

Informatik I, Programmierung

Java 02: Methoden

Peter Thiemann

Universität Freiburg, Germany

WS 2008/2009

Inhalt

Funktionale Methoden

Ausdrücke mit primitiven Datentypen

Methoden entwerfen

Methoden mit Fallunterscheidung

Methoden auf Vereinigungen von Klassen

Methoden auf rekursiven Klassen

Ausdrücke mit primitiven Datentypen

int, double, boolean

- ▶ Ausdrücke dienen zur Berechnung von neuen Werten
- ▶ Für primitive Datentypen sind viele *Infixoperatoren* vordefiniert, die auf die gewohnte Art verwendet werden.
 - ▶ `60 * .789`
 - ▶ `this.x + 2`
 - ▶ `Math.PI * radius`

Bemerkungen

- ▶ `this.x` liefert den Wert der Instanzvariable `x`
- ▶ `Math.PI` liefert den vordefinierten Wert von π (als Gleitkommazahl)

Ausdrücke mit mehreren Operatoren

Für die Operatoren gelten die üblichen Präzedenzregeln (Punkt- vor Strichrechnung usw.)

- ▶ `5 * 7 + 3` entspricht `(5 * 7) + 3`
- ▶ `position > 0 && position <= maxpos` entspricht `(position > 0) && (position <= maxpos)`

Empfehlung

Verwende generell Klammern

Arithmetische und logische Operatoren (Auszug)

Symbol	Parameter	Ergebnis	Beispiel	
!	boolean	boolean	!true	logische Negation
&&	boolean, boolean	boolean	a && b	logisches Und
	boolean, boolean	boolean	a b	logisches Oder
+	numerisch, numerisch	numerisch	a + 7	Addition
-	numerisch, numerisch	numerisch	a - 7	Subtraktion
*	numerisch, numerisch	numerisch	a * 7	Multiplikation
/	numerisch, numerisch	numerisch	a / 7	Division
%	numerisch, numerisch	numerisch	a % 7	Modulo
<	numerisch, numerisch	boolean	y < 7	kleiner als
<=	numerisch, numerisch	boolean	y <= 7	kleiner gleich
>	numerisch, numerisch	boolean	y > 7	größer als
>=	numerisch, numerisch	boolean	y >= 7	größer gleich
==	numerisch, numerisch	boolean	y == 7	gleich
!=	numerisch, numerisch	boolean	y != 7	ungleich

Der primitive Typ String

- ▶ String ist vordefiniert, ist aber ein Klassentyp
d.h. jeder String ist ein Objekt
- ▶ Ein Infixoperator ist definiert:

Symbol	Parameter	Ergebnis	Beispiel
+	String, String	String	s1 + s2 Stringverkettung

`"laber" + "fasel" // ==> "laberfasel"`

- ▶ Wenn einer der Parameter numerisch oder boolean ist, so wird er automatisch in einen String *konvertiert*.

`"x=" + 5 // ==> "x=5"`

`"this is " + false // ==> "this is false"`

Methodenaufrufe

Methoden von String

- ▶ Weitere Stringoperation sind als *Methoden* der Klasse String definiert und durch *Methodenaufrufe* verfügbar.
- ▶ Beispiele
 - ▶ `"arachnophobia".length()` Stringlänge
 - ▶ `"wakarimasu".concat("ka")` Stringverkettung
- ▶ Allgemeine Form
object.method(eArg, ...)
 - ▶ *object* Ausdruck, dessen Wert ein Objekt ist
 - ▶ *method* Name einer Methode dieses Objektes
 - ▶ *eArg* Argumentausdruck für die Methode
- ▶ Schachtelung möglich (Auswertung von links nach rechts)
`"mai".concat("karenda").length()`

Einige String Methoden

Methoden	Parameter	Ergebnis	Beispiel
<code>length</code>	<code>()</code>	<code>int</code>	<code>"xy".length()</code>
<code>concat</code>	<code>(String)</code>	<code>String</code>	<code>"xy".concat("zw")</code>
<code>toLowerCase</code>	<code>()</code>	<code>String</code>	<code>"XyZ".toLowerCase()</code>
<code>toUpperCase</code>	<code>()</code>	<code>String</code>	<code>"XyZ".toUpperCase()</code>
<code>equals</code>	<code>(String)</code>	<code>boolean</code>	<code>"XyZ".equals("xYz")</code>
<code>endsWith</code>	<code>(String)</code>	<code>boolean</code>	<code>"XyZ".endsWith("yZ")</code>
<code>startsWith</code>	<code>(String)</code>	<code>boolean</code>	<code>"XyZ".startsWith("Xy")</code>

- ▶ Insgesamt 54 Methoden (vgl. `java.lang.String`)

Methoden entwerfen

Objekte erhalten ihre Funktionalität durch *Methoden*

Beispiel

Zu einer Teelieferung (bestehend aus Teesorte, Kilopreis und Gewicht) soll der Gesamtpreis bestimmt werden.

- ▶ Implementierung durch Methode `cost()`
- ▶ Keine Parameter, da alle Information im Tea-Objekt vorhanden ist.
- ▶ Ergebnis ist ein Preis, repräsentiert durch den Typ `int`

- ▶ Verwendungsbeispiel:

```
Tea tAssam = new Tea("Assam", 2790, 150);  
tAssam.cost()  
soll 418500 liefern
```

Methodendefinition

```
// Repräsentation einer Rechnung für eine Teelieferung
```

```
class Tea {
```

```
    String kind; // Teesorte
```

```
    int price; // in Eurocent pro kg
```

```
    int weight; // in kg
```

```
// Konstruktor (wie vorher)
```

```
    Tea (String kind, int price, int weight) { ... }
```

```
// berechne den Gesamtpreis dieser Lieferung
```

```
    int cost() { ... }
```

```
}
```

- ▶ Methodendefinitionen nach Konstruktor
- ▶ Methode `cost()`
 - ▶ Ergebnistyp `int`
 - ▶ keine Parameter
 - ▶ Rumpf muss jetzt ausgefüllt werden

Klassendiagramm mit Methoden

Gleiche Information im Klassenkasten

Tea
kind : String
price : int
weight : int
cost() : int

- ▶ Dritte Abteilung enthält die Kopfzeilen der Methoden
Signaturen von Methoden

Entwicklung der Methode cost

- ▶ Jede Methode kann auf ihr *Empfängerobjekt* über die Variable `this` zugreifen

```
// berechne den Gesamtpreis dieser Lieferung  
int cost() { ... this ... }
```

Entwicklung der Methode cost

- ▶ Jede Methode kann auf ihr *Empfängerobjekt* über die Variable `this` zugreifen

```
// berechne den Gesamtpreis dieser Lieferung  
int cost() { ... this ... }
```

- ▶ Zugriff auf die Felder des Empfängerobjekts erfolgt mittels `this.feldname`

```
// berechne den Gesamtpreis dieser Lieferung  
int cost() { ... this.kind ... this.price ... this.weight ... }
```

(kind spielt hier keine Rolle)

Entwicklung der Methode cost

- ▶ Jede Methode kann auf ihr *Empfängerobjekt* über die Variable `this` zugreifen

```
// berechne den Gesamtpreis dieser Lieferung
int cost() { ... this ... }
```

- ▶ Zugriff auf die Felder des Empfängerobjekts erfolgt mittels `this.feldname`

```
// berechne den Gesamtpreis dieser Lieferung
int cost() { ... this.kind ... this.price ... this.weight ... }
```

(kind spielt hier keine Rolle)

- ▶ Der Rückgabewert der Methode wird durch die **return**-Anweisung spezifiziert.

```
18 // berechne den Gesamtpreis dieser Lieferung
19 int cost() {
20     return this.price * this.weight;
21 }
```

Methodentest

```

1 // Tests für die Methoden von Tea
2 class TeaExamples {
3     Tea tea1 = new Tea("Assam", 1590, 150);
4     Tea tea2 = new Tea("Darjeeling", 2790, 220);
5     Tea tea3 = new Tea("Ceylon", 1590, 130);
6
7     boolean testTea1 = check this.tea1.cost() expect 238500;
8     boolean testTea2 = check this.tea2.cost() expect 613800;
9     boolean testTea3 = check this.tea3.cost() expect 206700;
10
11     TeaExamples() { }
12 }

```

- ▶ Separate Testklasse
- ▶ Felder, deren Namen mit `test...` beginnen, werden von der IDE als Tests registriert
- ▶ Der `check`-Ausdruck "**check** *expr1* **expect** *expr2*" liefert `true`, falls beide Ausdrücke den gleichen Wert haben (Eine Java-Erweiterung)

Methoden mit Argumenten

Primitive Datentypen

Der Teelieferant sucht nach Billigangeboten, bei denen der Kilopreis kleiner als eine vorgegebene Schranke ist.

- ▶ Argumente von Methoden werden wie Felder deklariert

```
// liegt der Kilopreis dieser Lieferung unter limit?  
boolean cheaperThan(int limit) { ... this ... }
```

- ▶ Gewünschtes Verhalten:

```
check new Tea ("Earl Grey", 3945, 75).cheaperThan (2000) expect false  
check new Tea ("Ceylon", 1590, 400).cheaperThan (2000) expect true
```


Methoden mit Argumenten

Primitive Datentypen/2

► Methodensignatur

```
// liegt der Kilopreis dieser Lieferung unter limit?  
boolean cheaperThan(int limit) { ... this ... }
```

Methoden mit Argumenten

Primitive Datentypen/2

▶ Methodensignatur

```
// liegt der Kilopreis dieser Lieferung unter limit?  
boolean cheaperThan(int limit) { ... this ... }
```

▶ Im Rumpf der Methode dürfen die Felder des Objekts und die Parameter verwendet werden.

```
// liegt der Kilopreis dieser Lieferung unter limit?  
boolean cheaperThan(int limit) { ... this.price ... limit ... }
```

(kind und weight spielen hier keine Rolle)

Methoden mit Argumenten

Primitive Datentypen/2

► Methodensignatur

```
// liegt der Kilopreis dieser Lieferung unter limit?  
boolean cheaperThan(int limit) { ... this ... }
```

► Im Rumpf der Methode dürfen die Felder des Objekts und die Parameter verwendet werden.

```
// liegt der Kilopreis dieser Lieferung unter limit?  
boolean cheaperThan(int limit) { ... this.price ... limit ... }
```

(kind und weight spielen hier keine Rolle)

► Der Rückgabewert der Methode wird durch die **return**-Anweisung spezifiziert.

```
24 boolean cheaperThan(int limit) {  
25     return this.price < limit;  
26 }
```

Methoden mit Argumenten

Klassentypen

Der Teelieferant möchte Lieferungen nach ihrem Gewicht vergleichen.

- ▶ Argumente von Methoden werden wie Felder deklariert

```
// wiegt diese Lieferung mehr als eine andere?
boolean lighterThan(Tea that) { ... this ... that ... }
```

- ▶ Gewünschtes Verhalten:

```
Tea t1 = new Tea ("Earl Grey", 3945, 75);
Tea t2 = new Tea ("Ceylon", 1590, 400);

check t1.lighterThan (new Tea ("Earl Grey", 3945, 25)) expect false
check t2.cheaperThan (new Tea ("Assam", 1590, 500)) expect true
```

Methoden mit Argumenten

Klassentypen/2

► Methodensignatur

```
// wiegt diese Lieferung mehr als eine andere?  
boolean lighterThan(Tea that) { ... this ... that ... }
```

Methoden mit Argumenten

Klassentypen/2

▶ Methodensignatur

```
// wiegt diese Lieferung mehr als eine andere?  
boolean lighterThan(Tea that) { ... this ... that ... }
```

▶ Alle Felder beider Objekte sind verwendbar:

```
// wiegt diese Lieferung mehr als eine andere?  
boolean lighterThan(Tea that) {  
    ... this.kind ... that.kind ...  
    ... this.price ... that.price ...  
    ... this.weight ... that.weight ...  
}
```

Methoden mit Argumenten

Klassentypen/2

▶ Methodensignatur

```
// wiegt diese Lieferung mehr als eine andere?
boolean lighterThan(Tea that) { ... this ... that ... }
```

▶ Alle Felder beider Objekte sind verwendbar:

```
// wiegt diese Lieferung mehr als eine andere?
boolean lighterThan(Tea that) {
    ... this.kind ... that.kind ...
    ... this.price ... that.price ...
    ... this.weight ... that.weight ...
}
```

▶ Der Methodenrumpf verwendet das Feld `weight` von beiden Objekten

```
29 boolean lighterThan(Tea that) {
30     return this.weight < that.weight;
31 }
```

Rezept für den Methodenentwurf

Ausgehend von einer Klasse

1. erkläre kurz den Zweck der Methode (Kommentar)
2. definiere die Methodensignatur
3. gib Beispiele für die Verwendung der Methode
4. fülle den Rumpf der Methode gemäß dem Muster
 - ▶ **this** und die Felder **this.feldname** dürfen vorkommen
 - ▶ alle Parameter dürfen vorkommen
5. schreibe den Rumpf der Methode
6. definiere die Beispiele als Tests

Methoden mit Fallunterscheidung

Eine Bank verzinst eine Spareinlage jährlich mit einem gewissen Prozentsatz. Der Prozentsatz hängt von der Höhe der Einlage ab. Unter 5000 Euro gibt die Bank 4,9% Zinsen, bis unter 10000 Euro 5,0% und für höhere Einlagen 5,1%. Berechne den Zins für eine Einlage.

- ▶ Klassendiagramm dazu

Deposit
owner : String
amount [in Cent] : int
interest() : double

Analyse der Zinsberechnung

► Beispiele

```
check new Deposit ("Dieter", 120000).interest() expect 5880.0
check new Deposit ("Verona", 500000).interest() expect 25000.0
check new Deposit ("Franjo", 1100000).interest() expect 56100.0
```

- In der Methode `interest` gibt es drei Fälle, die von dem Betrag `this.amount` abhängen.
- Die drei Fälle werden mit einer *bedingten Anweisung* (If-Anweisung) unterschieden.

Bedingte Anweisung

► Allgemeine Form

```
if (bedingung) {  
    anweisung1; // ausgeführt, falls bedingung wahr  
} else {  
    anweisung2; // ausgeführt, falls bedingung falsch  
}
```

► Zur Zeit kennen wir nur die **return**-Anweisung

```
if (bedingung) {  
    return ausdruck1; // ausgeführt, falls bedingung wahr  
} else {  
    return ausdruck2; // ausgeführt, falls bedingung falsch  
}
```

Bedingte Anweisung geschachtelt

- ▶ Die bedingte Anweisung ist selbst eine Anweisung
- ⇒ Schachtelung möglich!

```
if (bedingung1) {  
    // ausgeführt, falls bedingung1 wahr  
    return ausdrück1;  
} else {  
    if (bedingung2) {  
        // ausgeführt, falls bedingung1 falsch und bedingung2 wahr  
        return ausdrück2;  
    } else {  
        // ausgeführt, falls bedingung1 und bedingung2 beide falsch  
        return ausdrück3;  
    }  
}
```

- ⇒ Passt genau zu den Anforderungen an `interest()`!

Bedingte Anweisung für Zinsberechnung

- ▶ Einsetzen der Bedingungen und Auflisten der verfügbaren Instanzvariablen liefert

```
// berechne den Zinsbetrag für diese Objekt
double interest () {
    if (this.amount < 500000) {
        // ausgeführt, falls Betrag < 5000 Euro
        return ... this.owner ... this.amount ... ;
    } else {
        if (this.amount < 1000000) {
            // ausgeführt, falls Betrag >= 5000 Euro und < 10000 Euro
            return ... this.owner ... this.amount ...;
        } else {
            // ausgeführt, falls Betrag >= 10000 Euro
            return ... this.owner ... this.amount ...;
        }
    }
}
```

Methode für Zinsberechnung

- ▶ Einsetzen der Zinssätze und der Zinsformel liefert

```
// berechne den Zinsbetrag für diese Objekt
double interest () {
    if (this.amount < 500000) {
        // ausgeführt, falls Betrag < 5000 Euro
        return this.amount * 4.9 / 100 ;
    } else {
        if (this.amount < 1000000) {
            // ausgeführt, falls Betrag >= 5000 Euro und < 10000 Euro
            return this.amount * 5.0 / 100;
        } else {
            // ausgeführt, falls Betrag >= 10000 Euro
            return this.amount * 5.1 / 100;
        }
    }
}
```

Verbesserte Zinsberechnung

- ▶ Nachteil der `interest()`-Methode:
Verquickung der Berechnung des Zinssatzes mit der Fallunterscheidung
- ▶ Dadurch taucht die Zinsformel 3-mal im Quelltext auf
- ▶ Besser: Kapselung der Zinsformel und der Fallunterscheidung jeweils in einer eigenen Methode

Deposit
owner : String
amount [in Cent] : int
rate() : double
payInterest() : double

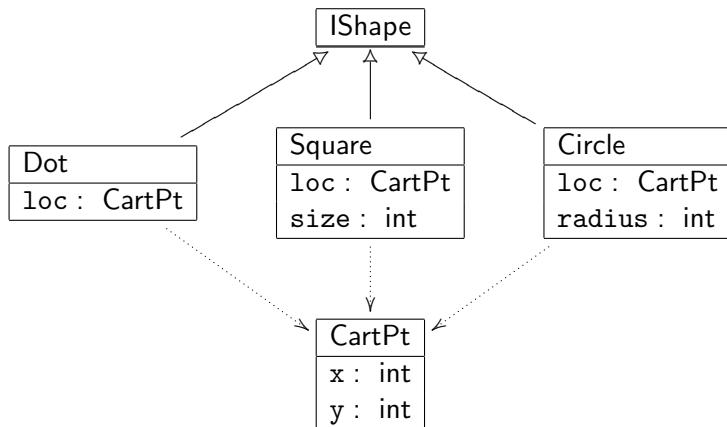
Interner Methodenaufruf

```
27 // bestimme den Zinssatz aus der Höhe der Einlage
28 double rate () {
29     if (this.amount < 500000) {
30         return 4.9;
31     } else {
32         if (this.amount < 1000000) {
33             return 5.0;
34         } else {
35             return 5.1;
36         }
37     }
38 }
41 // berechne den Zinsbetrag
42 double payInterest() {
43     return this.amount * this.rate() / 100;
44 }
```

- ▶ Die Methoden einer Klasse können sich gegenseitig aufrufen.

Methoden auf Vereinigungen von Klassen

Erinnerung: die Klassenhierarchie zu IShape mit Subtypen Dot, Square und Circle



Methoden für IShape

Das Programm zur Verarbeitung von geometrischen Figuren benötigt Methoden zur Lösung folgender Probleme.

1. *double area()*

Wie groß ist die Fläche einer Figur?

2. *double distToO()*

Wie groß ist der Abstand einer Figur zum Koordinatenursprung?

3. *boolean in(CartPt p)*

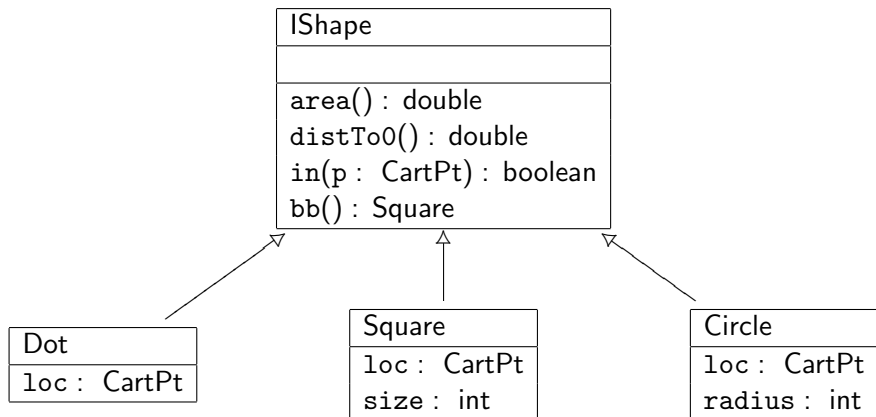
Liegt ein Punkt innerhalb einer Figur?

4. *Square bb()*

Was ist die Umrandung einer Figur? Die Umrandung ist das kleinste Rechteck, das die Figur vollständig überdeckt. (Für die betrachteten Figuren ist es immer ein Quadrat.)

Methodensignaturen im Interface IShape

- ▶ Die Methodensignaturen werden im Interface IShape definiert.
- ▶ Das stellt sicher, dass jedes Objekt vom Typ IShape die Methoden implementieren muss.



Java-Code für IShape

```
// geometrische Figuren
interface IShape {
    // berechne die Fläche dieser Figur
    public double area ();
    // berechne den Abstand dieser Figur zum Ursprung
    public double distTo0();
    // ist der Punkt innerhalb dieser Figur?
    public boolean in (CartPt p);
    // berechne die Umrandung dieser Figur
    public Square bb();
}
```

- ▶ **public** zeigt an, dass die Methode überall verfügbar sein soll, wo IShape bekannt ist.

Anwendungsbeispiele für `area()`

```
2 class ShapeExamples {
3     IShape dot = new Dot (new CartPt (4,3));
4     IShape squ = new Square (new CartPt (4,3), 3);
5     IShape cir = new Circle (new CartPt (12,5), 2);
6     // tests
9     boolean testDot1 = check dot.area() expect 0.0 within 0.1;
10    boolean testSqu1 = check squ.area() expect 9.0 within 0.1;
11    boolean testCir1 = check cir.area() expect 12.56 within 0.01;
24    // constructor
25    ShapeExamples () {}
26 }
```

Bemerkung

Beim Rechnen mit `double` können Rundungsfehler auftreten, so dass ein Test auf Gleichheit nicht angemessen ist. `check_expect_within` testet daher nicht auf Gleichheit mit dem erwarteten Wert, sondern ob die beiden Werte innerhalb einer Fehlerschranke übereinstimmen.

Implementierungen von `area()`

- ▶ Die Definition einer Methodensignatur für Methode m im Interface **erzwingt** die Implementierung von m mit dieser Signatur in **allen** implementierenden Klassen.

⇒ `area()` in `Dot`:

```
public double area() {  
    return 0;  
}
```

⇒ `area()` in `Square`:

```
public double area() {  
    return this.size * this.size;  
}
```

⇒ `area()` in `Circle`:

```
public double area() {  
    return this.radius * this.radius * Math.PI;  
}
```

Anwendungsbeispiele für `distTo0()`

```
2 class ShapeExamples {
3     IShape dot = new Dot (new CartPt (4,3));
4     IShape squ = new Square (new CartPt (4,3), 3);
5     IShape cir = new Circle (new CartPt (12,5), 2);
6     // tests
14    boolean testDot2 = check dot.distTo0() expect 5.0 within 0.01;
15    boolean testSqu2 = check squ.distTo0() expect 5.0 within 0.01;
16    boolean testCir2 = check cir.distTo0() expect 11.0 within 0.01;
24    // constructor
25    ShapeExamples () {}
26 }
```

Analyse von `distTo0()`

- ▶ Der Abstand eines Dot zum Ursprung ist der **Abstand seines loc Feldes zum Ursprung**.
- ▶ Der Abstand eines Square zum Ursprung ist der **Abstand seines Referenzpunktes zum Ursprung**.
- ▶ Der Abstand eines Circle zum Ursprung ist der **Abstand seines Mittelpunktes zum Ursprung abzüglich des Radius**. (Ausnahme: der Kreis enthält den Ursprung.)

Analyse von `distTo0()`

- ▶ Der Abstand eines Dot zum Ursprung ist der **Abstand seines `loc` Feldes zum Ursprung**.
 - ▶ Der Abstand eines Square zum Ursprung ist der **Abstand seines Referenzpunktes zum Ursprung**.
 - ▶ Der Abstand eines Circle zum Ursprung ist der **Abstand seines Mittelpunktes zum Ursprung abzüglich des Radius**. (Ausnahme: der Kreis enthält den Ursprung.)
- ⇒ Hilfsmethode “Abstand zum Ursprung” auf `CartPt` sinnvoll:

```
class CartPt {  
    ...  
    double distTo0() {  
        return Math.sqrt(this.x * this.x + this.y * this.y);  
    }  
}
```

Implementierungen von `distTo0()`

⇒ `distTo0()` in `Dot`:

```
public double double() {  
    return this.loc.distTo0();  
}
```

⇒ `distTo0()` in `Square`:

```
public double distTo0() {  
    return this.loc.distTo0;  
}
```

⇒ `distTo0()` in `Circle`:

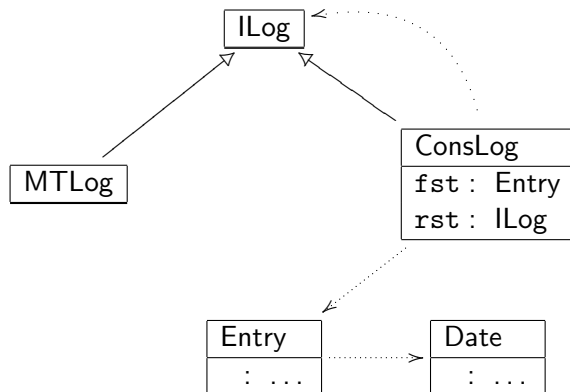
```
public double distTo0() {  
    return this.loc.distTo0() - this.radius;  
}
```

Entwurf von Methoden auf Vereinigungen von Klassen

1. Erkläre den Zweck der Methode (Kommentar) und definiere die Methodensignatur. **Füge die Methodensignatur jeder implementierenden Klasse hinzu.**
2. Gib Beispiele für die Verwendung der Methode **in jeder Variante.**
3. Fülle den Rumpf der Methode gemäß dem (bekannten) Muster
 - ▶ **this** und die Felder **this.feldname** dürfen vorkommen
 - ▶ alle Parameter dürfen vorkommen
 - ▶ alle Methodenaufrufe auf untergeordneten Objekten dürfen vorkommen
4. Schreibe den Rumpf der Methode **in jeder Variante.**
5. Definiere die Beispiele als Tests.

Methoden auf rekursiven Klassen

Erinnerung: das Laftagebuch



- Ziel: Definiere Methoden auf ILog

Muster: Methoden für ILog

- ▶ gewünschte Methodensignatur in **interface** ILog

```
// Zweck der Methode  
public ttt mmm();
```

- ▶ Implementierungsschablone in MTLLog **implements** ILog

```
public ttt mmm() { ... }
```

- ▶ Implementierungsschablone in ConsLog **implements** ILog

```
public ttt mmm() {  
    ... this.fst.mmmHelper() ...  
    ... this.rst.mmm() ... // rekursiver Aufruf, liefert Ergebnis auf rst  
}
```

- ▶ ggf. Hilfsmethode in Entry

```
uuu mmmHelper() { ... this.d.mmmHelper() ... this.distance ... }
```

- ▶ ggf. Hilfsmethode in Date

```
vvv mmmHelper() { ... this.day ... }
```

Beispiel: Gelaufene Strecke

Ermittle aus dem Lauftagebuch die insgesamt gelaufenen Kilometer.

▶ in ILog

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

- ▶ Die Gesamtkilometerzahl für ein leeres Tagebuch ist 0.
- ▶ Die Gesamtkilometerzahl für ein nicht-leeres Tagebuch ist die gelaufene Distanz plus die Gesamtkilometerzahl des restlichen Tagebuchs.

Beispiel: Gelaufene Strecke

Implementierungen

- ▶ Die Gesamtkilometerzahl für ein leeres Tagebuch ist 0.

- ▶ Die Gesamtkilometerzahl für ein nicht-leeres Tagebuch ist die gelaufene Distanz plus die Gesamtkilometerzahl des restlichen Tagebuchs.

Beispiel: Gelaufene Strecke

Implementierungen

- ▶ Die Gesamtkilometerzahl für ein leeres Tagebuch ist 0.
- ▶ in MTLLog

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

- ▶ Die Gesamtkilometerzahl für ein nicht-leeres Tagebuch ist die gelaufene Distanz plus die Gesamtkilometerzahl des restlichen Tagebuchs.

Beispiel: Gelaufene Strecke

Implementierungen

- ▶ Die Gesamtkilometerzahl für ein leeres Tagebuch ist 0.
- ▶ in MTLog

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

- ▶ Die Gesamtkilometerzahl für ein nicht-leeres Tagebuch ist die gelaufene Distanz plus die Gesamtkilometerzahl des restlichen Tagebuchs.
- ▶ in ConsLog

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

Zusammenfassung

Arrangements von Mustern und Klassen

- ▶ Einfache Klassen:
einfache Methoden, die nur die Felder der eigenen Klasse verwenden
- ▶ Zusammengesetzte Klassen:
Methoden verwenden die eigenen Felder, sowie Methoden und Felder der enthaltenen Objekte
- ▶ Vereinigung von Klassen:
Methoden im Interface müssen in jeder Variante definiert werden
- ▶ Rekursive Klassen:
Beim Entwurf der Methoden wird angenommen, dass die (rekursiven) Methodenaufrufe auf dem Start-Interface bereits das richtige Ergebnis liefern.