Programmierzertifikat Objekt-Orientierte Programmierung mit Java

Vorlesung 04: Imperative Methoden

Peter Thiemann

Universität Freiburg, Germany

SS 2008

Inhalt

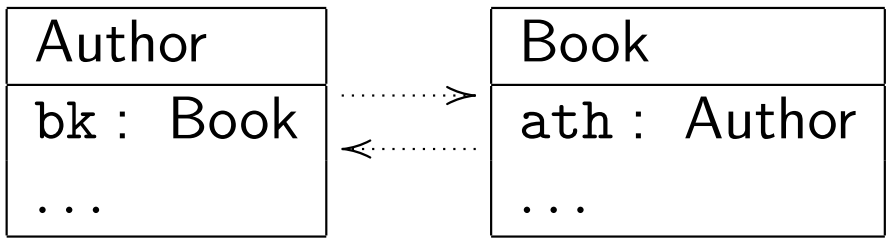
Imperative Methoden

Zirkuläre Datenstrukturen

Verwalte die Informationen über Bücher für eine Buchhandlung. Ein Buchtitel wird beschrieben durch den Titel, den Preis, die vorrätige Menge und den Autor. Ein Autor wird beschrieben durch Vor- und Nachname, das Geburtsjahr und sein Buch.

- (stark vereinfacht)
- Neue Situation:
  - Autor und Buch sind zwei unterschiedliche Konzepte.
  - Der Autor enthält sein Buch.
  - Das Buch enthält seinen Autor.

Klassendiagramm: Autor und Buch



- Frage: Wie werden Objekte von Author und Buch erzeugt?

Autoren und Bücher erzeugen

- Autor zuoberst

```
new Author ("Donald", "Knuth", 1938,
            new Book ("The Art of Computer Programming", 100, 2,
                      ????))
```

Bei ???? müsste der selbe Autor wieder eingesetzt sein. . .

Autoren und Bücher erzeugen

- Autor zuoberst

```
new Author ("Donald", "Knuth", 1938,
            new Book ("The Art of Computer Programming", 100, 2,
                      ????))
```

Bei ???? müsste der selbe Autor wieder eingesetzt sein. . .

- Buch zuoberst

```
new Book ("The Art of Computer Programming", 100, 2,
          new Author ("Donald", "Knuth", 1938,
                      ????))
```

Bei ???? müsste das selbe Buch wieder eingesetzt sein. . .

Der Wert `null`

- Lösung: Verwende `null` als Startwert für das Buch des Autors und **überschreibe** das Feld im Buch-Konstruktor.
- `null` ist ein vordefinierter Wert, der zu allen Klassen- und Interfacetypen passt. D.h., jede Variable bzw. Feld von Klassen- oder Interfacetyp kann auch `null` sein.

Autoren und Bücher wirklich erzeugen

```
// book authors
class Author {
  String fst; // first name
  String lst; // last name
  int dob; // year of birth
  Book bk;

  Author (String fst, String lst, int dob) {
    this.fst = fst;
    this.lst = lst;
    this.dob = dob;
  }
}
```

```
// Books in a library
class Book {
    String title;
    int price;
    int quantity;
    Author ath;

    Book (String title, int price,
          int quantity, Author ath) {
        this.title = title;
        this.price = price;
        this.quantity = quantity;
        this.ath = ath;

        this.ath.bk = this;
    }
}
```

Autoren und Bücher wirklich erzeugen

Verwendung der Konstruktoren

```
> Author auth = new Author("Donald", "Knuth", 1938);
> auth
Author(
    fst = "Donald",
    lst = "Knuth",
    dob = 1938,
    bk = null)
> Book book = new Book("TAOCP", 100,2, auth);
> auth
Author(
    fst = "Donald",
    lst = "Knuth",
    dob = 1938,
    bk = Book(
            title = "TAOCP",
            price = 100,
            quantity = 2,
            ath = Author))
```

Verbesserung

Fremde Felder nicht schreiben!

- Eine Methode / Konstruktor sollte nicht direkt in die Felder von Objekten fremder Klassen hereinschreiben.
- Das könnte zu illegalen Komponentenwerten in diesen Objekten führen.

$\Rightarrow$ Objekte sollten Methoden zum Setzen von Feldern bereitstellen (soweit sinnvoll).

- Konkret: Die Author-Klasse erhält eine Methode `addBook()`, die im Konstruktor von Book aufgerufen wird.

Verbesserter Code

```
// book authors
class Author {
  String fst; // first name
  String lst; // last name
  int dob; // year of birth
  Book bk = null;

  Author (String fst, String lst, int dob) {
    this.fst = fst;
    this.lst = lst;
    this.dob = dob;
  }

  void addBook (Book bk) {
    this.bk = bk;
    return;
  }
}
```

```
// Books in a library
class Book {
    String title;
    int price;
    int quantity;
    Author ath;

    Book (String title, int price,
          int quantity, Author ath) {
        this.title = title;
        this.price = price;
        this.quantity = quantity;
        this.ath = ath;

        this.ath.addBook(this);
    }
}
```

Der Typ void

- Die `addBook()` Methode hat als Rückgabetyp void.
- void als Rückgabetyp bedeutet, dass die Methode kein greifbares Ergebnis liefert und nur für ihren Effekt aufgerufen wird.
- Im Rumpf von `addBook()` steht eine *Folge von Anweisungen*. Sie werden der Reihe nach ausgeführt.
- Die letzte Anweisung **return** (ohne Argument) beendet die Ausführung der Methode.

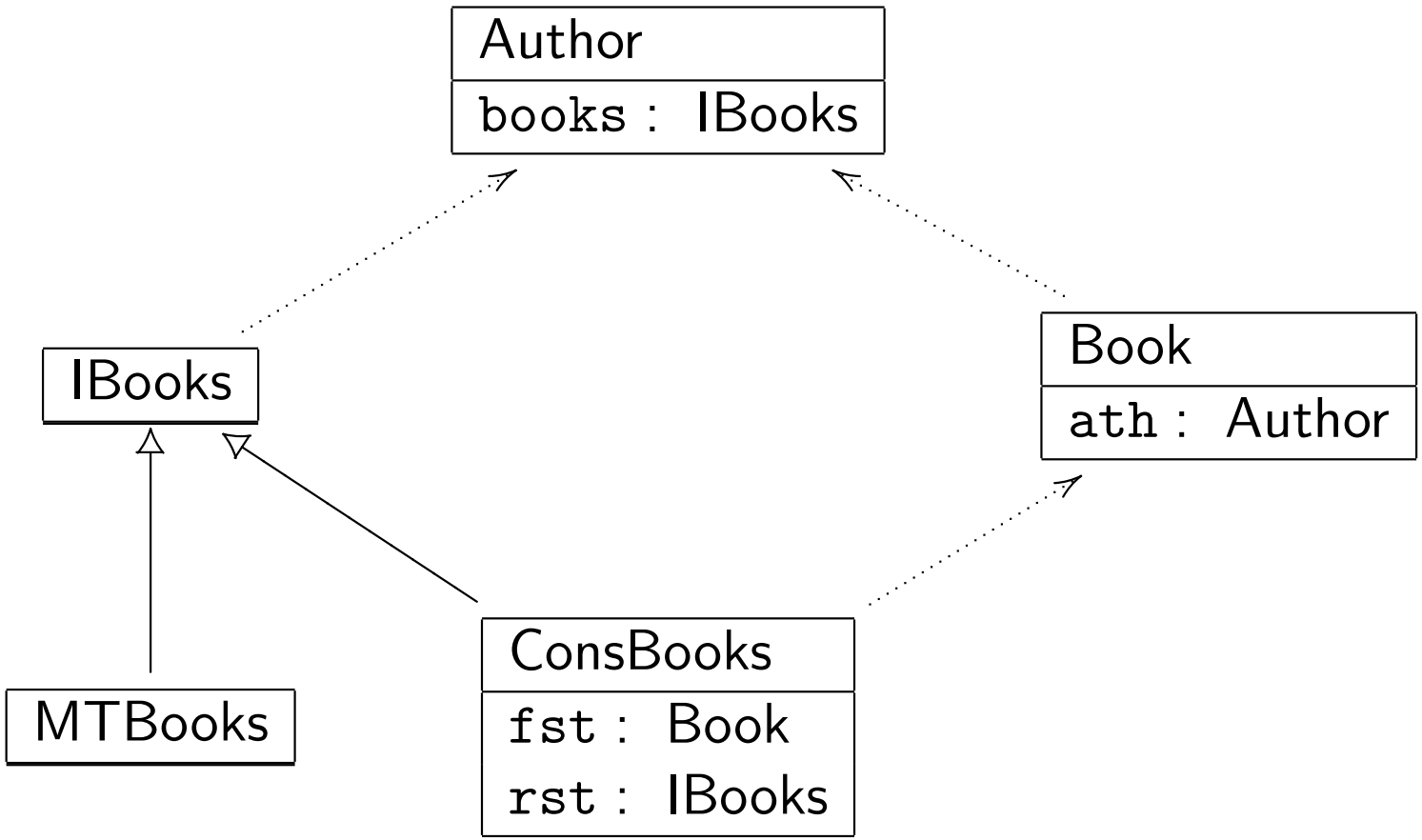
Verbesserung von addBook()

Fehlererkennung

```
void addBook (Book bk) {
  if (this.bk == null) {
    this.bk = bk;
    return;
  } else {
    Util.error("adding a second book");
  }
}
```

Ein Autor kann viele Bücher schreiben

- Ein Autor ist nun mit einer Liste von Büchern assoziiert.
- Listen von Büchern werden auf die bekannte Art und Weise repräsentiert.



Code für Autoren mit mehreren Büchern

```
// book authors
class Author {
  String fst; // first name
  String lst; // last name
  int dob; // year of birth
  IBooks books = new MTBooks();

  Author (String fst, String lst, int dob) {
    this.fst = fst;
    this.lst = lst;
    this.dob = dob;
  }

  void addBook (Book bk) {
    this.books =
      new ConsBooks (bk, this.books);
    return;
  }
}
```

```
// Listen von Büchern
interface IBooks { }
```

```
class MTBooks implements IBooks {
    MTBooks () {}
}
```

```
class ConsBooks implements IBooks {
    Book fst;
    IBooks rst;

    ConsBooks (Book fst, IBooks rst) {
        this.fst = fst;
        this.rst = rst;
    }
}
```

Zusammenfassung

Entwurf von Klassen mit zirkulären Objekten

1. Bei der Datenanalyse stellt sich heraus, dass (mindestens) zwei Objekte wechselseitig ineinander enthalten sein sollten.
2. Bei der Erstellung des Klassendiagramms gibt es einen Zyklus bei den Enthaltenseins-Pfeilen. Dieser Zyklus muss nicht offensichtlich sein, z.B. kann ein Generalisierungspfeil rückwärts durchlaufen werden.
3. Die Übersetzung in Klassendefinitionen funktioniert mechanisch.
4. Wenn zirkuläre Abhängigkeiten vorhanden sind:
   - Können tatsächlich zirkuläre Beispiele erzeugt werden?
   - Welche Klasse C ist als Startklasse sinnvoll und über welches Feld `fz` von C läuft die Zirkularität?
   - Initialisiere das `fz` Feld mit einem Objekt, das keine Werte vom Typ C enthält (notfalls müssen Felder des Objekts mit `null` besetzt werden).
   - Definiere eine `add()` Methode, die `fz` passend abändert.
   - Ändere die Konstruktoren, so dass sie `add()` aufrufen.
5. Codiere die zirkulären Beispiele.

Die Wahrheit über Konstruktoren

- Die **new**-Operation erzeugt neue Objekte.
- Zunächst sind alle Felder mit `0` (Typ int), `false` (Typ boolean), `0.0` (Typ double) oder `null` (Klassen- oder Interfacetyp) vorbesetzt.
- Der Konstruktor weist den Feldern Werte zu und kann weitere Operationen ausführen.
- Die Initialisierung kann merkwürdige Effekte haben, da manche Feldinitialisierungen ablaufen, **bevor** der Konstruktor ausgeführt wird.

Merkwürdige Initialisierung

```
class StrangeExample {
    int x;

    StrangeExample () { this.x = 100; }

    boolean test = check this.x expect 100;
}
```

- Was sind die Werte von `this.x` und `this.test` nach Konstruktion des Objekts?

Merkwürdige Initialisierung

```
class StrangeExample {
    int x;

    StrangeExample () { this.x = 100; }

    boolean test = check this.x expect 100;
}
```

- Was sind die Werte von `this.x` und `this.test` nach Konstruktion des Objekts?
- `this.x` = 100 `this.test` = `false`

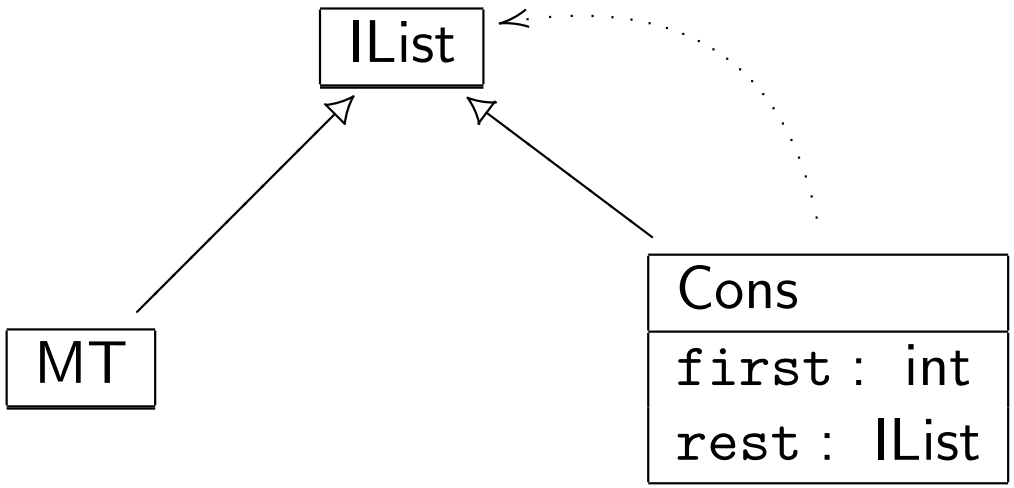
Merkwürdige Initialisierung

```
class StrangeExample {
    int x;

    StrangeExample () { this.x = 100; }

    boolean test = check this.x expect 100;
}
```

- Was sind die Werte von `this.x` und `this.test` nach Konstruktion des Objekts?
- `this.x` = 100 `this.test` = `false`
- Ablauf:
  - Erst werden alle Felder vorbesetzt.
  - Dann laufen alle Feldinitialisierungen ab.
  - Zuletzt wird der Rumpf des Konstruktors ausgeführt.

Zyklische Listen

- Jeder Listendatentyp enthält zyklische Referenzen im Klassendiagramm.



- Also müssen auch damit zyklische Strukturen erstellbar sein!

Zyklische Listen erstellen

```
class CyclicList {
  Cons alist = new Cons (1, new MT ());

  Example () {
    this.alist.rest = this.alist;
  }
}
```

- Aufgabe: Erstelle eine Methode `length()` für IList, die die Anzahl der Elemente einer Liste bestimmt. Was liefert

```
new Example ().alist.length()
```

als Ergebnis? Warum?

Vermeiden von unerwünschter Zirkularität

Durch Geheimhaltung

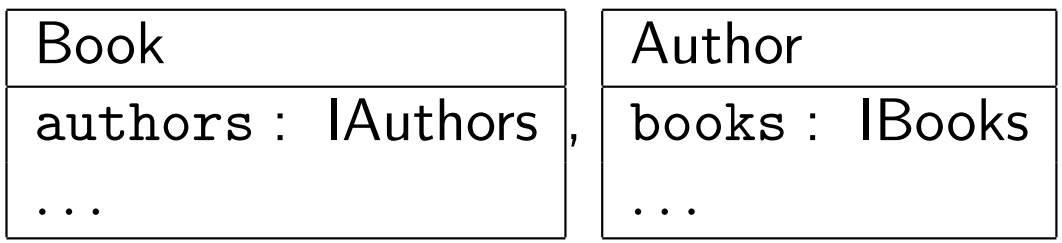
```
class Cons implements IList {
  private int first;
  private IList rest;

  public Cons (int first, IList rest) {
    this.first = first;
    this.rest = rest;
  }
}
```

- Mit dieser Definition ist es unmöglich, das `rest`-Feld zu überschreiben.

Viele Autoren und viele Bücher

Verwalte die Informationen über Bücher für eine Buchhandlung. Ein Buchtitel wird beschrieben durch den Titel, den Preis, die vorrätige Menge und die Autoren. Ein Autor wird beschrieben durch Vor- und Nachname, das Geburtsjahr und seine Bücher.

Beteiligte Klassen

- Listen von Büchern: IBooks, MTBooks, ConsBooks
- Listen von Autoren: IAuthors, MTAuthors, ConsAuthors
- 

Code für viele Autoren und viele Bücher

```
// book authors
class Author {
  String fst; // first name
  String lst; // last name
  int dob; // year of birth
  IBooks books = new MTBooks();

  Author (String fst, String lst, int dob) {
    this.fst = fst;
    this.lst = lst;
    this.dob = dob;
  }

  void addBook (Book bk) {
    this.books =
      new ConsBooks (bk, this.books);
    return;
  }
}
```

```
// Books in a library
class Book {
  String title;
  int price;
  int quantity;
  IAuthors authors;

  Book (String title, int price,
        int quantity, IAuthors authors) {
    this.title = title;
    this.price = price;
    this.quantity = quantity;
    this.authors = authors;

    this.authors.????(this);
  }
}
```

Hilfsmethode für Konstruktor

- **Interessant**: Implementierung des Book Konstruktors erfordert den Entwurf einer nichttrivialen Methode für IAuthors!
- Gesucht: Methode zum Hinzufügen des neuen Buchs zu **allen** Autoren.
- Methodensignatur im Interface IAuthors

```
// Autorenliste
interface IAuthors {
    // füge das Buch zu alle Autoren auf dieser Liste hinzu
    void addBookToAll(Book bk);
}
```

$\Rightarrow$ Die Methode liefert kein Ergebnis.

- Einbindung in den Konstruktor von Book durch

```
    this.authors.addBookToAll(this);
```

Implementierung der Hilfsmethode

```
class MTAuthors
  implements IAuthors {
  MTAuthors () {}

  void addBookToAll (Book bk) {
    return;
  }
}
```

```
class ConsAuthors
  implements IAuthors {
  Author first;
  IAuthors rest;

  ConsAuthors (Author first, IAuthors rest) {
    this.first = first;
    this.rest = rest;
  }

  void addBookToAll (Book bk) {
    this.first.addBook (bk);
    this.rest.addBookToAll (bk);
    return;
  }
}
```

Zuweisungen und Zustand

Zuweisungen und Zustand

- In Java steht der (Infix-) Operator = **immer** für eine *Zuweisung* (an ein Feld oder eine Variable).
- Eine Methode mit Ergebnistyp void liefert kein Ergebnis, sondern erzielt nur einen *Effekt*.
- Die Anweisungsfolge

  Anweisung1; *Anweisung2*;

  bedeutet, dass zuerst *Anweisung1* ausgeführt wird und danach *Anweisung2*. Ein etwaiges Ergebnis wird dabei ignoriert.
- Die Werte in allen Instanzvariablen können sich ändern.

Beispiel: Bankkonto

Entwerfe eine Repräsentation für ein Bankkonto. Das Bankkonto soll drei typische Aufgaben erledigen: Geld einzahlen, Geld abheben und Kontostand abfragen. Jedes Bankkonto gehört einer Person.

Bankkonto

Funktionaler Ansatz

- Eine Account Klasse mit zwei Feldern, dem Kontostand und dem Kontoinhaber, ist erforderlich. Die anfängliche Einlage sollte größer als 0 sein.
- Die Klasse benötigt mindestens drei **public** Methoden
  - einzahlen: `Account deposit (int a)`
  - abheben: `Account withdraw (int a)`
  - Kontostand: `String balance()`

  In allen Fällen muss `a > 0` und der Abhebebetrag sollte kleiner gleich dem Kontostand sein.

Bankkonto

Imperativer Ansatz

- Eine Account Klasse mit zwei Feldern, dem Kontostand und dem Kontoinhaber, ist erforderlich. Die anfängliche Einlage sollte größer als 0 sein.
- Die Klasse benötigt mindestens drei **public** Methoden
  - einzahlen: `void deposit (int a)`
  - abheben: `void withdraw (int a)`
  - Kontostand: `String balance()`

  In allen Fällen muss `a > 0` und der Abhebebetrag sollte kleiner gleich dem Kontostand sein.

Implementierungen des Bankkontos im Vergleich

```
// Bankkonto funktional
class Account {
  int amount;
  String holder;

  Account (int amount, String holder) {
    this.amount = amount;
    this.holder = holder;
  }

  Account deposit (int a) {
    return new Account
         (this.amount + a, this.holder);
  }

  Account withdraw (int a) {
    return new Account
         (this.amount − a, this.holder);
  }
```

```
// Bankkonto imperativ
class Account {
  int amount;
  String holder;

  Account (int amount, String holder) {
    this.amount = amount;
    this.holder = holder;
  }

  void deposit (int a) {
    this.amount = this.amount + a;
    return;
  }

  Account withdraw (int a) {
    this.amount = this.amount − a;
    return;
  }
```

Implementierung der `balance()` Methode

- Die `balance()` Methode ist in beiden Implementierungen gleich.

```
  String balance() {
    return this.holder.concat (": ".concat (String.valueOf (this.amount)));
  }
}
```

Beispiel: Animation mit dem `idraw` Paket

```
// animierte Welt mit Grafikausgabe
abstract class World {
  Canvas theCanvas = new Canvas();

  // öffne Zeichenfläche, starte die Uhr
  void bigBang(int w, int h, double s) {...}

  // ein Uhrticken verarbeiten
  abstract void onTick();

  // einen Tastendruck verarbeiten
  abstract void onKeyEvent (String ke);

  // zeichne die Welt
  abstract void draw ();

  // stoppe die Welt
  public World endOfWorld (String s) {...}
}
```

```
// Kontrollieren einer Zeichenfläche
class Canvas {
  int width; int height;

  // Anzeigen der Zeichenfläche
  void show();

  // Kreis zeichnen
  void drawCircle(Posn p, int r, IColor c);

  // Scheibe zeichnen
  void drawDisk(Posn p, int r, IColor c);

  // Rechteck zeichnen
  void drawRect(Posn p, int w, int h, IColor c

  // String zeichnen
  void drawString(Posn p, String s);
}
```

Aufgabe: Fallender Block

Bei der Animation "Fallender Block" erscheint am oberen Rand des Bildschirms auf Position (10,20) jeweils ein Block erscheint, der mit einem Pixel pro Sekunde bis auf den unteren Rand der Zeichenfläche fällt.

- Die Implementierung soll die animierte Welt World verwenden. Im Vorspann der Klassen (noch vor `class`) muss stehen:
  - `import idraw.*;`
  - `import geometry.*;`
  - `import colors.*;`

$\Rightarrow$ Es wird also eine Subklasse BlockWorld von World benötigt!

- Alle Methoden in BlockWorld werden durch die abstrakte Superklasse World erzwungen.

Entwurf der Welt der fallenden Blöcke

```
class BlockWorld extends World {
    private DrpBlock block; // modelliert den fallenden Block

    public BlockWorld () {...}
    // Zeichnen der Welt
    public void draw () {
        this.block.draw(this.theCanvas);
        return;
    }
    // Änderung der Welt; Effekt: Ändert das Feld block
    public void onTick () {
        this.block.drop();
        return;
    }

    public void onKeyEvent (String ke) {
        return; // nichts zu tun
    }
}
```

Fallender Block, Imperative Version

```
class DrpBlock {
    private int x;
    private int y;
    private int SIZE = 10;
    public DrpBlock () {
        this.x = 10;
        this.y = 20;
    }

    public void drop() {
        this.y = this.y + 1;
        return;
    }

    public boolean isAt (int h) {
        return this.y >= h;
    }

    public void draw (Canvas c);
}
```

Vergleichen von Objekten

Statischer Typ vs dynamischer Typ

- Der *statische Typ* (kurz: Typ) eines Ausdrucks ist der Typ, den Java für den Ausdruck aus dem Programmtext ausrechnet.
- Der *dynamische Typ* (*Laufzeittyp*) ist eine Eigenschaft eines Objekts. Es ist der Klassenname, mit dem das Objekt erzeugt worden ist.

Statischer Typ vs dynamischer Typ

- Der *statische Typ* (kurz: Typ) eines Ausdrucks ist der Typ, den Java für den Ausdruck aus dem Programmtext ausrechnet.
- Der *dynamische Typ* (*Laufzeittyp*) ist eine Eigenschaft eines Objekts. Es ist der Klassenname, mit dem das Objekt erzeugt worden ist.

Beispiele

- Angenommen A **extends** B (Klassentypen).

```
A a = new A (); // rhs: Typ A, dynamischer Typ A
B b = new B (); // rhs: Typ B, dynamischer Typ B
B x = new A (); // rhs: Typ A, dynamischer Typ A
// für x gilt: Typ B, dynamischer Typ A
```

- Bei einem Interfacetyp ist der dynamische Typ **immer** ein Subtyp.
- Im Rumpf einer Methode definiert in der Klasse C hat `this` den statischen Typ C. Der dynamische Typ kann ein Subtyp von C sein, falls die Methode vererbt worden ist.

Regeln für die Bestimmung des statischen Typs

- Falls Variable (Feld, Parameter) `x` durch ttt x deklariert ist, so ist der Typ von `x` genau ttt.
- Der Ausdruck `new C(...)` hat den Typ C.
- Wenn e ein Ausdruck vom Typ C ist und C eine Klasse mit Feld f vom Typ ttt ist, dann hat $e.f$ den Typ ttt.
- Wenn e ein Ausdruck vom Typ C ist und C eine Klasse oder Interface mit Methode m vom Rückgabetyp ttt ist, dann hat $e.m(\dots)$ den Typ ttt.
- Beim Aufruf eines Konstruktors oder einer Funktion müssen die Typen der Argumente jeweils Subtypen der Parametertypen sein.
- Bei einer Zuweisung muss der Typ des Audrucks auf der rechten Seiten ein Subtyp des Typs der Variable (Feld) sein.

Vergleichen von Objekten

Beispiel: Daten

```
class DateComparison {
  Date d1 = new Date(27,3,1941);
  Date d2 = new Date(8,5,1945);
  Date d3 = new Date(8,5,1945);

  boolean testD1D2 = d1 == d2; // Operator == auf Objekten
  boolean testD2D3 = d2 == d3;
  boolean testD3D3 = d3 == d3;
}
```

Vergleichen von Objekten

Beispiel: Daten

```
class DateComparison {
  Date d1 = new Date(27,3,1941);
  Date d2 = new Date(8,5,1945);
  Date d3 = new Date(8,5,1945);

  boolean testD1D2 = d1 == d2; // Operator == auf Objekten
  boolean testD2D3 = d2 == d3;
  boolean testD3D3 = d3 == d3;
}
```

Ergebnis

```
DateComparison(
   d1 = Date(...), d2 = Date(...), d3 = Date(...),
   testD1D2 = false,
   testD2D3 = false,
   testD3D3 = true)
```

Verschiedene Gleichheitsoperationen

- Der Gleichheitsoperator == ist auch auf Objekte anwendbar, aber liefert nicht das erwartete(?) Ergebnis!
- Er testet, ob beide Argument *dasselbe* Objekt bezeichnen.
- Oft ist **komponentenweise** Gleichheit gewünscht (*extensionale Gleichheit*).
- Muss programmiert werden (oft als `equals()` Methode, vgl. String).

Gleichheit für einfache Klassen

- Einfache Klassen enthalten Felder von primitivem Typ.
- Ihre Werte können mit == verglichen werden (bzw. mit `equals()` für String).
- Beispiel: Vergleichsmethode `boolean same(Date that)` für die Date-Klasse.

```
// is this date the same as that date?
boolean same (Date that) {
  return (this.day == that.day) &&
         (this.month == that.month) &&
         (this.year == that.year);
}
```

Gleichheit für Mengen

Betrachte eine Klasse Set2, bei der jedes Objekt eine Menge von int mit *genau zwei Elementen* repräsentiert.

```
// Mengen von genau zwei Zahlen
class Set2 {
    private int one;
    private int two;

    public Set2 (int one, int two) { ... }

    // Elementtest
    public boolean contains (int x) {
        return (x == this.one) || (x == this.two);
    }
}
```

Gleichheit für Mengen

Implementierung

- Komponentenweise Gleichheit nicht angemessen.
- Zwei Mengen sind gleich, wenn sie sich gegenseitig enthalten

$$\{16, 42\} = \{42, 16\}$$

- `same()` Methode in Set2:

```
// Gleichheitstest
public boolean same (Set2 that) {
    return (this.contains (that.one))
        && (this.contains (that.two))
        && (that.contains (this.one))
        && (that.contains (this.two));
}
```

Gleichheit und Vererbung

```
class SpecialDate extends Date {
    private int rating;

    SpecialDate (int day, int month, int year, int rating) {
        super (day, month, year);
        this.rating = rating;
    }
}
```

- Spezielle Daten könnten mit einer Bewertung versehen sein.
- Die `same()` Methode aus der Klasse Date ist anwendbar, allerdings liefert sie nicht die erwarteten Ergebnisse.

```
class DateTest {
    SpecialDate s1 = new SpecialDate (12,8,2001,4000);
    SpecialDate s2 = new SpecialDate (12,8,2001,5000);

    boolean testss1 = s1.same (s1); // ==> true !!!
    boolean testss2 = s1.same (s2); // ==> true ???
}
```

Gleichheit und Vererbung

same-Methode in der Subklasse

- Eine spezialisierte Version der `same()` Methoden in der Subklasse ist erforderlich

```
boolean same (SpecialDate that) {
    return super.same (that) && (this.rating == that.rating);
}
```

- Damit funktioniert das Beispiel

```
class DateTest { // mit same(SpecialDate) in Klasse SpecialDate
    SpecialDate s1 = new SpecialDate (12,8,2001,4000);
    SpecialDate s2 = new SpecialDate (12,8,2001,5000);

    boolean testss1 = s1.same (s1); // ==> true !!!
    boolean testss2 = s1.same (s2); // ==> false !!!
}
```

Gleichheit und Vererbung

Weitere Probleme

- Andere Beispiele funktionieren nicht wie erwartet.
- `same`-Gleichheit ist nicht transitiv!

```
class DateTest { // mit same(SpecialDate) in Klasse SpecialDate
    SpecialDate s1 = new SpecialDate (12,8,2001,4000);
    SpecialDate s2 = new SpecialDate (12,8,2001,5000);
    Date d2 = new Date (12,8,2001);

    boolean testsd = s1.same (d2); // ==> true ???
    boolean testds = d2.same (s2); // ==> true ???
    boolean testss = s1.same (s2); // ==> false !!!
}
```

Gleichheit und Vererbung

Weitere Probleme

- Andere Beispiele funktionieren nicht wie erwartet.
- `same`-Gleichheit ist nicht transitiv!

```
class DateTest { // mit same(SpecialDate) in Klasse SpecialDate
    SpecialDate s1 = new SpecialDate (12,8,2001,4000);
    SpecialDate s2 = new SpecialDate (12,8,2001,5000);
    Date d2 = new Date (12,8,2001);

    boolean testsd = s1.same (d2); // ==> true ???
    boolean testds = d2.same (s2); // ==> true ???
    boolean testss = s1.same (s2); // ==> false !!!
}
```

Überraschung

- In den ersten beiden Fällen wird die Methode `same` der Klasse Date aufgerufen!
- Ursache: *Überladung* von Methoden.

Überladung von Methoden

- Überladung: in einer Klasse gibt es mehrere Methoden mit gleichem Namen, die sich nur in Anzahl oder Typ der Parameter unterscheiden.
- Die Auswahl der tatsächlich aufgerufenen Methode erfolgt durch Java aufgrund des ermittelten Argumenttyps.

Überladung von Methoden

- Überladung: in einer Klasse gibt es mehrere Methoden mit gleichem Namen, die sich nur in Anzahl oder Typ der Parameter unterscheiden.
- Die Auswahl der tatsächlich aufgerufenen Methode erfolgt durch Java aufgrund des ermittelten Argumenttyps.

Beispiel

- In der Klasse SpecialDate gibt es **zwei** Methoden mit Namen `same`, **die sich nur im Parametertyp unterscheiden**:
  1. `boolean same (Date that)` (geerbt von Date)
  2. `boolean same (SpecialDate that)` (selbst definiert)
- In `testsd` wird #1, die geerbte Methode, aufgerufen, da `d2` den Typ Date hat.
- In `testds` wird auch #1 aufgerufen, da das Empfängerobjekt den Typ Date hat.

Transitive Gleichheit

- Zufriedenstellende Implementierung benötigt **zwei** Methoden!
- Schwierigkeit: Feststellen, ob das Argumentobjekt den gleichen *dynamischen Typ* wie das Empfängerobjekt hat.
- Die Methode `same (Date that)` muss in Date definiert sein und in allen Subklassen von Date überschrieben werden.
- Sie stellt lediglich fest, welchen *dynamischen Typ* das Empfängerobjekt zur Laufzeit hat.
- Dann testet sie mit dem **instanceof**-Operator, ob das Argumentobjekt zu einer Subklasse dieses dynamischen Typs gehört.
- Die Hilfsmethode `reallysame (Date that)` führt denselben Test in **umgekehrter** Richtung aus, wobei schon sichergestellt ist, dass das Argumentobjekt zu einer Superklasse des Empfängertyps gehört.
- Nun sind die dynamischen Typen gleich und die Felder können verglichen werden. Die Felder von `that` müssen zunächst durch einen *Typcast* sichtbar gemacht werden.

Transitive Gleichheit (Implementierung)

Basisfall

```
class Date {
  private int day;
  private int month;
  private int year;

  protected boolean reallysame (Date that) {
    return (this.day == that.day) &&
           (this.month == that.month) &&
           (this.year == that.year);
  }

  public boolean same (Date that) {
    return that.reallysame (this);
  }
}
```

Code für Subklassen

```
class SpecialDate extends Date {
  private int rating;

  SpecialDate (int day, int month, int year, int rating) {
    super (day, month, year);
    this.rating = rating;
  }
  // dynamic type of that is a supertype of type of this
  protected boolean reallysame (Date that) {
    return (that instanceof SpecialDate)
        && super.reallysame (that)
        && (this.rating == ((SpecialDate)that).rating);
  }

  public boolean same (Date that) {
    return (that instanceof SpecialDate)
        && that.reallysame (this);
  }
}
```

Der **instanceof**-Operator

- Der boolesche Ausdruck

  ausdruck **instanceof** *objekttyp*

  testet ob der dynamische Typ des Werts von *ausdruck* ein Subtyp von *objekttyp* ist.
- Angenommen A **extends** B (Klassentypen):

```
A a = new A();
B b = new B();
B c = new A(); // statischer Typ B, dynamischer Typ A

a instanceof A // ==> true
a instanceof B // ==> true
b instanceof A // ==> false
b instanceof B // ==> true
c instanceof A // ==> true (testet den dynamischen Typ)
c instanceof B // ==> true
```

Der Typcast-Operator

- Der Ausdruck (*Typcast*)

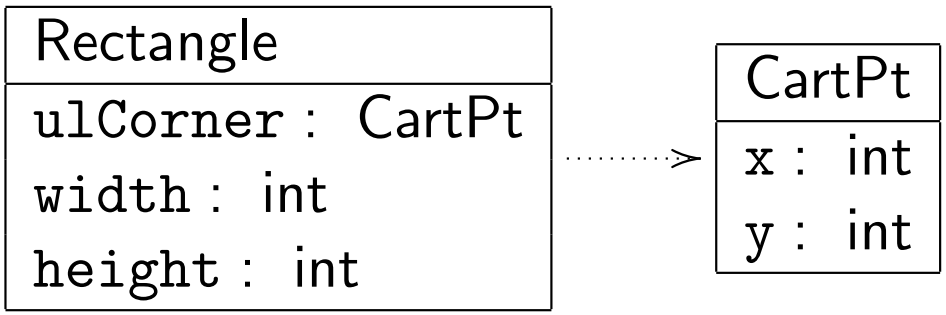
  (*objekttyp*) *ausdruck*

  hat den statischen Typ *objekttyp*, falls der statische Typ von *ausdruck* entweder ein Supertyp oder ein Subtyp von *objekttyp* ist.
- Zur Laufzeit testet der Typcast, ob der **dynamische Typ** des Werts von *ausdruck* ein Subtyp von *objekttyp* ist und bricht das Programm ab, falls das nicht zutrifft. (Vorher sicherstellen!)
- Angenommen A **extends** C und B **extends** C (Klassentypen), aber A und B stehen in keiner Beziehung zueinander:

```
A a = new A(); B b = new B(); C c = new C(); C d = new A();

(A)a // statisch ok, dynamisch ok
(B)a // Typfehler
(C)a // statisch ok, dynamisch ok
(B)d // statisch ok, dynamischer Fehler
(A)d // statisch ok, dynamisch ok
```

Gleichheit für zusammengesetzte Objekte

Beispiel




```
boolean same (Rectangle that) {
  return (this.x == that.x) && (this.y == that.y)
      && (this.ulCorner.same (that.ulCorner));
}
```

- Rufe die Gleichheit auf den untergeordneten Objekten auf.

Gleichheit für Vereinigungen von Klassen

- Definiere die `same()` Methode im Interface.
- Verwende die Vorgehensweise für Vererbung.
- Für abstrakte Klassen reicht es, die Methoden `same` und `reallysame` abstrakt zu belassen (da niemals Objekte existieren können, die diesen Klassentyp als Laufzeittyp besitzen).

Alternative Lösung

Ohne Verwendung von **instanceof** und Typcast

Am Beispiel von IShape:

- Voraussetzung: alle Varianten sind bekannt.
- Erweitere das Interface um Methoden, die die jeweilige Variante erkennen und ggf. ein IShape Objekt in ein Objekt vom spezifischen Typ umwandeln. Die Methoden liefern `null`, falls die Umwandlung nicht möglich ist.

```
interface IShape {
  Dot toDot();
  Square toSquare();
  Circle toCircle();

  boolean same (IShape that);
}
```

Alternative Lösung

Die abstrakte Klasse

```
abstract class AShape implements IShape {
  Dot toDot () { return null; }
  Square toSquare () { return null; }
  Circle toCircle () { return null; }

  abstract boolean same (IShape that);
}
```

Alternative Lösung

Implementierung für Dot

```
class Dot implements IShape {
  Dot toDot () { return this; }

  boolean same (IShape that) {
    Dot thatDot = that.toDot();
    return (thatDot != null)
        && (this.loc.same (thatDot.loc));
  }
}
```

Intensionale Gleichheit

- **Extensionale Gleichheit** testet ob zwei Objekte gleich sind und sich gleich verhalten.
- Diese Aufgabe hat in Java die `equals` Methode.
- **Intensionale Gleichheit** testet ob ihre Argumente dasselbe Objekt bezeichnen, in dem Sinn, dass eine Änderung am einen Argument immer die selbe Änderung am anderen Argument bewirkt.
- Diese Aufgabe hat in Java der == Operator. (Er testet die Gleichheit von Referenzen.)
- In Java vordefiniert:

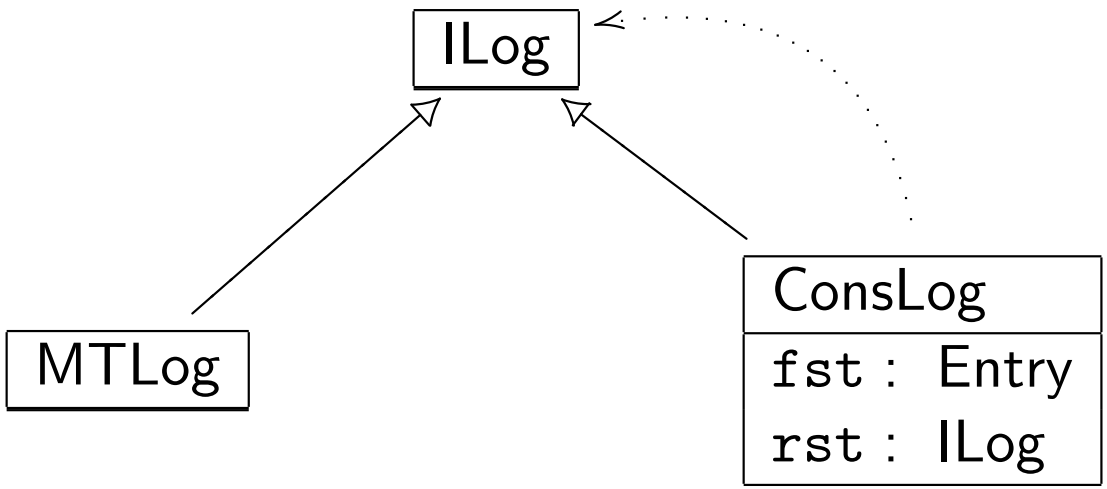
```
class Object {
  public boolean equals (Object other) {
    return this == other;
  }
}
```

- Jede Klasse ist automatisch Subklasse von Object und erbt diese (meist so nicht gewünschte) Implementierung.

Iteration

Iteration

Erinnerung: das Lauftagebuch



- Ziel: Definiere Methoden auf ILog auf andere Art
- Betrachte Methoden `totalDistance()` und `length()`.

Implementierung von `totalDistance()`

- in ILog

```
// berechne die Gesamtkilometerzahl
double totalDistance();
```

- in MTLog

```
double totalDistance() {
  return 0;
}
```

- in ConsLog

```
double totalDistance() {
  return this.fst.distance + this.rst.totalDistance();
}
```

Implementierung von `length()` ist sehr ähnlich

- in ILog

```
// berechne die Gesamtkilometerzahl
int length();
```

- in MTLog

```
int length() {
  return 0;
}
```

- in ConsLog

```
int length() {
  return 1 + this.rst.length();
}
```

Problem mit den Implementierungen

- Bei sehr langen Listen erfolgt ein “Stackoverflow”, weil die maximal mögliche Schachtelungstiefe von rekursiven Aufrufen überschritten wird.
- Erster Schritt: Führe einen **Akkumulator** (extra Parameter, in dem das Ergebnis angesammelt wird) ein und mache die Methoden endrekursiv.

Alternative Implementierung von `totalDistance()`

Mit Akkumulator

- in ILog

```
// berechne die Gesamtkilometerzahl
double totalDistanceAcc(double acc);
```

- in MTLog

```
double totalDistanceAcc(double acc) {
  return acc;
}
```

- in ConsLog

```
double totalDistanceAcc(double acc) {
  return this.rst.totalDistanceAcc (acc + this.fst.distance);
}
```

- Aufruf

```
double myDistance = log.totalDistanceAcc (0);
```

Implementierung von `lengthAcc()` ist sehr ähnlich

- in ILog

```
// berechne die Gesamtkilometerzahl
int lengthAcc(int acc);
```

- in MTLog

```
int lengthAcc(int acc) {
  return acc;
}
```

- in ConsLog

```
int lengthAcc(int acc) {
  return this.rst.lengthAcc (acc + 1);
}
```

- Aufruf

```
int myLength = log.lengthAcc (0);
```

Gewonnen?

- Die endrekursiven Versionen der Methoden mit Akkumulator *könnten* in konstanten Platz implementiert werden.
- Aber Java (bzw. die Java Virtual Machine, JVM) tut das nicht.
- Abhilfe: Durchlaufe die Liste imperativ mit einer **while**-Schleife.

Die **while**-Anweisung

- Allgemeine Form

```
while (bedingung) {
  anweisungen;
}
```

- *bedingung* ist ein boolescher Ausdruck.
- Bei der Ausführung der **while**-Anweisung wird zuerst die *bedingung* getestet.
- Ist sie `false`, so ist die Ausführung der **while**-Anweisung beendet. Ist sie `true`, so werden die *anweisungen* ausgeführt und danach wieder die *bedingung* getestet.
- Der letzte Schritt wird solange wiederholt, bis die Ausführung der **while**-Anweisung beendet ist.

Interface für Listendurchlauf

Problem

Die Codefragmente für die **while**-Anweisung sind über die beiden Klassen MTLog und ConsLog verstreut.

Abhilfe

Definiere Interface für das Durchlaufen der ILog Liste, so dass die Codefragmente an einer Stelle zusammenkommen.

```
interface ILog {
  ...
  // teste ob diese Liste leer ist
  boolean isEmpty();
  // liefere das erste Element, falls nicht leer
  Entry getFirst();
  // liefere den Rest der Liste, falls nicht leer
  ILog getRest();
  ...
}
```

Implementierung des Interface für Listendurchlauf

- in MTLog

```
boolean isEmpty () { return true; }
Entry getFirst () { return null; }
ILog getRest () { return null; }
```

- in ConsLog

```
boolean isEmpty () { return false; }
Entry getFirst () { return this.fst; }
ILog getRest () { return this.rst; }
```

Verwendung des Interface für Listendurchlauf

- In (neuer) Superklasse ALog von MTLog und ConsLog

```
int length () {
  return lengthAux (0, this);
}
```

- In beliebiger Hilfsklasse (z.B. in ALog)

```
int lengthAux (int acc, ILog list) {
  if (list.isEmpty()) {
    return acc;
  } else {
    return lengthAux (acc + 1, list.getRest());
  }
}
```

Verwendung des Interface für Listendurchlauf

- In (neuer) Superklasse ALog von MTLog und ConsLog

```
int length () {
  return lengthAux (0, this);
}
```

- In beliebiger Hilfsklasse (z.B. in ALog)

```
int lengthAux (int acc, ILog list) {
  if (list.isEmpty()) {
    return acc;
  } else {
    return lengthAux (acc + 1, list.getRest());
  }
}
```

- Diese Methode kann sofort in eine **while**-Anweisung umgesetzt werden!
  - Aus den Parametern werden lokale Variablen.
  - Aus dem rekursiven Aufruf werden Zuweisungen auf diese Variablen.

Imperative Version von `lengthAux`

```
int lengthAux (int acc0, ILog list0) {
  int acc = acc0;
  ILog list = list0;
  while (!list.isEmpty()) {
    acc = acc + 1;
    list = list.getRest();
  }
  return acc;
}
```

Aufruf aus ALog

```
int length () {
  return lengthAux (0, this);
}
```

Mit Hilfe von *Inlining* kann der Aufruf von `lengthAux` eliminiert werden. (D.h., ersetze den Aufruf durch seine Definition.)

Alles in der `length` Methode

```
int length () {
  int acc = 0;
  ILog list = this;
  while (!list.isEmpty()) {
    acc = acc + 1;
    list = list.getRest();
  }
  return acc;
}
```

- Läuft in konstantem Platz.
- Verarbeitet beliebig lange Listen.

Implementierung von `totalDistance` mit Durchlaufinterface

. . . nach dem gleichen Schema wie `length`!

- In Superklasse ALog von MTLog und ConsLog

```
double totalDistance () {
  return totalDistanceAux (0, this);
}
```

- In beliebiger Hilfsklasse (z.B. in ALog)

```
double totalDistanceAux (double acc, ILog list) {
  if (list.isEmpty()) {
    return acc;
  } else {
    Entry e = list.getFirst(); // Zugriff aufs Listenelement
    return totalDistanceAux (acc + e.distance, list.getRest());
  }
}
```

Imperative Implementierung von `totalDistanceAux`

```
double totalDistanceAux (double acc0, ILog list0) {
  double acc = acc0;
  ILog list = list0;
  while (!list.isEmpty()) {
    Entry e = list.getFirst(); // Zugriff aufs Listenelement
    acc = acc + e.distance;
    list = list.getRest();
  }
  return acc;
}
```

- Wende inlining an. . .

Imperative Implementierung von `totalDistance`

```
double totalDistance () {
  double acc = 0;
  ILog list = this;
  while (!list.isEmpty()) {
    Entry e = list.getFirst(); // Zugriff aufs Listenelement
    acc = acc + e.distance;
    list = list.getRest();
  }
  return acc;
}
```

Veränderliche rekursive Datenstrukturen

Veränderliche rekursive Datenstrukturen

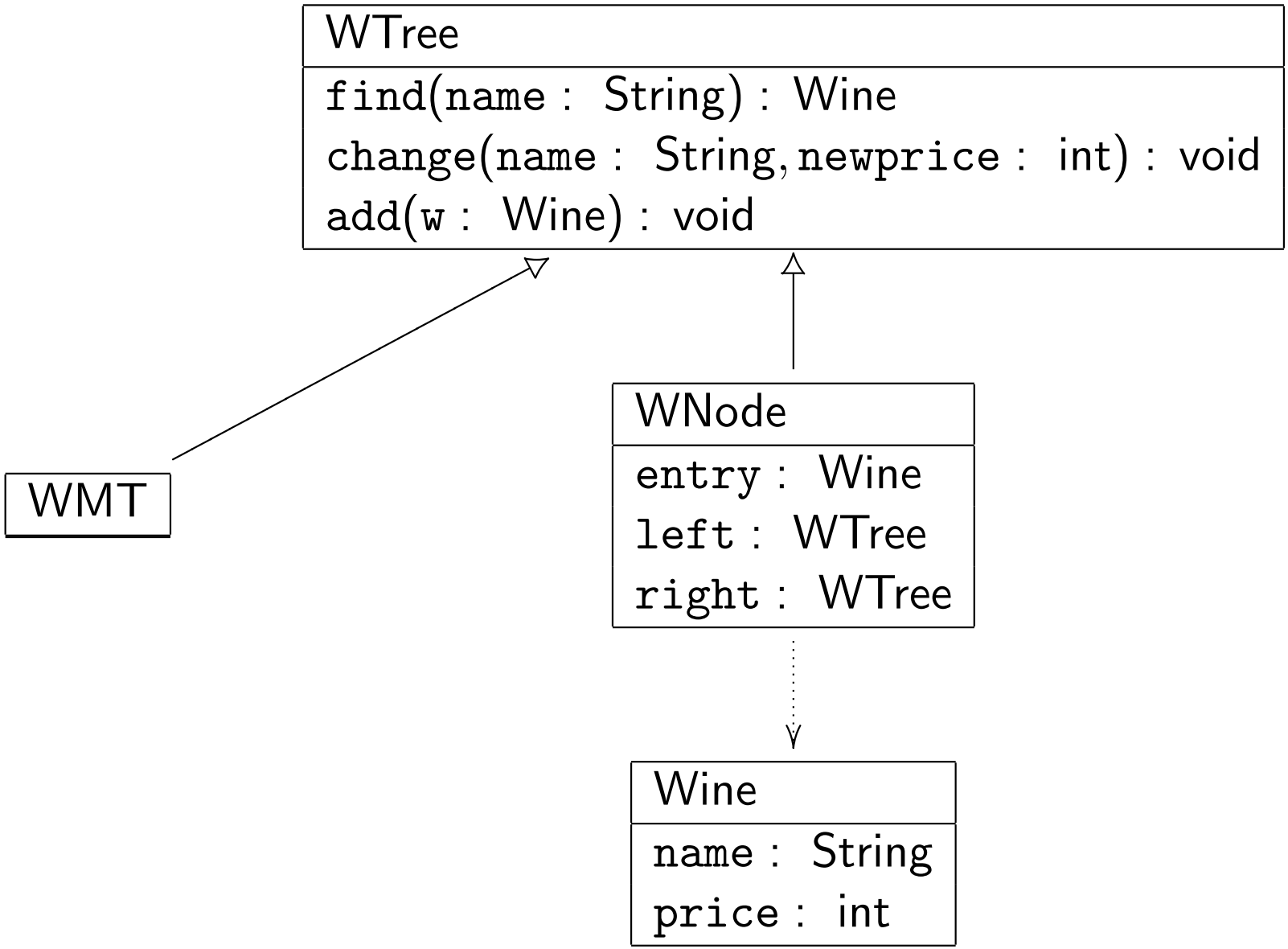
Finite Map

Ein Weingroßhändler will seine Preisliste verwalten. Er wünscht folgende Operationen

- *zu einem Wein den Preis ablegen,*
- *einen Preiseintrag ändern,*
- *den Preis eines Weins abfragen.*

- Abstrakt gesehen ist die Preisliste eine **endliche Abbildung** von Wein (repräsentiert durch einen String) auf Preise (repräsentiert durch ein int). (*finite map*)
- Da in der Preisliste einige tausend Einträge zu erwarten sind, sollte sie als Suchbaum organisiert sein.

Datenmodellierung Weinpreisliste



Erinnerung Suchbaum

- Ein Suchbaum ist entweder leer (WMT) oder besteht aus einem Knoten (WNode), der ein Wine-Objekt entry, sowie zwei Suchbäume `left` und `right` enthält.
- Invariante:
  - Alle Namen von Weinen in `left` sind kleiner als der von `entry`.
  - Alle Namen von Weinen in `right` sind größer als der von `entry`.

Die `find`-Methode

- in WMT

```
Wine find (String name) {
  return null;
}
```

- in WNode

```
Wine find (String name) {
  int r = this.entry.compareName (name);
  if (r == 0) {
    return this.entry;
  } else {
    if (r > 0) { // this wine's name is greater than the one we are looking for
      return this.left.find (name);
    } else {
      return this.right.find (name);
    }
  }
}
```

Die Wunschliste für `Wine`

`int compareName (String name)` liefert 0, falls die Namen übereinstimmen, > 0, falls der gesuchte Name kleiner ist und < 0 sonst.

```
int compareName (String name) {
  return this.name.compareTo (name); // library method
}
```

Die Wunschliste für `Wine`

`int compareName (String name)` liefert 0, falls die Namen übereinstimmen, > 0, falls der gesuchte Name kleiner ist und < 0 sonst.

```
int compareName (String name) {
  return this.name.compareTo (name); // library method
}
```

Aus der `java.lang.String` Dokumentation

`public int compareTo(String anotherString)`
Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string. The result is zero if the strings are equal; compareTo returns 0 exactly when the equals(Object) method would return true.

Beobachtung

- Die `find`-Methode ist bereits endrekursiv und (nichts) akkumulierend.

$\Rightarrow$ Sie kann in eine **while**-Anweisung umgewandelt werden.

- Voraussetzung: passendes Durchlauf-Interface auf WTree

Beobachtung

- Die `find`-Methode ist bereits endrekursiv und (nichts) akkumulierend.

$\Rightarrow$ Sie kann in eine **while**-Anweisung umgewandelt werden.

- Voraussetzung: passendes Durchlauf-Interface auf WTree

Durchlaufen von WTree

```
interface WTree {
  ...
  boolean isEmpty ();
  Wine getEntry ();
  WTree getLeft ();
  WTree getRight ();
}
```

Implementierung in den Klassen

```
class WMT implements WTree {
  ...
  boolean isEmpty () { return true; }
  Wine getEntry () { return null; }
  WTree getLeft () { return null; }
  WTree getRight () { return null; }
}
```

```
class WNode implements WTree {
  ...
  boolean isEmpty () { return false; }
  Wine getEntry () { return this.entry; }
  WTree getLeft () { return this.left; }
  WTree getRight () { return this.right; }
}
```

Rekursive `find`-Methode mit Durchlauf-Interface

```
Wine findAux (String name, WTree wtree) {
  if (wtree.isEmpty ()) {
    return null;
  } else {
    int r = wtree.getEntry().compareName (name);
    if (r == 0) {
      return wtree.getEntry();
    } else {
      if (r > 0) { // this wine's name is greater than the one we are looking for
        return this.find (name, wtree.getLeft());
      } else {
        return this.find (name, wtree.getRight());
      }
    }
  }
}
```

Iterative `findAux`-Methode

```
Wine findAux (String name, WTree wtree0) {
  WTree wtree = wtree0;
  while (!wtree.isEmpty()) {
    int r = wtree.getEntry().compareName (name);
    if (r == 0) {
      return wtree.getEntry();
    } else {
      if (r > 0) { // this wine's name is greater than the one we are looking for
        wtree = wtree.getLeft();
      } else {
        wtree = wtree.getRight();
      }
    }
  }
  return null;
}
```

Iterative `find`-Methode

```
Wine find (String name) {
  WTree wtree = this;
  while (!wtree.isEmpty()) {
    int r = wtree.getEntry().compareName (name);
    if (r == 0) {
      return wtree.getEntry();
    } else {
      if (r > 0) { // this wine's name is greater than the one we are looking for
        wtree = wtree.getLeft();
      } else {
        wtree = wtree.getRight();
      }
    }
  }
  return null;
}
```

Die change-Methode

- in WMT

```
void change (String name, int np) {
  return;
}
```

- in WNode

```
void change (String name, int np) {
  int r = this.entry.compareName (name);
  if (r == 0) {
    this.entry.price = np;
    return;
  } else {
    if (r > 0) { // this wine's name is greater than the one we are looking for
      this.left.change (name, np); return;
    } else {
      this.right.change (name, np); return;
    }
  }
}
```

Beobachtung

- Auch `change` ist endrekursiv und akkumulierend.

$\Rightarrow$ **while**-Anweisung möglich.

- Durchlauf-Interface ist bereits vorbereitet.
- Weitere Schritt sind analog zu `find`:
  - rekursive `changeAux`-Methode unter Verwendung des Durchlauf-Interface
  - Umschreiben in iterative Methode
  - Inlining

Rekursive changeAux-Methode mit Durchlauf-Interface

```
void changeAux (String name, int np, WTree wtree) {
  if (wtree.isEmpty()) {
    return;
  } else {
    int r = wtree.getEntry().compareName (name);
    if (r == 0) {
      wtree.getEntry().price = np;
      return;
    } else {
      if (r > 0) { // this wine's name is greater than the one we are looking for
        this.changeAux (name, np, wtree.getLeft()); return;
      } else {
        this.changeAux (name, np, wtree.getRight()); return;
      }
    }
  }
}
```

Iterative changeAux-Methode

```
void changeAux (String name, int np, WTree wtree0) {
  WTree wtree = wtree0;
  while (!wtree.isEmpty()) {
    int r = wtree.getEntry().compareName (name);
    if (r == 0) {
      wtree.getEntry().price = np;
      return;
    } else {
      if (r > 0) { // this wine's name is greater than the one we are looking for
        wtree = wtree.getLeft();
      } else {
        wtree = wtree.getRight();
      }
    }
  }
  return;
}
```

Iterative change-Methode

```
void change (String name, int np) {
  WTree wtree = this;
  while (!wtree.isEmpty()) {
    int r = wtree.getEntry().compareName (name);
    if (r == 0) {
      wtree.getEntry().price = np;
      return;
    } else {
      if (r > 0) { // this wine's name is greater than the one we are looking for
        wtree = wtree.getLeft();
      } else {
        wtree = wtree.getRight();
      }
    }
  }
  return;
}
```

Die add-Methode

- Die add-Methode hat Ergebnistyp void und muss den unterliegenden Suchbaum verändern.
- Das funktioniert mit dem aktuellen Design nicht richtig.

Die add-Methode

- Die add-Methode hat Ergebnistyp void und muss den unterliegenden Suchbaum verändern.
- Das funktioniert mit dem aktuellen Design nicht richtig.
- Hier ein Versuch: in WMT

```
void add (Wine w) {
  ...;
}
```

An dieser Stelle steht der Methode nichts zur Verfügung: sie kann nichts bewirken. Also muss der Test, ob ein leerer Suchbaum besucht wird, schon vor Eintritt in den Baum geschehen und dort in `left` oder `right` der leere Baum überschrieben werden.

Die add-Methode

- Die add-Methode hat Ergebnistyp void und muss den unterliegenden Suchbaum verändern.
- Das funktioniert mit dem aktuellen Design nicht richtig.
- Hier ein Versuch: in WMT

```
void add (Wine w) {
  ...;
}
```

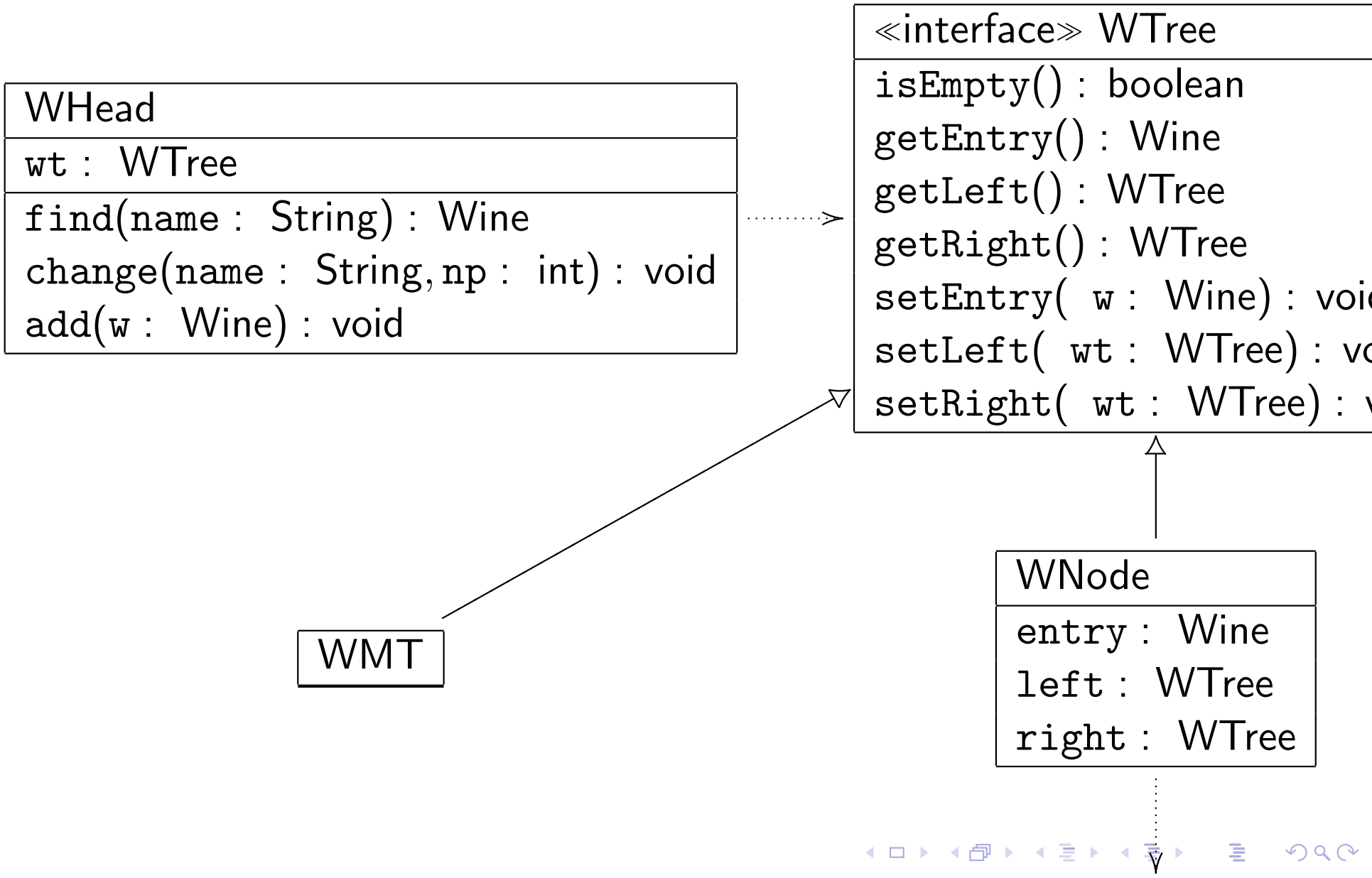
An dieser Stelle steht der Methode nichts zur Verfügung: sie kann nichts bewirken. Also muss der Test, ob ein leerer Suchbaum besucht wird, schon vor Eintritt in den Baum geschehen und dort in `left` oder `right` der leere Baum überschrieben werden.

- Weiteres Problem: Jeder Baum ist zu Beginn leer. Was soll beim Einfügen des ersten Eintrags überschrieben werden?

Ausweg

- Andere Datenmodellierung, bedingt durch die Veränderlichkeit der Baumstruktur.
- Füge der Datenstruktur ein separates Headerobjekt hinzu, das den eigentlichen Suchbaum enthält.
- Dieses Headerobjekt enthält die Operationen.
- Der eigentliche Suchbaum, bestehend aus WMT und WNode Objekten, implementiert lediglich das Durchlauf-Interface.

Neue Datenmodellierung



Implementierung in den Klassen

```
class WMT implements WTree {
  ...
  boolean isEmpty () { return true; }
  Wine getEntry () { return null; }
  WTree getLeft () { return null; }
  WTree getRight () { return null; }
  public void setEntry(Wine w) { return; }
  public void setLeft(WTree wt) { return; }
  public void setRight(WTree wt) { return; }
}
```

Implementierung in Wnode

```
class WNode implements WTree {
  ...
  boolean isEmpty () { return false; }
  Wine getEntry () { return this.entry; }
  WTree getLeft () { return this.left; }
  WTree getRight () { return this.right; }
  public void setEntry(Wine w) { this.entry = w; return; }
  public void setLeft(WTree wt) { this.left = wt; return; }
  public void setRight(WTree wt) { this.right = wt; return; }
}
```

Code für WHead

```
// Interface und Funktionalität für Suchbaum
class WHead {
  private WTree wt;
  private final wmt = new WMT ();

  public WHead () {
    this.wt = this.wmt;
  }

  public Wine find (String name) {
    WTree wtree = this.wt; ...
  }
  public void change (String name, int newprice) {
    WTree wtree = this.wt; ...
  }
```

- Die Implementierung der Methoden `find` und `change` ist unverändert (bis auf die erste Zeile).
- Das Attribut **final** an einem Feld bewirkt, dass der Wert des Feldes nur einmal während der Initialisierung gesetzt werden darf.

```
public void add (Wine w) {
  if (this.wt.isEmpty ()) {
    this.wt = new WNode (w, this.wmt, this.wmt); return;
  } else {
    String name = w.name; WTree wtree = this.wt;
    while (!wtree.isEmpty()) {
      Wine e = wtree.getEntry();
      int r = e.compareName (name);
      if (r == 0) { // überschreibe vorhandenen Eintrag
        wtree.setEntry(w); return;
      } else {
        if (r > 0) {
          WTree w1 = wtree.getLeft();
          if (w1.isEmpty ()) {
            wtree.setLeft(new WNode (w, wmt, wmt)); return;
          } else {
            wtree = w1;
          }
        } else {
          WTree w1 = wtree.getRight();
          if (w1.isEmpty()) {
            wtree.setRight(new Node (w, wmt, wmt)); return;
          } else {
            wtree = w1;
          }
        }
      }
```

Fazit

- Für *funktionale Datenstrukturen* können alle Operationen direkt (entweder rekursiv oder per **while**-Anweisung) definiert werden.
- Bei einer Änderung wird eine neue Instanz der Datenstruktur erzeugt, die Objekte gemeinsam mit der alten Instanz verwendet. Die alte Instanz kann weiter verwendet werden.
- Für *imperative Datenstrukturen* müssen in der Regel
  - die Struktur von den gewünschten Operationen getrennt werden
  - auf der Struktur sind nur Durchläufe möglich
  - für Verwaltungszwecke und zur Behandlung von Randfälle zusätzliche Objekte angelegt werden.
- Bei einer Änderung wird die alte Instanz zerstört und kann nicht mehr verwendet werden.