Vorlesung 04: Imperative Methoden

Peter Thiemann

Universität Freiburg, Germany

SS 2010

Inhalt

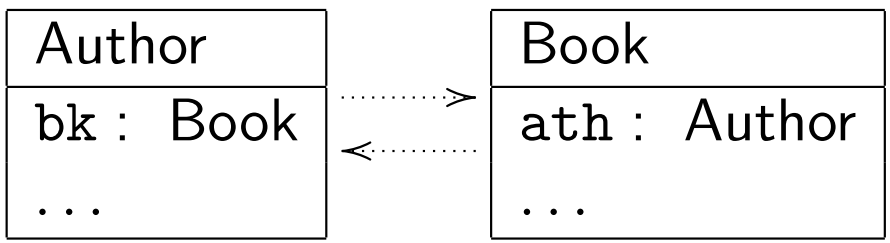
Imperative Methoden

Zirkuläre Datenstrukturen

Verwalte Informationen über Bücher. Ein Buchtitel wird beschrieben durch den Titel, den Preis, die vorrätige Menge und den Autor. Ein Autor wird beschrieben durch Vor- und Nachnamen, 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 derselbe 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 derselbe Autor wieder eingesetzt sein...

- Buch zuoberst

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

Bei ???? müsste dasselbe 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 (*Referenztypen*) passt. D.h., jede Variable bzw. Feld von Klassen- oder Interfacetyp kann auch `null` sein.
- Ein Feldzugriff oder Methodenaufruf auf `null` schlägt fehl. Daher muss vor jedem Feldzugriff bzw. Methodenaufruf sichergestellt werden, dass der jeweilige Empfänger nicht `null` ist!
- Vorsicht: `null` ist der Startwert für alle Instanzvariable, die nicht explizit initialisiert werden.

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 niemals direkt in die Felder von Objekten fremder Klassen hineinschreiben*.
- Das könnte zu illegalen Komponentenwerten in diesen Objekten führen.

$\Rightarrow$ Objekte sollten Methoden zum Setzen von Feldern bereitstellen (soweit von außerhalb des Objektes erforderlich).

- 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()

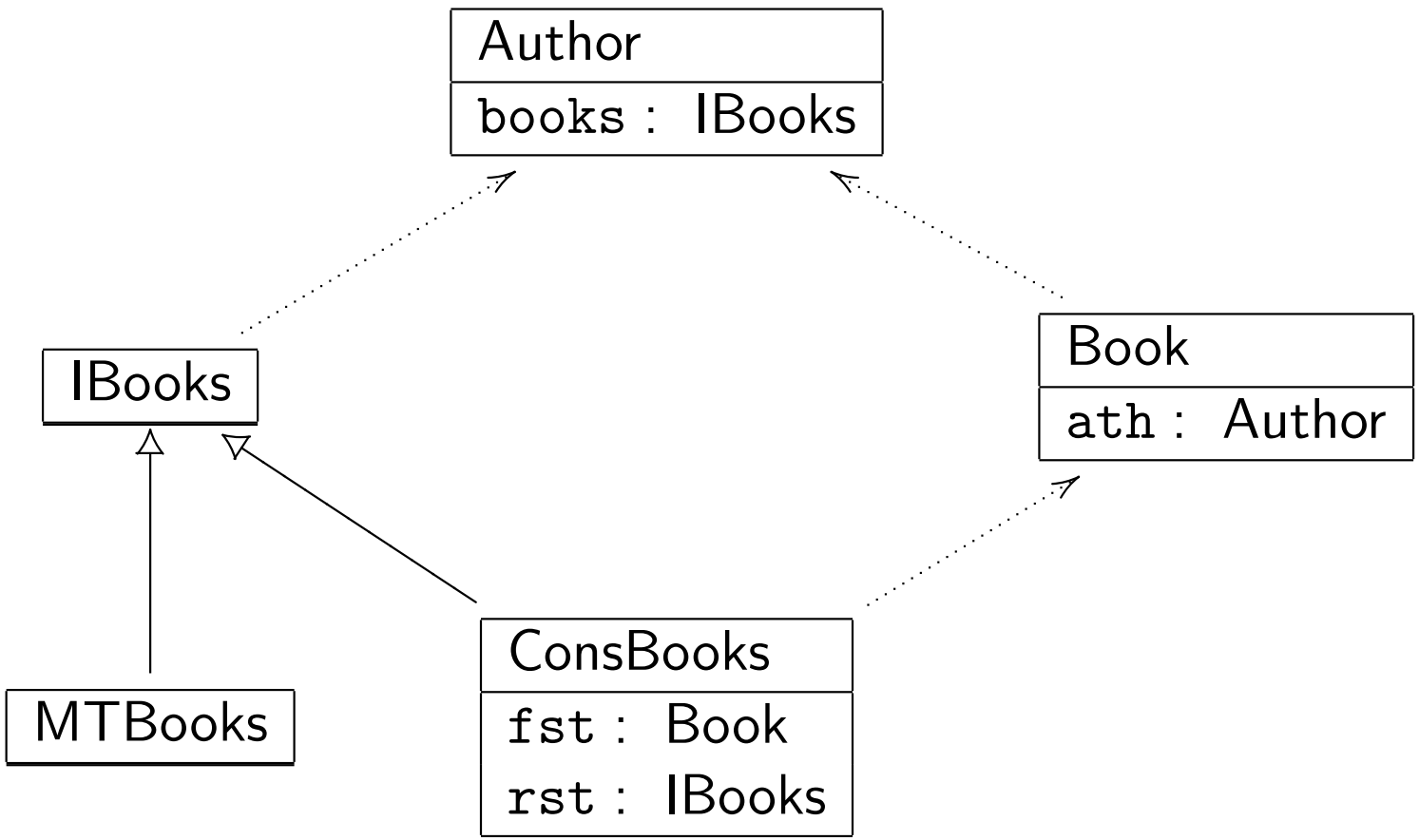
Fehlererkennung

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

- addBook soll fehlschlagen, falls schon ein Buch eingetragen ist.
- InputOutput.error(...) beendet die Programmausführung mit einer Fehlermeldung.

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 unerwartete Effekte haben, da die Feldinitialisierungen ablaufen, **bevor** der Konstruktor ausgeführt wird.

Initialisierungsreihenfolge

```
class StrangeExample {
    int x;

    StrangeExample () { this.x = 100; }

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

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

Initialisierungsreihenfolge

```
class StrangeExample {
    int x;

    StrangeExample () { this.x = 100; }

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

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

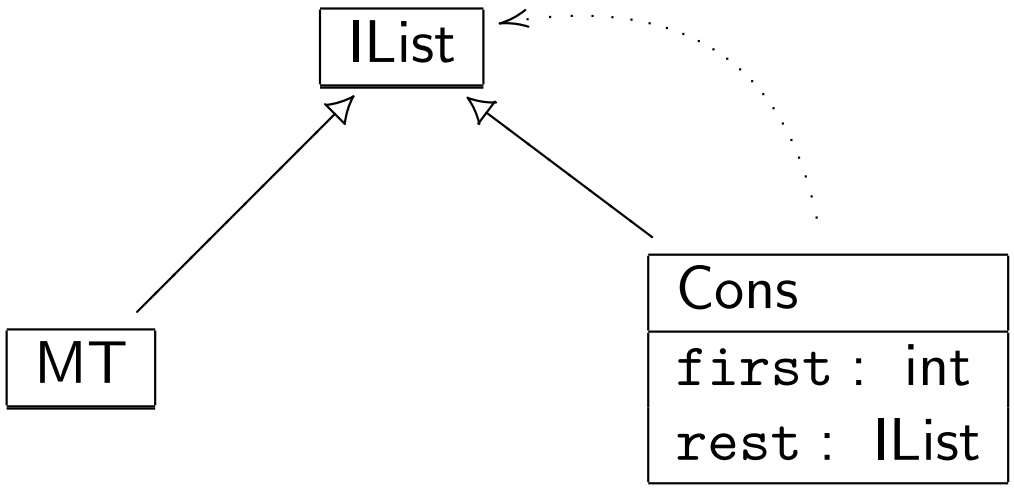
Initialisierungsreihenfolge

```
class StrangeExample {
    int x;

    StrangeExample () { this.x = 100; }

    boolean test = this.x == 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 mit 0 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 ());

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

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

```
new CyclicList ().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;
  }
}
```

- Die Instanzvariablen `first` und `rest` sind als **private** deklariert.
- Nur Methoden von Cons können direkt auf `first` und `rest` zugreifen.
- So ist es unmöglich, aus anderen Klassen das `first`- oder das `rest`-Feld zu lesen oder zu überschreiben.

Geheimhaltung mit Lesezugriff

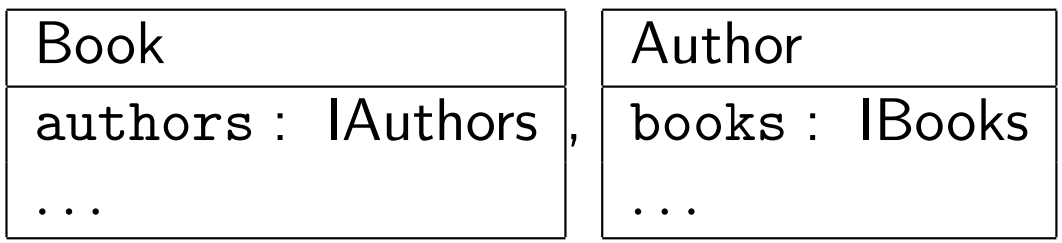
```
class Cons implements IList {
  private int first;
  private IList rest;

  public Cons (int first, IList rest) { ... }

  public int getFirst() { return first; }
  public IList getRest() { return rest; }
}
```

- Externer Lesezugriff durch **public** *Getter-Methoden*
- Kein externer Schreibzugriff

Viele Autoren und viele Bücher

Verwalte Informationen über Bücher. 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

- 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 allen Autoren auf dieser Liste hinzu
    public void addBook(Book bk);
}
```

$\Rightarrow$ Die Methode liefert kein Ergebnis.

- Einbindung in den Konstruktor von Book durch

```
    this.authors.addBook(this);
```

Implementierung der Hilfsmethode

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

  public void addBook(Book bk) {
    return;
  }
}
```

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

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

  public void addBook (Book bk) {
    this.first.addBook (bk);
    this.rest.addBook (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 Kontoauszug anfordern. Jedes Bankkonto gehört einer Person.

Bankkonto

- 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)`
  - Kontoauszug: `String balance()`

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

Implementierung des Bankkontos

```
// 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;
  }
```

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

Vererbung

Personen, Sänger und Rockstars

```
// Personen
class Person {
  private String name;
  private int count;

  Person(String name) {
    this.name = name;
    this.count = 0;
  }
```

Methoden von Person

```
// liefert den Namen
public String getName() {
  return this.name;
}
// spricht eine Nachricht
public String say(String message) {
  return message;
}
// steckt Schläge ein
public String slap() {
  if (count<2) {
    count = count + 1;
    return "argh";
  } else {
    count = 0;
    return "ouch";
  }
}
```

Testen von Person

```
> Person jimmy = new Person("jimmy");
> jimmy.getName( )
"jimmy"
> jimmy.say("peanut man")
"peanut man"
> jimmy.slap()
"argh"
> jimmy.slap()
"argh"
> jimmy.slap()
"ouch"
> jimmy.slap()
"argh"
```

Sänger als Subklasse von Person

```
// Ein Sänger ist eine spezielle Person
class Singer extends Person {
  Singer(String name) {
    super(name);
  }

  public String sing(String song) {
    return say(song + " tra-la-la");
  }
}
```

- Das Schlüsselwort **extends** deutet an, dass eine Klasse von einer anderen erbt. Hier wird Singer als Subklasse von Person definiert.
- Die erste Anweisung im Konstruktor kann **super**(...) sein. Dadurch wird ein Konstruktor der direkten Superklasse aufgerufen.
- In der Subklasse sind sämtliche Methoden und Felder der Superklasse verfügbar, die nicht durch **private** geschützt sind.

Testen von Sängern

```
> Singer jerry = new Singer("jerry");
> jerry.getName( )
"jerry"
> jerry.say("peanut man")
"peanut man"
> jerry.sing("born in the usa")
"born in the usa tra-la-la"
> jerry.slap()
"argh"
> jerry.slap()
"argh"
> jerry.slap()
"ouch"
> jerry.slap()
"argh"
```

Rockstar als Subklasse von Singer

```
// Ein Rockstar ist ein spezieller Sänger
class Rockstar extends Singer {
  Rockstar(String name) {
    super(name);
  }
  public String say(String message) {
    return super.say("Dude, " + message);
  }
  public String slap() {
    return "Pain just makes me stronger";
  }
}
```

- Die Methoden sing und getName werden von den Superklassen *geerbt*.
- Die Methoden say und slap werden *überschrieben*.
- Der Aufruf **super**.say(...) ruft die Implementierung der Methode say in der nächsten Superklasse auf.

Testen von Rockstars

```
> Rockstar bruce = new Rockstar("bruce");
> bruce.getName()
"bruce"
> bruce.say("trust me")
"Dude, trust me"
> bruce.sing("born in the usa")
"Dude, born in the usa tra-la-la"
> bruce.slap()
"Pain just makes me stronger"
> bruce.slap()
"Pain just makes me stronger"

> Singer bruce1 = bruce;
> bruce1.say("it's me")
"Dude, it's me"
> bruce1.sing("mc")
"Dude, mc tra-la-la"
```

Vererbung und Dynamic Dispatch

- Jedes Objekt besitzt einen unveränderlichen *Laufzeittyp*, die Klasse, von der es konstruiert worden ist.
- Bei einem Methodenaufruf wird immer die Methodenimplementierung der dem Laufzeittyp nächstgelegenen Superklasse ausgewählt. (*dynamic dispatch*)
- Die Auswahl erfolgt dynamisch **zur Laufzeit des Programms** und ist unabhängig vom Typ der Variable, in der das Objekt abgelegt ist.

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, long 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, Color c);

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

  // Rechteck zeichnen
  void drawRect(Posn p, int w, int h, Color 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 javakurs.idraw.*;`
  - `import java.awt.Color;`

$\Rightarrow$ 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 = this.block.drop();
    if (block.isAt(100)) { this.endOfWorld("bottom reached"); }
    return;
  }

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

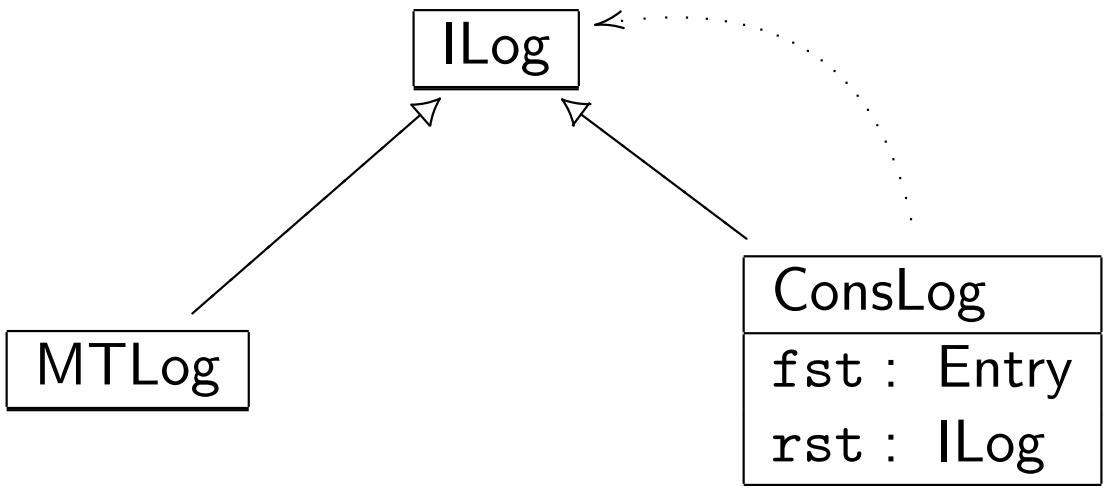
Fallender Block, Imperative Version

```
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) {
        c.drawRect (new Posn (x, y), SIZE, SIZE, Color.WHITE);
    }
}
```

Iteration

Iteration

Erinnerung: das Lauftagebuch



- Ziel: Definiere effiziente Methoden auf ILog
- Beispiel: Methode `length()`

Implementierung von `length()`

- in ILog

```
// berechne die Anzahl der Einträge
int length();
```

- in MTLog

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

- in ConsLog

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

Ein Effizienzproblem

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

Implementierung von `lengthAcc()`

- in ILog

```
// berechne die Anzahl der Einträge
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 Methoden mit Akkumulator sind endrekursiv und *könnten* in konstantem Platz implementiert werden.
- Aber Java (bzw. die Java Virtual Machine, JVM) tut das nicht.
- Abhilfe: Durchlaufe die Liste mit einer **while**-Schleife.

Die **while**-Anweisung

- Allgemeine Form

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

- *bedingung* ist ein boolescher Ausdruck.
- Die *anweisungen* bilden den *Schleifenrumpf*.
- Die Ausführung der **while**-Anweisung läuft wie folgt ab.
- Werte die *bedingung* aus.
  - Ist sie `false`, so ist die Ausführung der **while**-Anweisung beendet.
  - Ist sie `true`, so werden die *anweisungen* ausgeführt.
- Dieser 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

Schritt 1: Definiere neue Superklasse von MTLog und ConsLog

- Neue Klasse muss ILog implementieren

```
class ALog implements ILog {
  public int length () { ... }
  public boolean isEmpty () { ... }
  public boolean getFirst () { ... }
  public ILog getRest () { ... }
}
```

- MTLog und ConsLog sind Subklassen von ALog. Sie erben die Implementierung der Methode `length` und die Implementierungsdeklaration **implements** ILog.

```
class MTLog extends ALog {...}
class ConsLog extends ALog {...}
```

Verwendung des Interface für Listendurchlauf

Schritt 2: Listenlänge mit Hilfe des Durchlaufinterfaces

```
// in Klasse ALog
public int length () {
  return lengthAux (0, this);
}
private int lengthAux (int acc, ILog list) {
  if (list.isEmpty()) {
    return acc;
  } else {
    return lengthAux (acc + 1, list.getRest());
  }
}
```

Verwendung des Interface für Listendurchlauf

Schritt 2: Listenlänge mit Hilfe des Durchlaufinterfaces

```
// in Klasse ALog
public int length () {
  return lengthAux (0, this);
}
private int lengthAux (int acc, ILog list) {
  if (list.isEmpty()) {
    return acc;
  } else {
    return lengthAux (acc + 1, list.getRest());
  }
}
```

- Eine endrekursive Methode wie `lengthAux` kann sofort in eine **while**-Anweisung umgesetzt werden:
  - Aus den Parametern werden lokale Variablen.
  - Aus dem rekursiven Aufruf werden Zuweisungen auf diese Variablen.

Verwendung des Interface für Listendurchlauf

Schritt 3: Iterative Version von `lengthAux`

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

- Aufruf bleibt gleich:

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

Verwendung des Interface für Listendurchlauf

Schritt 3: Iterative Version von `lengthAux`

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

- Aufruf bleibt gleich:

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

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

Verwendung des Interface für Listendurchlauf

Schritt 4: Alles in der `length` Methode

```
// in Klasse ALog
public 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.

Analog: Iterative Implementierung von `totalDistance`

```
// in Klasse ALog
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

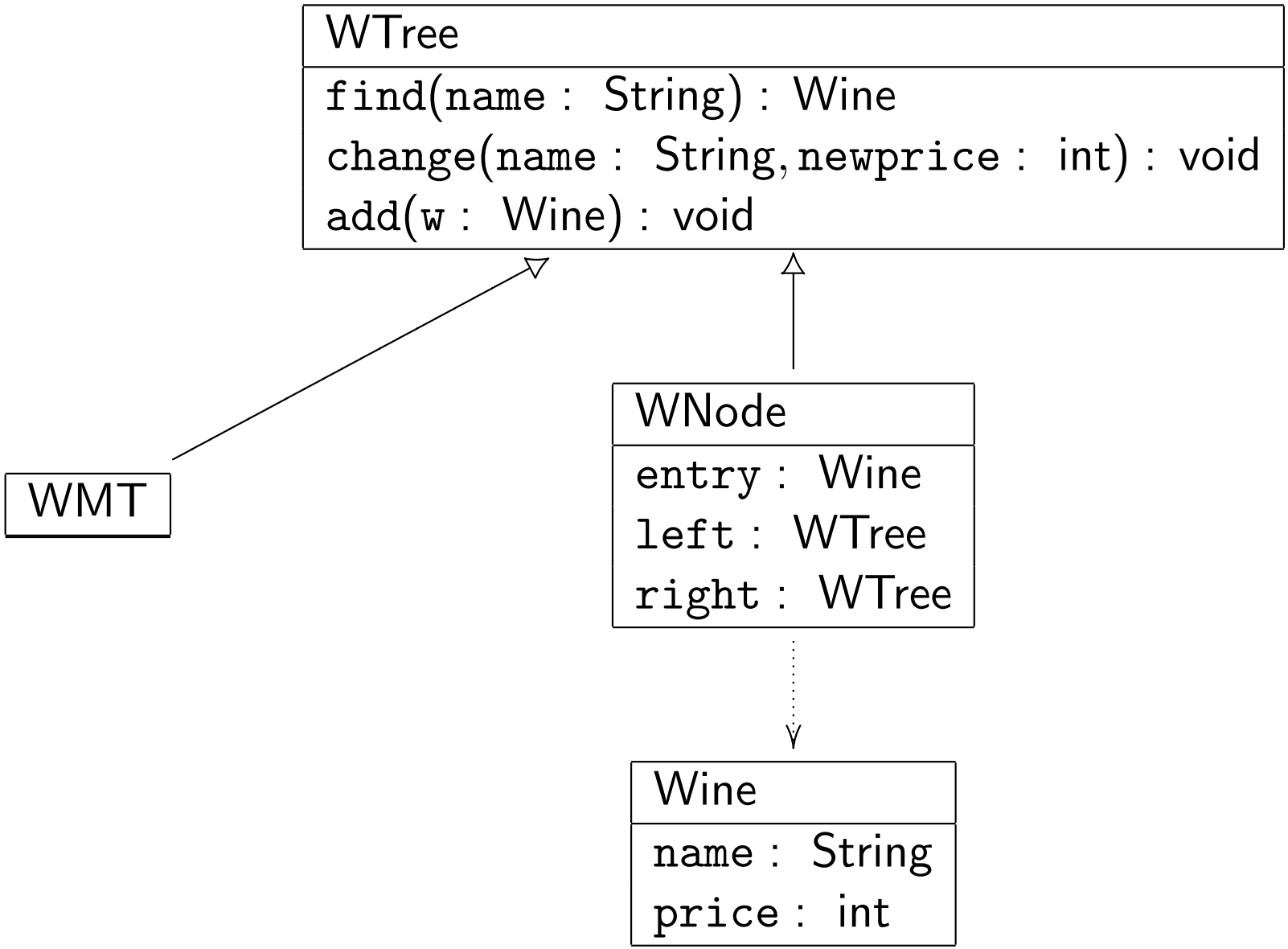
Finite Map

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



Binärer Suchbaum

- Ein binärer Suchbaum ist entweder leer (WMT) oder besteht aus einem Knoten (WNode), der ein Wine-Objekt entry, sowie zwei binäre 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) {
  ...;
}
```

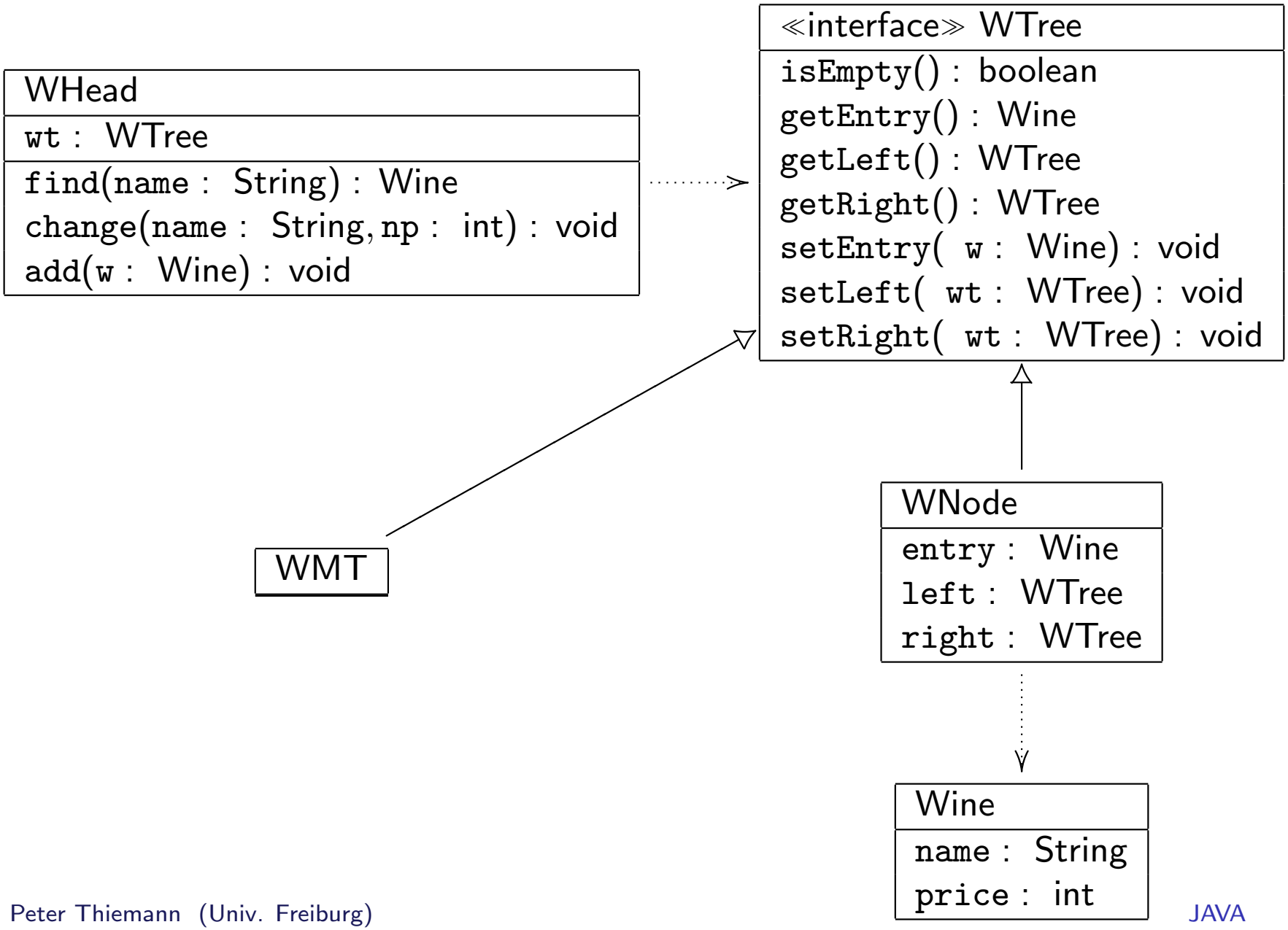
An dieser Stelle steht der Methode nichts zur Verfügung: sie kann nichts bewirken. Also muss der Test, ob ein leerer Suchbaum besucht wird, schon vor Eintritt in den Baum geschehen und dort in `left` oder `right` der leere Baum überschrieben werden.

- 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.