer abstrakten Klasse können keine Objekte erzeugt werden.
- Eine abstrakte Klasse kann trotzdem Konstruktoren besitzen, die (nur) von Konstruktoren ihrer Unterklassen aufgerufen werden können (und daher sinnvollerweise `protected` sind).
- Aufgrund der üblichen Untertyp-Polymorphie, können Variablen einer abstrakten Klasse Objekte von (konkreten) Unterklassen referenzieren.

Abstrakte Methoden

- Eine Methode kann mit dem Schlüsselwort `abstract` deklariert werden, um lediglich ihre Signatur und ihren Resultattyp zu vereinbaren, aber noch keine Implementierung anzugeben. (Private, unveränderliche und statische Methoden können nicht abstrakt sein.)

- Abstrakte Methoden können in Unterklassen durch konkrete Methoden mit derselben Signatur überschrieben (d. h. implementiert) werden.

- Eine Klasse, die abstrakte Methoden deklariert oder erbt und nicht implementiert, muss abstrakt sein.

6.1.2 Beispiel

```
// Abstrakte Klasse: Allgemeines geometrisches Objekt.
abstract class Figure {
  // Unterklassenöffentliche Objektvariablen: Breite und Höhe.
  protected double width, height;

  // Unterklassenöffentlicher Konstruktor:
  // Breite und Höhe initialisieren.
  protected Figure (double w, double h) {
    width = w;
    height = h;
  }

  // Öffentliche Objektmethoden: Breite und Höhe abfragen.
  public double width () { return width; }
  public double height () { return height; }

  // Öffentliche abstrakte Methode: Fläche berechnen.
  public abstract double area ();
}
```

```
// Konkrete Unterklasse von Figure: Rechteck.
class Rectangle extends Figure {
  // Öffentlicher Konstruktor.
  public Rectangle (double w, double h) { super(w, h); }

  // Implementierung der geerbten abstrakten Methode area.
  public double area () { return width * height; }

  // Zusätzliche Objektmethode: Diagonale berechnen.
  public double diag () {
    return Math.sqrt(width*width + height*height);
  }
}

// Konkrete Unterklasse von Figure: Ellipse.
class Ellipse extends Figure {
  // Öffentlicher Konstruktor.
  public Ellipse (double w, double h) { super(w, h); }

  // Implementierung der geerbten abstrakten Methode area.
  public double area () { return Math.PI/4 * width * height; }
}
```

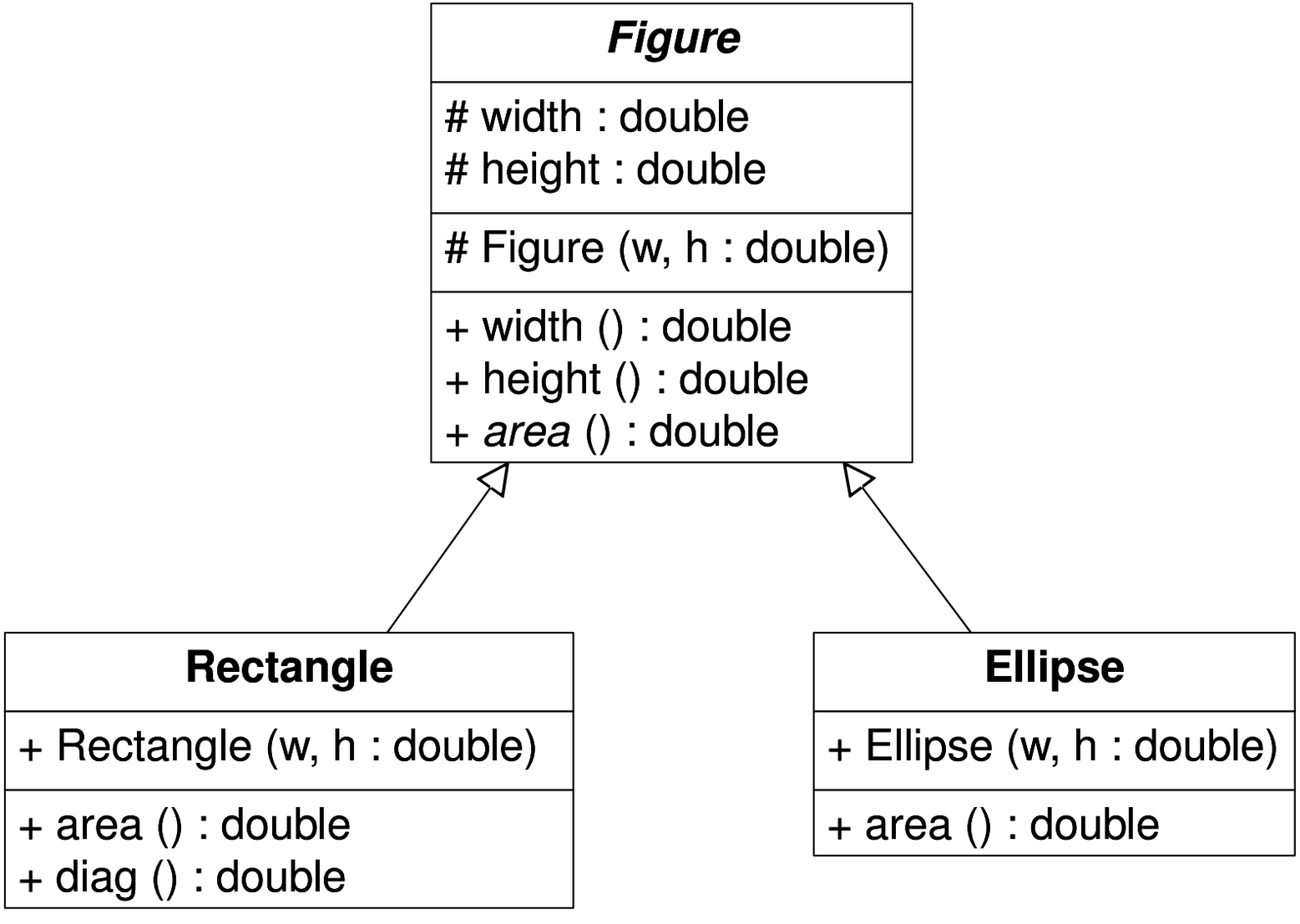


Figure
width : double
height : double
Figure (w, h : double)
+ width () : double
+ height () : double
+ area () : double
Rectangle
+ Rectangle (w, h : double)
+ area () : double
+ diag () : double
Ellipse
+ Ellipse (w, h : double)
+ area () : double

```
// Testklasse (kann prinzipiell abstrakt sein).
abstract class Test {
  // Aufruf zum Beispiel: java Test r 2.5 3.2 e 1.7 5 ...
  public static void main (String [] args) {
    // Unterschiedliche geometrische Objekte erzeugen.
    Figure [] figs = new Figure [args.length/3];
    for (int i = 0; i < args.length; i += 3) {
      char x = args[i].charAt(0);
      double w = Double.parseDouble(args[i+1]);
      double h = Double.parseDouble(args[i+2]);
      if (x == 'r') figs[i/3] = new Rectangle(w, h);
      else figs[i/3] = new Ellipse(w, h);
    }
    // Fläche und ggf. Diagonale aller Objekte ausgeben.
    for (int i = 0; i < figs.length; i++) {
      System.out.print(figs[i].area());
      if (figs[i] instanceof Rectangle) {
        System.out.print(" " + ((Rectangle)figs[i]).diag());
      }
      System.out.println();
    }
  }
}
```

6.2 Schnittstellen

6.2.1 Grundsätzliches

- Eine *Schnittstelle* (*interface*) entspricht einer abstrakten Klasse, die ausschließlich abstrakte Methoden, Konstanten (d. h. unveränderliche statische Felder) und ggf. geschachtelte Klassen und Schnittstellen enthält.

- Alle Bestandteile einer Schnittstelle sind implizit und zwingend öffentlich, d. h. das Schlüsselwort `public` kann, muss aber nicht angegeben werden, während `protected` und `private` verboten sind.

- Alle Methoden einer Schnittstelle sind implizit und zwingend abstrakt, d. h. das Schlüsselwort `abstract` kann, muss aber nicht angegeben werden, während die Angabe eines Methodenrumpfs verboten ist.

- Alle Felder einer Schnittstelle sind implizit statisch und unveränderlich, d. h. die Schlüsselwörter `static` und `final` können, müssen aber nicht angegeben werden. Jedes Feld muss einen Initialisierungsausdruck besitzen, der einmalig beim Laden/Initialisieren der Schnittstelle ausgewertet wird.

- Eine Schnittstelle besitzt keinerlei Konstruktoren, weder explizit deklarierte noch einen implizit deklarierten parameterlosen Konstruktor.

- Eine Schnittstelle kann eine oder mehrere andere Schnittstellen erweitern (Schlüsselwort `extends`), d. h. Schnittstellen erlauben Mehrfachvererbung.

- Eine (abstrakte oder konkrete) Klasse kann eine oder mehrere Schnittstellen *implementieren* (Schlüsselwort `implements`), um anzuzeigen, dass sie die durch die Schnittstellen repräsentierten Eigenschaften besitzt.
  Wenn die Klasse konkret ist, muss sie Implementierungen für alle abstrakten Methoden enthalten.

- Eine Variable, deren Typ eine Schnittstelle ist, kann Objekte aller Klassen referenzieren, die die Schnittstelle (direkt oder indirekt) implementieren.

- Typtests und Typumwandlungen (vgl. § 5.7) funktionieren auch für Schnittstellen. Eine Schnittstelle wird hierbei als Obertyp aller Klassen betrachtet, die die Schnittstelle direkt oder indirekt implementieren. Außerdem wird jede Schnittstelle als Untertyp der Wurzelklasse `Object` betrachtet.
  Anders als in § 5.7, wo nur Klassen betrachtet wurden, sind mit Schnittstellen auch *Querumwandlungen* (cross-casts) möglich, d. h. Umwandlungen, bei denen Ursprungs- und Zieltyp in keiner Ober- oder Untertypbeziehung zueinander stehen.

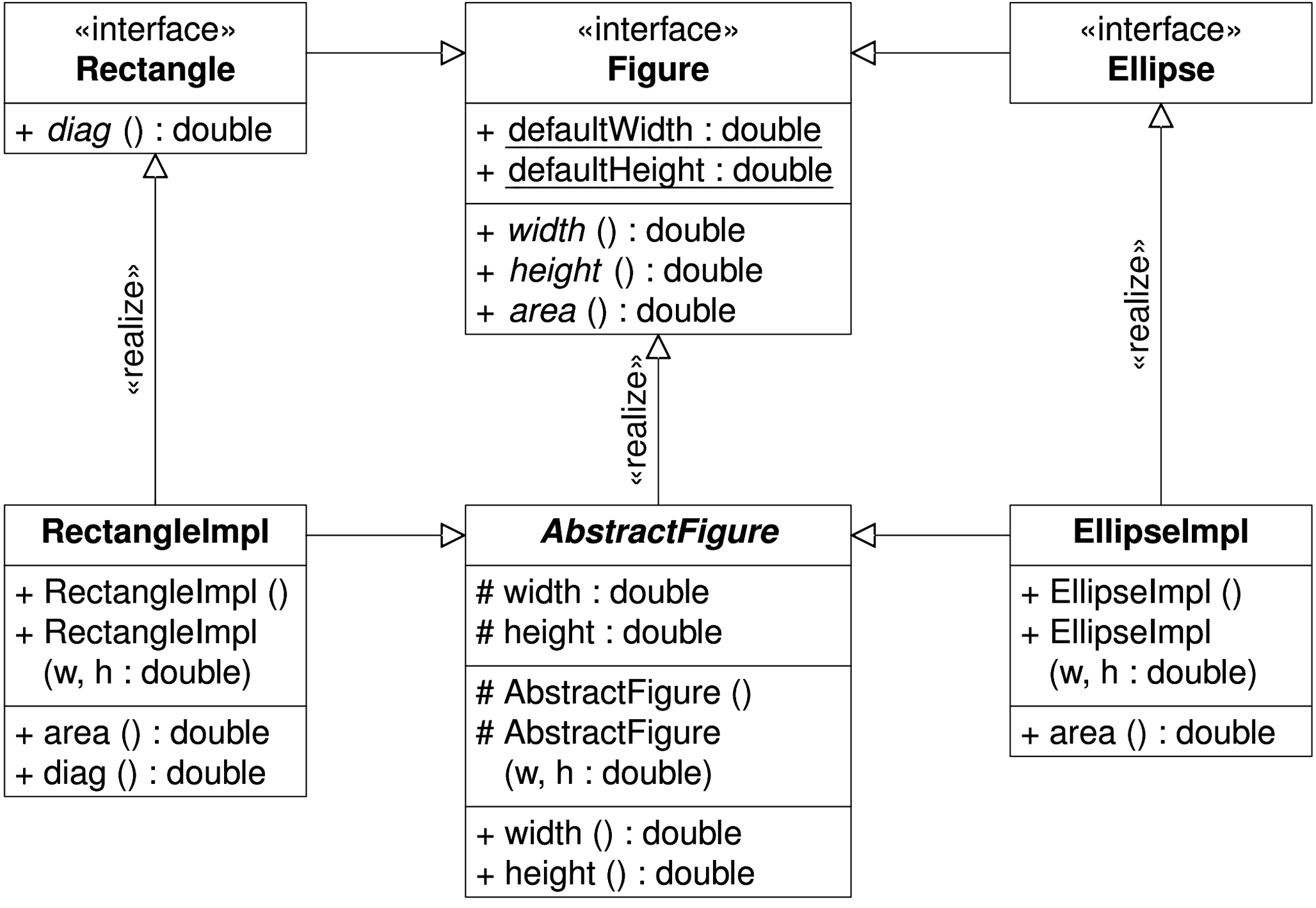
- Eine Schnittstelle kann leer sein, d. h. insbesondere keine Methoden deklarieren.
  Wenn eine Klasse eine solche Schnittstelle implementiert, zeigt sie damit lediglich an, dass sie logisch eine bestimmte Eigenschaft besitzt (vgl. § 8.8).

6.2.2 Beispiel (vgl. § 6.1.2)

```
// Schnittstelle für beliebige geometrische Objekte.
interface Figure {
  // Methoden einer Schnittstelle sind implizit public abstract.
  double width ();       // Breite.
  double height ();      // Höhe.
  double area ();        // Fläche.

  // Felder einer Schnittstelle sind implizit public static final.
  double defaultWidth = 4;
  double defaultHeight = 3;
}

// Schnittstelle für Rechtecke.
interface Rectangle extends Figure {
  double diag ();        // Diagonale.
}

// Schnittstelle für Ellipsen.
interface Ellipse extends Figure {
  // Keine zusätzlichen Methoden.
}
```

```
// Abstrakte Klasse zur teilweisen Implementierung
// der Schnittstelle Figure.
abstract class AbstractFigure implements Figure {
  // Unterklassenöffentliche Objektvariablen: Breite und Höhe.
  protected double width, height;

  // Unterklassenöffentliche Konstruktoren:
  // Breite und Höhe initialisieren.
  protected AbstractFigure (double w, double h) {
    width = w;
    height = h;
  }
  protected AbstractFigure () {
    this(defaultWidth, defaultHeight);
  }

  // Implementierung der Schnittstellen-Methoden width und height.
  public double width () { return width; }
  public double height () { return height; }

  // Die Schnittstellen-Methode area wird nicht implementiert.
}
```

```
// Konkrete Klasse zur Implementierung von Rechtecken.
class RectangleImpl extends AbstractFigure implements Rectangle {
  // Öffentliche Konstruktoren.
  public RectangleImpl (double w, double h) { super(w, h); }
  public RectangleImpl () { super(); }

  // Implementierung der Schnittstellen-Methoden area und diag.
  public double area () { return width * height; }
  public double diag () {
    return Math.sqrt(width*width + height*height);
  }
}

// Konkrete Klasse zur Implementierung von Ellipsen.
class EllipseImpl extends AbstractFigure implements Ellipse {
  // Öffentliche Konstruktoren.
  public EllipseImpl (double w, double h) { super(w, h); }
  public EllipseImpl () { super(); }

  // Implementierung der Schnittstellen-Methode area.
  public double area () { return Math.PI/4 * width * height; }
}
```


«interface»
Rectangle
+ diag () : double
«interface»
Figure
+ defaultWidth : double
+ defaultHeight : double
+ width () : double
+ height () : double
+ area () : double
«interface»
Ellipse
«realize»
«realize»
«realize»
RectangleImpl
+ RectangleImpl ()
+ RectangleImpl
(w, h : double)
+ area () : double
+ diag () : double
AbstractFigure
width : double
height : double
AbstractFigure ()
AbstractFigure
(w, h : double)
+ width () : double
+ height () : double
EllipseImpl
+ EllipseImpl ()
+ EllipseImpl
(w, h : double)
+ area () : double

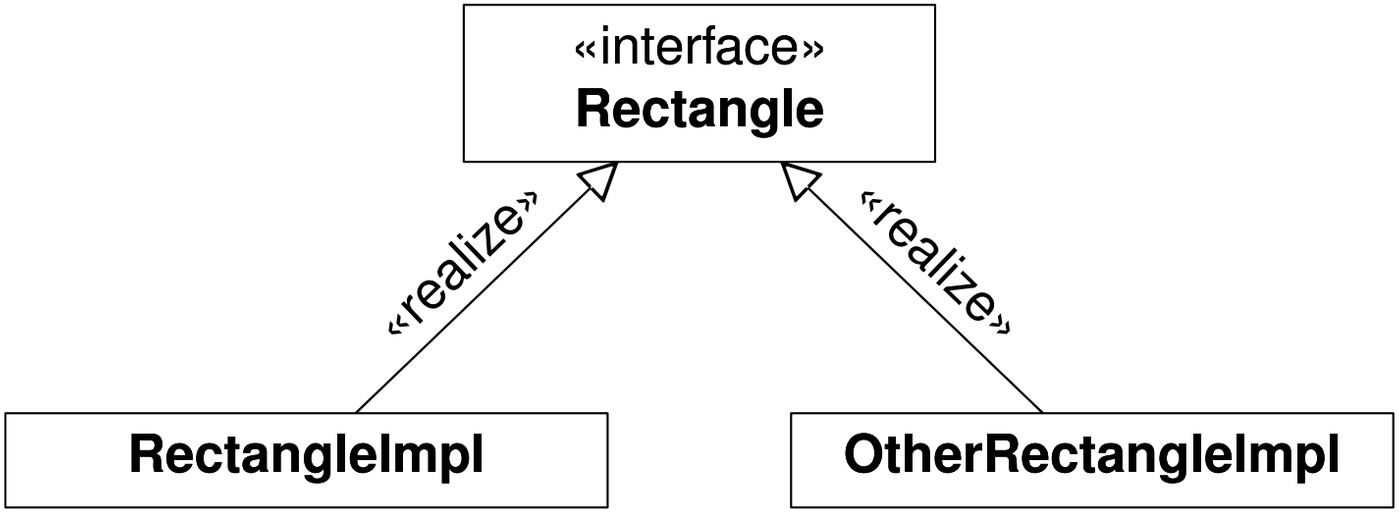
```
// Testklasse (kann prinzipiell abstrakt sein).
abstract class Test {
  // Aufruf zum Beispiel: java Test r 2.5 3.2 e 1.7 5 ...
  public static void main (String [] args) {
    // Unterschiedliche geometrische Objekte erzeugen.
    Figure [] figs = new Figure [args.length/3];
    for (int i = 0; i < args.length; i += 3) {
      char x = args[i].charAt(0);
      double w = Double.parseDouble(args[i+1]);
      double h = Double.parseDouble(args[i+2]);
      if (x == 'r') figs[i/3] = new RectangleImpl(w, h);
      else figs[i/3] = new EllipseImpl(w, h);
    }
    // Fläche und ggf. Diagonale aller Objekte ausgeben.
    for (int i = 0; i < figs.length; i++) {
      System.out.print(figs[i].area());
      if (figs[i] instanceof Rectangle) {
        System.out.print(" " + ((Rectangle)figs[i]).diag());
      }
      System.out.println();
    }
  }
}
```

Anmerkungen

- Obwohl die Implementierung mit Schnittstellen umfangreicher und „komplizierter“ ist als die Implementierung in § 6.1.2, besitzt sie mindestens zwei wichtige Vorteile:

- Schnittstellen (interfaces) und zugehörige Implementierungen (Klassen) sind vollständig voneinander getrennt, was zu einer flexibleren und leichter erweiterbaren Systemarchitektur führt.

- Zu einer Schnittstelle kann es mehrere unterschiedliche Implementierungsklassen geben, wobei Code, der nur die Schnittstellenmethoden verwendet, mit jeder dieser Klassen unverändert funktioniert. Zum Beispiel:



```
// Andere Klasse zur Implementierung von Rechtecken.
class OtherRectangleImpl extends AbstractFigure
                         implements Rectangle {
  // Zusätzliche Objektvariablen
  // zur Speicherung von Fläche und Diagonale.
  private double area, diag;

  // Konstruktor berechnet sofort Fläche und Diagonale.
  public OtherRectangleImpl (double w, double h) {
    super(w, h);
    area = w * h;
    diag = Math.sqrt(w*w + h*h);
  }
  public OtherRectangleImpl () {
    this(defaultWidth, defaultHeight);
  }

  // Objektmethoden liefern nur die zuvor berechneten Werte.
  public double area () { return area; }
  public double diag () { return diag; }
}
```

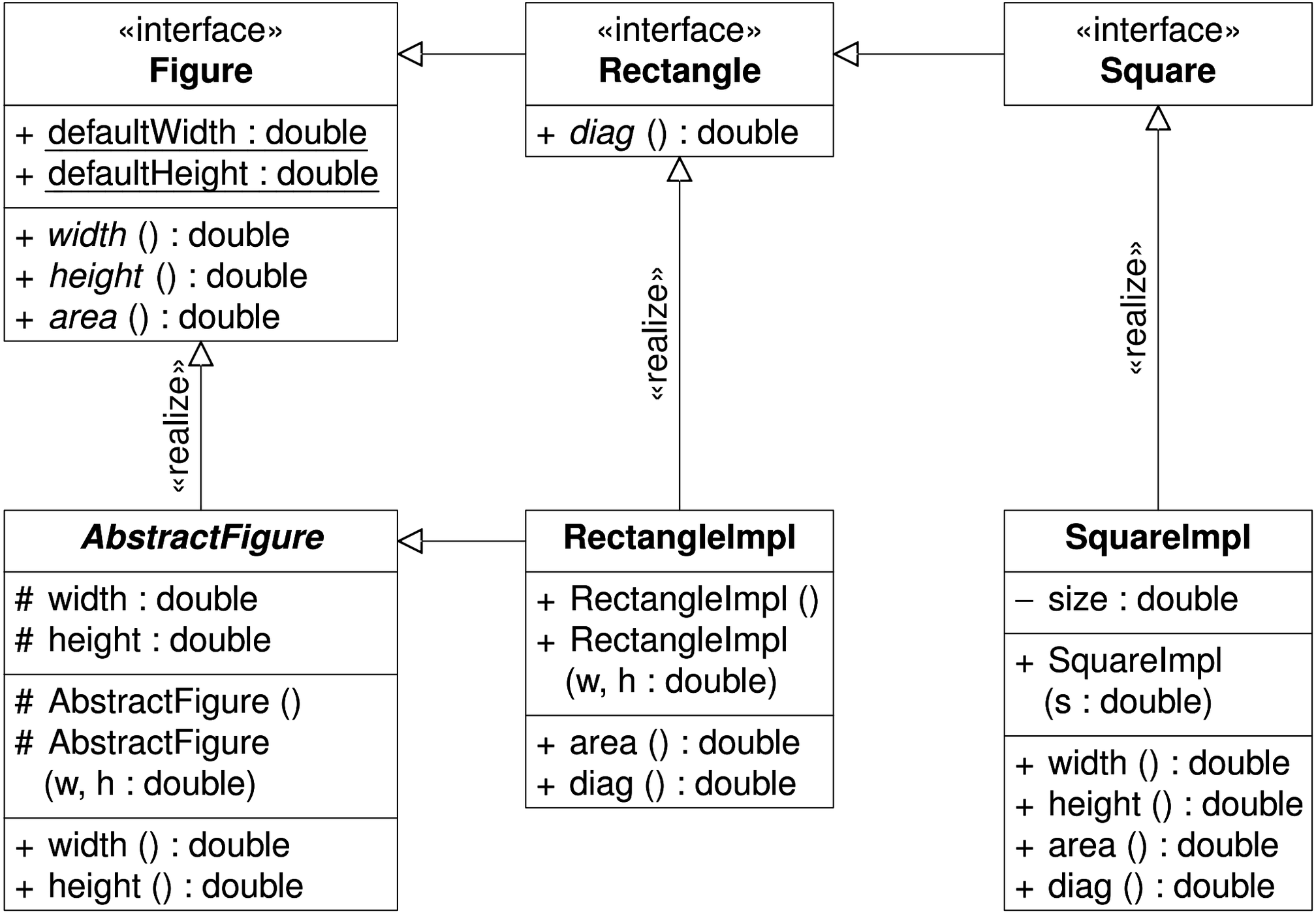
6.2.3 Typ- und Implementierungshierarchie

- Wenn man das vorige Beispiel noch um Quadrate erweitert, sollte `Square` ein Untertyp von `Rectangle` sein, weil ein Quadrat ein Spezialfall eines Rechtecks ist, das die gleichen Eigenschaften (Methoden `width`, `height`, `area` und `diag`) besitzt und daher überall verwendet werden kann, wo ein Rechteck erwartet wird.

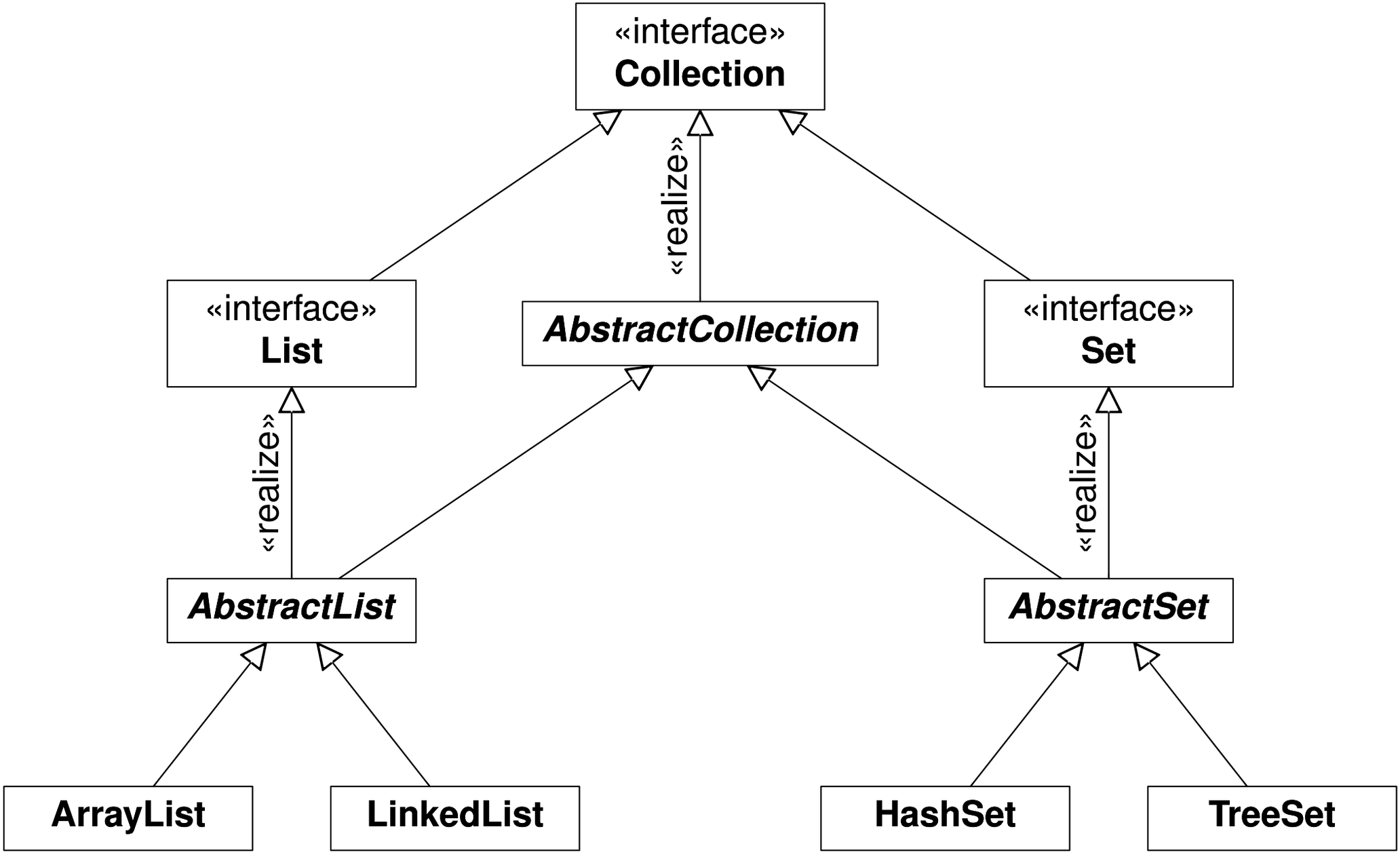
- Wenn man nur mit Klassen arbeitet (§ 6.1.2), muss `Square` dann zwangsläufig auch die Objektvariablen `width` und `height` von `Rectangle` erben, obwohl zur Implementierung eines Quadrats eine einzige Objektvariable `size` ausreichen würde.

- Wenn man Schnittstellen verwendet (§ 6.2.2), können die `extends`-Beziehungen zwischen ihnen unabhängig von den Beziehungen zwischen den zugehörigen Implementierungsklassen sein, d. h. Typ- und Implementierungshierarchie können unterschiedlich strukturiert sein.

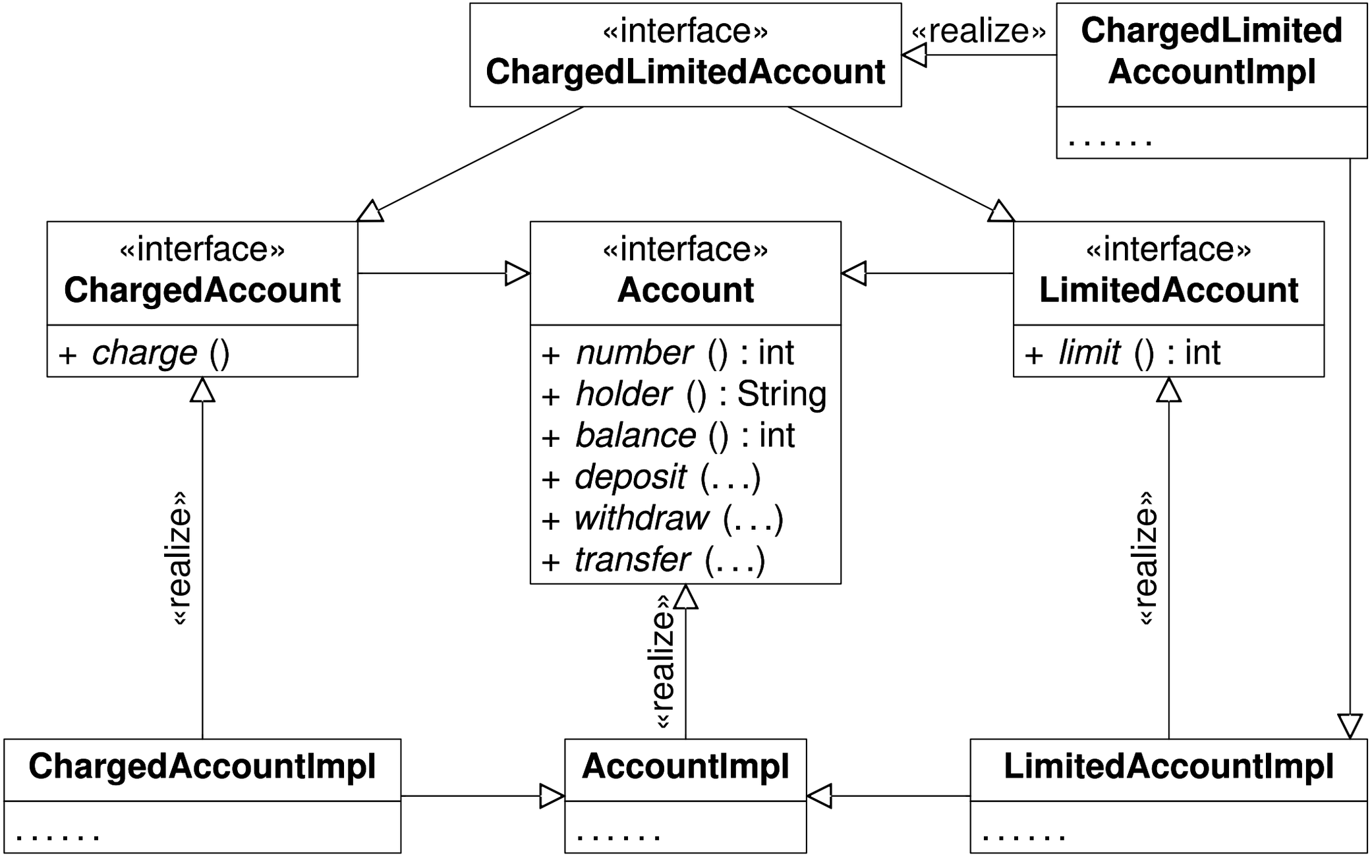
- Insbesondere muss `SquareImpl` nicht von `RectangleImpl` erben.


«interface»
Figure
+ defaultWidth : double
+ defaultHeight : double
+ width () : double
+ height () : double
+ area () : double
«interface»
Rectangle
+ diag () : double
«interface»
Square
«realize»
«realize»
«realize»
AbstractFigure
width : double
height : double
AbstractFigure ()
AbstractFigure (w, h : double)
+ width () : double
+ height () : double
RectangleImpl
+ RectangleImpl ()
+ RectangleImpl (w, h : double)
+ area () : double
+ diag () : double
SquareImpl
– size : double
+ SquareImpl (s : double)
+ width () : double
+ height () : double
+ area () : double
+ diag () : double

6.2.4 Beispiel aus der Java-Standardbibliothek



6.2.5 Mehrfachvererbung mit Schnittstellen



Erläuterungen

- Auf Schnittstellen- bzw. Typebene lässt sich wie gewünscht ausdrücken, dass ein `ChargedLimitedAccount` *sowohl* die Eigenschaften von `ChargedAccount` (zusätzliche Methode `charge` gegenüber `Account`) *als auch* die Eigenschaften von `LimitedAccount` (zusätzliche Methode `limit`) besitzt.

- Auf Klassen- bzw. Implementierungsebene kann `ChargedLimitedAccountImpl` aber nur *entweder* von `ChargedAccountImpl` *oder* von `LimitedAccountImpl` erben.

- Daher wählt man als Oberklasse sinnvollerweise diejenige, von der es „mehr zu erben“ gibt, im Beispiel `LimitedAccountImpl`, weil die Implementierung des Kreditlimits aufwendiger ist als die Implementierung des Gebührenzählers.

- Die Funktionalität der anderen Klasse muss dann notgedrungen repliziert werden.

6.2.6 Namenskonflikte

Objektmethoden

- Wenn eine Klasse oder eine Schnittstelle mehrere Objektmethoden mit derselben Signatur (von ihrer Oberklasse und/oder einer oder mehreren Ober-Schnittstellen) erbt, kann höchstens eine von ihnen (nämlich die von der Oberklasse geerbte) konkret sein, während alle anderen zwangsläufig abstrakt sind.

- Wenn alle genannten Methoden abstrakt sind, werden sie in der Klasse bzw. Schnittstelle quasi zu einer einzigen abstrakten Methode mit der entsprechenden Signatur zusammengefasst.
  Wenn die Methoden unterschiedliche Resultattypen besitzen, muss einer dieser Typen ein trivialer, direkter oder indirekter Untertyp aller anderen sein, und die verschmolzene Methode besitzt diesen Typ als Resultattyp.

- Wenn eine der Methoden konkret ist, implementiert sie alle abstrakten Methoden, d. h. in diesem Fall besitzt die Klasse genau eine konkrete Methode mit der entsprechenden Signatur.
  Wenn die Methoden unterschiedliche Resultattypen besitzen, muss der Resultattyp der konkreten Methode ein trivialer, direkter oder indirekter Untertyp aller anderen Resultattypen sein.

Felder

- Wenn eine Klasse oder eine Schnittstelle mehrere gleichnamige Felder (Objekt- oder Klassenvariablen) erbt, ist der Zugriff auf diese Felder mit dem einfachen Feldnamen `field` *mehrdeutig*.

- Eine solche Mehrdeutigkeit kann mit einer der folgenden Möglichkeiten aufgelöst werden:

  - `super.field`
    (Zugriff auf ein Feld der Oberklasse des aktuellen Objekts)

  - `((supertype)object).field`
    (Zugriff auf ein Feld der Oberklasse oder einer Ober-Schnittstelle `supertype` des Objekts `object`)

  - `type.field`
    (Zugriff auf ein statisches Feld der Klasse oder Schnittstelle `type`)

7 Anonyme Klassen

7.1 Grundsätzliches

- Wenn von einer Klasse nur ein einziges Objekt benötigt wird, kann man auf eine explizite Deklaration der Klasse verzichten und stattdessen eine *anonyme* Klasse verwenden.

- Eine anonyme Klasse wird definiert, indem dem Ausdruck `new type (arguments)` zur Erzeugung ihres einzigen Objekts ein Klassenrumpf `{ ..... }` nachgestellt wird.

- Wenn `type` eine Klasse bezeichnet, dann ist die anonyme Klasse eine direkte Unterklasse dieser Klasse.
  Wenn `type` eine Schnittstelle bezeichnet, dann ist die anonyme Klasse eine direkte Unterklasse der Wurzelklasse `Object`, die diese Schnittstelle implementiert.

- Eine anonyme Klasse kann keinen expliziten Konstruktor besitzen.
  Sie besitzt implizit einen öffentlichen Konstruktor, der die Aufrufparameter `arguments` unverändert an den entsprechenden Oberklassenkonstruktor weiterreicht.

- Anonyme Klassen unterliegen gewissen Einschränkungen; beispielsweise dürfen sie keine statischen Elemente enthalten.

7.2 Beispiel

- Ausgabe einer Wertetabelle für eine beliebige Funktion, die als Parameter übergeben wird

Lösung mit Funktionszeigern in C

```
#include <stdio.h>

// Zeiger auf Funktion mit Parameter- und Resultattyp double.
typedef double (*Function) (double);

// Wertetabelle der Funktion f für x von x1 bis x2
// mit Schrittweite dx ausgeben.
void print (Function f, double x1, double x2, double dx) {
  double x;
  for (x = x1; x <= x2; x += dx) {
    printf("%g\t%g\n", x, f(x));
  }
}
```

```
// Zwei exemplarische Funktionen: x*x und 1/x.
double f1 (double x) { return x*x; }
double f2 (double x) { return 1/x; }

// Testprogramm.
int main () {
  // Wertetabellen von x*x und 1/x ausgeben.
  print(f1, 1, 10, 1);
  print(f2, 1, 10, 1);
  return 0;
}
```

Lösung in Java

```
// Schnittstelle (oder auch abstrakte Klasse) zur Repräsentation
// von Funktionen mit Parameter- und Resultattyp double.
interface Function {
  // Funktionswert an der Stelle x berechnen.
  double compute (double x);
}

class Table {
  // Wertetabelle der Funktion f für x von x1 bis x2
  // mit Schrittweite dx ausgeben.
  public static
  void print (Function f, double x1, double x2, double dx) {
    for (double x = x1; x <= x2; x += dx) {
      System.out.println(x + "\t" + f.compute(x));
    }
  }
}
```

```
// Explizite Klasse zur Repräsentation der Funktion x*x.
class F1 implements Function {
  // Geeignete Implementierung der Schnittstellen-Methode compute.
  public double compute (double x) { return x*x; }
}

class Test {
  public static void main (String [] args) {
    // Wertetabelle von x*x ausgeben.
    Table.print(new F1(), 1, 10, 1);

    // Wertetabelle von 1/x ausgeben.
    Table.print(
      // Objekt einer anonymen Klasse erzeugen,
      // die die Schnittstelle Function implementiert
      // und hierfür eine geeignete Implementierung
      // der Methode compute enthält.
      new Function () {
        public double compute (double x) { return 1/x; }
      },
      1, 10, 1);
  }
}
```

8 Ausnahmen

8.1 Grundsätzliches

- Wenn während der Ausführung eines Java-Programms bestimmte semantische Bedingungen verletzt werden (z. B. ganzzahlige Division durch 0, Dereferenzierung einer Nullreferenz, fehlerhafter Indexwert), wirft die Java Virtual Machine eine entsprechende *Ausnahme* (vgl. § 8.3).

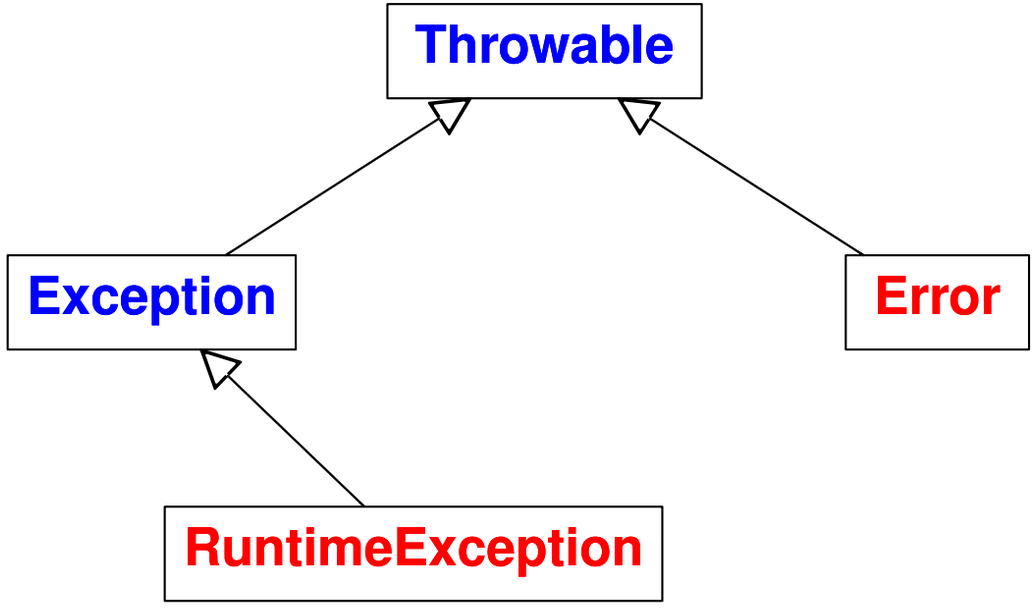
- Ebenso können Ausnahmen explizit mit Hilfe von `throw`-Anweisungen geworfen werden, um semantische Fehler (wie z. B. unzulässige Kontoüberziehungen) zu signalisieren (vgl. § 8.4 und § 8.5).

- Ausnahmen können mit `try-catch`-Anweisungen „aufgefangen“ und behandelt werden, z. B. durch Ausgabe einer Fehlermeldung (vgl. § 8.6).

- Ausnahmen sind Objekte der Klasse `Throwable` oder einer ihrer (direkten oder indirekten) Unterklassen. Sie enthalten mindestens folgende Information:
  - eine Fehlermeldung (z. B. „/ by zero“ oder der konkrete fehlerhafte Indexwert);
  - die aktuelle Aufrufverschachtelung (stack trace).

8.2 Geprüfte und ungeprüfte Ausnahmen

- Es wird zwischen *geprüften* und *ungeprüften* Ausnahmen (checked/unchecked exceptions, in der Abbildung blau/rot) unterschieden:



  Unterklassen von `RuntimeException` und `Error` stellen ungeprüfte Ausnahmen dar, während alle anderen Unterklassen von `Throwable` geprüfte Ausnahmen darstellen.

- Wenn eine Methode oder ein Konstruktor eine geprüfte Ausnahme werfen kann, muss dies im Kopf *deklariert* werden (vgl. § 8.7).

8.3 Vordefinierte Ausnahmeklassen

- Bei der Ausführung bestimmter Operatoren können u. U. die folgenden ungeprüften Ausnahmen auftreten (vgl. § 3.3.4):

Operator	*Ausnahme*
`new`	`OutOfMemoryError`
`new []`	`NegativeArraySizeException` `OutOfMemoryError`
`.`	`NullPointerException`
`[]`	`NullPointerException` `ArrayIndexOutOfBoundsException`
`(`*`type`*`)`	`ClassCastException`
`/` `%`	`ArithmeticException`
`=`	`ArrayStoreException`

- Die Methode `clone` der Wurzelklasse `Object` kann eine geprüfte Ausnahme des Typs `CloneNotSupportedException` werfen (vgl. § 8.8).

- Darüber hinaus gibt es zahlreiche weitere geprüfte und ungeprüfte Ausnahmen, die von verschiedenen Methoden der Java-Standardbibliothek geworfen werden können.

8.4 Benutzerdefinierte Ausnahmeklassen

- Jede direkte oder indirekte Unterklasse von `Throwable` ist eine Ausnahmeklasse, d. h. ihre Objekte können mittels `throw` als Ausnahmen geworfen werden (vgl. § 8.5).

- Häufig bieten benutzerdefinierte Ausnahmeklassen analog zu `Throwable` zwei öffentliche Konstruktoren an, die jeweils via `super` den entsprechenden Oberklassenkonstruktor aufrufen:
  - einen parameterlosen Konstruktor;
  - einen Konstruktor mit einem Parameter `message` des Typs `String`.

- Die entsprechenden Konstruktoren von `Throwable` speichern in dem erzeugten Objekt einerseits die optionale Fehlermeldung (die mittels `getMessage` abgefragt werden kann) und andererseits die aktuelle Aufrufverschachtelung (die mittels `printStackTrace` ausgegeben werden kann).

- Je nach Anwendung können aber auch Konstruktoren mit anderen oder zusätzlichen Parametern sinnvoll sein.

Beispiel

```
// Geprüfte Ausnahme zur Signalisierung
// von unzulässigen Kontoüberziehungen.
class OverdrawException extends Exception {
  public final LimitedAccount account;  // Betroffenes Konto.
  public final int amount;              // Unzulässiger Betrag.

  // Konstruktor.
  public OverdrawException (LimitedAccount account, int amount) {
    super();
    this.account = account;
    this.amount = amount;
  }

  // Umwandlung in Zeichenkette.
  public String toString () {
    // super.toString() liefert den Namen der Ausnahmeklasse
    // (d. h. hier "OverdrawException").
    return super.toString() + " on account no. " + account.number();
  }
}
```

8.5 Werfen von Ausnahmen

- Die Anweisung `throw` *`object`* (vgl. § 3.4.1) wirft das Objekt *`object`* als Ausnahme.
- Dadurch werden nacheinander alle dynamisch umschließenden Anweisungen, Methoden, Konstruktoren etc. *abrupt* beendet, bis die Ausnahme von einem `catch`-Block aufgefangen wird (vgl. § 8.6).
- Wenn kein solcher `catch`-Block gefunden wird, wird das Programm (bzw. der aktuelle Thread) beendet.
- Der statische Typ von *`object`* muss ein (trivialer, direkter oder indirekter) Untertyp von `Throwable` sein.
- *`object`* kann prinzipiell ein beliebiger Ausdruck sein, der ein entsprechendes Objekt als Resultat liefert.
  In der Regel handelt es sich um einen Objekterzeugungsausdruck.
- Beispiel (vgl. § 5.6):

```
if (balance() - amount < -limit) {
  throw new OverdrawException(this, amount);
}
```

8.6 Auffangen von Ausnahmen

8.6.1 Syntax einer Try-Anweisung

- Eine `try`-Anweisung (vgl. § 3.4.3) besteht aus einem `try`-Block, gefolgt von einem oder mehreren `catch`-Blöcken und/oder einem `finally`-Block:

```
try {
  ......
}
catch (Exception1 e) {
  ......
}
catch (Exception2 e) {
  ......
}
catch (Exception3 e) {
  ......
}
finally {
  ......
}
```

- Die Parametertypen der `catch`-Blöcke müssen Unterklassen von `Throwable` sein. Wenn eine dieser Klassen eine geprüfte Ausnahme bezeichnet, muss der `try`-Block tatsächlich eine Ausnahme dieses Typs (oder eines Untertyps) werfen können.
- Wenn eine dieser Klassen eine Unterklasse einer anderen ist, muss ihr `catch`-Block vor dem `catch`-Block für die Oberklasse stehen.

8.6.2 Ausführung einer Try-Anweisung

- Zunächst wird der `try`-Block ausgeführt.
- Wenn die Ausführung des `try`-Blocks mit einer *Ausnahme* endet, wird der dynamische Typ des geworfenen Ausnahmeobjekts nacheinander mittels `instanceof` mit den Parametertypen der `catch`-Blöcke verglichen und der erste passende Block ausgeführt; das Ausnahmeobjekt wird zuvor an den Parameter des `catch`-Blocks zugewiesen.
  Wenn kein passender `catch`-Block gefunden wird (insbesondere wenn kein `catch`-Block vorhanden ist), wird die Ausnahme weitergeworfen.
- Wenn die Ausführung des `try`-Blocks *normal* endet oder durch eine *Sprunganweisung* (`break`, `continue` oder `return`) vorzeitig beendet wird, wird kein `catch`-Block ausgeführt.
- Nach Ausführung des `try`-Blocks und ggf. eines `catch`-Blocks wird ein vorhandener `finally`-Block unter allen Umständen ausgeführt.

8.6.3 Ausgang einer Try-Anweisung

❒ Ein Block endet *abrupt*, wenn er eine Ausnahme wirft oder durch eine Sprunganweisung vorzeitig beendet wird. Andernfalls endet der Block *normal*.

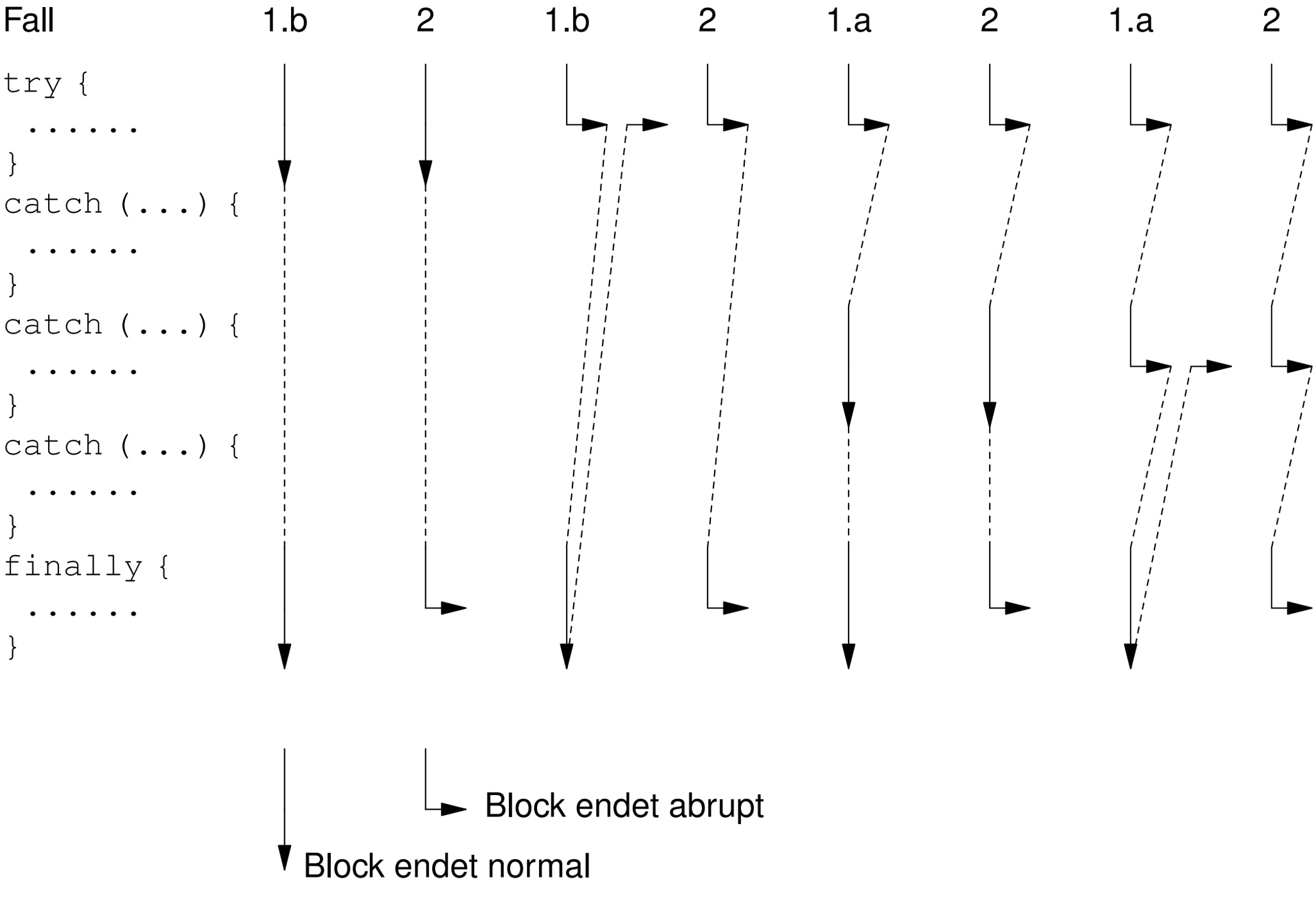
Damit lassen sich folgende Fälle unterscheiden:

1. Der `finally`-Block fehlt oder endet normal.

   a) Der `try`-Block endete mit einer Ausnahme, zu der es einen passenden `catch`-Block gibt.
   In diesem Fall endet die gesamte `try`-Anweisung auf dieselbe Weise wie der `catch`-Block. (Insbesondere kann der `catch`-Block die aufgefangene Ausnahme weiterwerfen oder eine andere Ausnahme werfen, die dann von dieser Try-Anweisung nicht mehr aufgefangen wird).

   b) Der `try`-Block endete normal oder vorzeitig durch eine Sprunganweisung oder mit einer Ausnahme ohne passenden `catch`-Block.
   In diesen Fällen endet die gesamte `try`-Anweisung auf dieselbe Weise wie der `try`-Block.

2. Der `finally`-Block endet abrupt.
   In diesem Fall endet die gesamte `try`-Anweisung auf dieselbe Weise abrupt, egal wie der `try`- und ein ggf. ausgeführter `catch`-Block endeten. (Insbesondere kann die vom `try`- oder `catch`-Block eventuell geworfene Ausnahme verloren gehen.)


Fall
1.b
2
1.b
2
1.a
2
1.a
2
try {
.....
}
catch (...) {
.....
}
catch (...) {
.....
}
catch (...) {
.....
}
finally {
.....
}
Block endet abrupt
Block endet normal

8.6.4 Beispiel

```
try {
  // charAt wirft StringIndexOutOfBoundsException.
  String s = "abc";
  System.out.println(s.charAt(3));
}
catch (StringIndexOutOfBoundsException e) {
  // Die Aufrufverschachtelung von e wird auf System.out ausgegeben.
  e.printStackTrace(System.out);

  // parseInt wirft NumberFormatException, da msg nicht einfach die
  // Ziffernfolge "3" enthält, sdrn. "String index out of range: 3".
  String msg = e.getMessage();
  int i = Integer.parseInt(msg);
  System.out.println(i);
}
catch (NumberFormatException e) {
  // Dieser catch-Block wird NICHT mehr ausgeführt!
  System.out.println("NumberFormatException");
}

// Die gesamte Try-Anweisung endet mit einer NumberFormatException.
```

8.7 Deklaration von Ausnahmen

- Wenn eine Routine (d. h. eine Methode oder ein Konstruktor) eine geprüfte Ausnahme eines bestimmten Typs *werfen kann*, weil ihr Rumpf
  - eine entsprechende `throw`-Anweisung enthält
  - und/oder selbst eine Routine aufruft, die eine solche Ausnahme werfen kann

  und
  - die Ausnahme nicht von einer `try`-Anweisung aufgefangen wird,

  so muss dies im Kopf der Routine durch eine entsprechende `throws`-Klausel deklariert werden.

- Wenn ein Initialisierungsausdruck einer Objektvariablen oder ein Objektinitialisierer eine geprüfte Ausnahme werfen kann, muss diese Ausnahme von jedem Konstruktor der Klasse deklariert werden.
  (Da der implizit definierte Standardkonstruktor keine Ausnahmen deklariert, muss die Klasse mindestens einen expliziten Konstruktor besitzen.)

- Initialisierungsausdrücke von Klassenvariablen und Klasseninitialisierer dürfen keine geprüften Ausnahmen werfen, weil diese nirgends deklariert werden können.

- Wenn eine Methode eine geerbte Methode überschreibt oder verbirgt (vgl. § 5.9), muss ihre `throws`-Klausel logisch eine (echte oder triviale) Teilmenge der `throws`-Klausel der ursprünglichen Methode sein, d. h. es dürfen keine zusätzlichen Ausnahmen deklariert werden.

Beispiel

- Auszug aus der Klasse `LimitedAccount`:

```
// Methode kann OverdrawException direkt via throw werfen.
private void check (int amount) throws OverdrawException {
  if (balance() - amount < -limit) {
    throw new OverdrawException(this, amount);
  }
}

// Methode kann OverdrawException
// indirekt durch Aufruf von check werfen.
public void withdraw (int amount) throws OverdrawException {
  check(amount);
  super.withdraw(amount);
}
```

- Damit die hier gezeigte Überschreibung der Methode `withdraw` in der Klasse `LimitedAccount` korrekt ist, muss bereits die ursprüngliche Methode in der Klasse `Account` mit `throws OverdrawException` deklariert werden, obwohl die Ausnahme von dieser Methode gar nicht geworfen werden kann.

- Begründung: Auch wenn `a` den statischen Typ `Account` besitzt, kann ein Methodenaufruf wie z. B. `a.withdraw(1000)` aufgrund dynamischen Bindens die o. g. Methode `withdraw` der Klasse `LimitedAccount` aufrufen und somit tatsächlich eine Ausnahme des Typs `OverdrawException` werfen.

8.8 Kopieren von Objekten

- Die Wurzelklasse `Object` (vgl. § 5.10) enthält eine Methode

  ```
  protected Object clone () throws CloneNotSupportedException
  ```

  die eine Kopie des aktuellen Objekts erstellt, sofern seine Klasse die Schnittstelle `Cloneable` implementiert; andernfalls erhält man eine geprüfte Ausnahme des Typs `CloneNotSupportedException`.

- `clone` erzeugt eine „flache Kopie“ eines Objekts, d. h. es werden nur die Werte seiner (eigenen und geerbten) Objektvariablen kopiert; referenzierte Objekte werden nicht rekursiv kopiert.

- Damit Klienten die Methode `clone` aufrufen können, überschreibt eine Klasse `C`, die `Cloneable` implementiert, diese Methode in der Regel wie folgt durch eine öffentliche Methode, die die Ausnahme `CloneNotSupportedException` nicht mehr wirft und als Resultattyp `C` statt `Object` besitzen kann:

  ```
  public C clone () {
    try { return (C)super.clone(); }
    catch (CloneNotSupportedException e) { return null; }
  }
  ```

- Gegebenenfalls kann das von `super.clone` erzeugte Objekt noch verändert werden, z. B. indem referenzierte Objekte rekursiv kopiert werden (vgl. § 4.13):

```
class List implements Cloneable {
  ......

  public List clone () {
    try {
      List c = (List)super.clone();
      if (tail != null) c.tail = tail.clone();
      return c;
    }
    catch (CloneNotSupportedException e) {
      return null;
    }
  }
}
```

Anmerkungen

- Die Methode `clone` wird nicht in der Schnittstelle `Cloneable`, sondern in der Klasse `Object` deklariert. Das Implementieren der leeren Schnittstelle dient nur als Kennzeichnung, dass `clone` Objekte der implementierenden Klasse kopieren darf.
- Wenn ein Objekt mittels `clone` erzeugt wird, finden keine der in § 4.9 genannten Initialisierungen statt; insbesondere wird kein Konstruktor der Klasse ausgeführt.
- Die Implementierung von `clone` in der Klasse `Object` kann ein Objekt auch dann kopieren, wenn es nicht-öffentliche geerbte Objektvariablen enthält, die man selbst nicht kopieren könnte.
- Auch wenn eine Klasse `Cloneable` implementiert und die von `Object` geerbte Methode `clone` daher zur Laufzeit keine Ausnahme werfen wird, kann der Aufruf `super.clone()` aus Sicht des Compilers trotzdem eine Ausnahme des Typs `CloneNotSupportedException` werfen, die abgefangen (oder im Methodenkopf deklariert) werden muss.
  Damit die Methode `clone` aus Sicht des Compilers unter allen Umständen einen Resultatwert liefert, muss der `catch`-Block eine Dummy-`return`-Anweisung enthalten.
- Jeder Arraytyp implementiert die Schnittstelle `Cloneable` und überschreibt die Methode `clone` so, dass sie öffentlich ist, keine Ausnahme werfen kann und den Arraytyp als Resultattyp besitzt.

8.9 Abschließende Bemerkungen

- Der Ansatz, Fehlersituationen durch das Werfen bzw. Auffangen von Ausnahmen zu signalisieren bzw. zu behandeln, besitzt wesentliche Vorteile gegenüber anderen Mechanismen (z. B. Rückgabe bestimmter Fehlercodes und entsprechende Überprüfungen von Resultatwerten):
  - Normaler Code und Fehlerbehandlung können sauber getrennt in `try`- bzw. `catch`-Blöcken formuliert werden.
  - Da eine nicht aufgefangene Ausnahme letztlich zu einem Programmabbruch führt, können Fehler nicht einfach übersehen oder ignoriert werden.
  - Da eine Ausnahme, die von einer Routine nicht aufgefangen wird, automatisch zu ihrem Aufrufer weitergeworfen wird, müssen Fehler nicht in jeder Routine überprüft und dann entweder behandelt oder explizit weitergeleitet werden.
- Da eine Routine, die eine Fehlersituation entdeckt, häufig nicht weiß, wie der Fehler geeignet behandelt werden soll (z. B. durch Ausgabe auf `System.out` oder durch Öffnen eines Dialogfensters), ist es sinnvoll, dass sie den Fehler nur durch Werfen einer Ausnahme signalisiert und die Behandlung einer anderen Routine überlässt.

- Das Konzept, dass geprüfte Ausnahmen in einer Routine entweder aufgefangen oder im Kopf deklariert werden müssen, ist zwar theoretisch überzeugend, erweist sich in der Praxis aber häufig als lästig – und führt gelegentlich zur Formulierung leerer `catch`-Blöcke, die Ausnahmen unbemerkt „verschlucken“ können und das Konzept damit konterkarieren.

9 Pakete

9.1 Grundsätzliches

- *Pakete* (*packages*) dienen zur Gruppierung logisch zusammengehörender Typen (Klassen und Schnittstellen).

- Jedes Paket stellt einen eigenen *Namensbereich* dar, d. h. unterschiedliche Pakete können unterschiedliche Typen (und Teilpakete; vgl. § 9.2) mit dem gleichen (unqualifizierten) Namen enthalten.
  Die Namen aller Typen und Teilpakete eines Pakets müssen jedoch eindeutig sein.

- Typen können öffentlich (Schlüsselwort `public`) oder paketöffentlich (ohne Angabe einer Zugriffsbeschränkung) sein (vgl. § 4.3).
  Paketöffentliche Typen können nur innerhalb desselben Pakets verwendet werden, öffentliche Typen in allen Paketen.

- Felder und Methoden eines Typs, die paket- oder unterklassenöffentlich sind, können in allen Typen desselben Pakets verwendet werden.

- Um Typen aus anderen Paketen anzusprechen, muss ihr Name entweder mit dem vollständigen Namen des Pakets *qualifiziert* werden (z. B. `java.io.InputStream`), oder der Name muss zuvor *importiert* werden (vgl. § 9.3).

9.2 Paketnamen

- Paketnamen bestehen aus einem oder mehreren *einfachen Namen*, die durch Punkte getrennt sind, z. B. `java`, `java.awt`, `java.awt.color`.
- Ein Paket mit dem Namen `P.Q` (wobei `P` ein vollständiger Paketname und `Q` ein einfacher Name ist) ist ein *Teilpaket* (*subpackage*) des Pakets `P`.
- Für ein Teilpaket gelten dieselben Zugriffsrechte wie für andere Pakete, d. h. es darf ebenfalls nur die öffentlichen Typen seiner übergeordneten Pakete verwenden.
- Paketnamen (wie z. B. `java.awt.color`) werden normalerweise auf entsprechende Verzeichnisnamen (im Beispiel `java/awt/color`) abgebildet.
- Es gibt bestimmte Namenskonventionen für Paketnamen, bei deren Einhaltung sichergestellt ist, dass Code von unterschiedlichen Entwicklern keine gleichnamigen Pakete enthält.
- Insbesondere sollten die Namen öffentlich zur Verfügung gestellter Pakete immer mit dem (umgedrehten) Internet-Domainnamen der erstellenden Organisation beginnen, z. B. `de.htw_aalen.infotronik`.
- Innerhalb einer Organisation sollte es zusätzliche Regeln zur Bildung eindeutiger Paketnamen geben.

9.3 Import-Deklarationen

- Um Typen aus anderen Paketen mit ihren einfachen Namen anzusprechen zu können, müssen sie zuvor mit Hilfe von `import`-Deklarationen bekanntgemacht werden.

- Eine Deklaration der Art `import P.T` (mit einem vollständigen Paketnamen `P` und einem einfachen Namen `T`) importiert den öffentlichen Typ `T` aus dem Paket `P` („Einzelimport“), d. h. anschließend kann der Typ `P.T` mit seinem einfachen Namen `T` angesprochen werden.

- Eine Deklaration der Art `import P.*` (mit einem vollständigen Paketnamen `P`) importiert alle öffentlichen Typen aus dem Paket `P` („Massenimport“), d. h. anschließend können alle diese Typen mit ihren einfachen Namen angesprochen werden.

- Die Typen des vordefinierten Pakets `java.lang` müssen nicht explizit importiert werden (d. h. jede Quelldatei enthält automatisch eine Deklaration `import java.lang.*`).

- Namen aus Massenimporten können von einzeln importierten Namen sowie von Namen innerhalb des aktuellen Pakets verdeckt werden, zum Beispiel:

```
package test;
import java.sql.Date;
import java.util.*;
class List { ..... }
```

  Hier bezeichnet `Date` den Typ `java.sql.Date` (und nicht `java.util.Date`) und `List` den Typ `test.List` (und nicht `java.util.List`).

- Massenimporte sind zwar bequem zu schreiben, helfen dem Leser aber nicht bei der Suche nach importierten Namen, zum Beispiel:

```
import java.awt.*;
import javax.swing.*;
```

  Aus welchem Paket stammen die Namen `Box` und `Button`?

9.4 Quelldateien

- Am Anfang einer Quelldatei (compilation unit) kann der vollständige Name des Pakets, zu dem die in der Datei definierten Typen gehören sollen, mit Hilfe einer `package`-Deklaration vereinbart werden, z. B.:

  ```
  package de.htw_aalen.infotronik.heinlein.test;
  ```

  Wenn kein Paket vereinbart wird, gehören die Typen zu einem anonymen Paket.

- Anschließend können beliebig viele `import`-Deklarationen in beliebiger Reihenfolge angegeben werden, z. B.:

  ```
  import java.util.List;          // Einzelimport.
  import java.awt.*;              // Massenimport.
  ```

- Anschließend können beliebig viele Typen definiert werden, von denen normalerweise maximal einer öffentlich sein darf.
  Wenn ein öffentlicher Typ definiert wird, muss der Name der Quelldatei normalerweise mit dem Namen des Typs (plus Erweiterung `.java`) übereinstimmen.

- Alle o. g. Bestandteile einer Quelldatei können auch fehlen, d. h. prinzipiell könnte eine Quelldatei auch leer sein.

- Typdefinitionen können sich beliebig gegenseitig (auch zirkulär) referenzieren, auch über Paket- und Quelldateigrenzen hinweg.