Inhalt

Zahlen

int ganze Zahlen im Bereich $[-2^{31}, 2^{31} - 1]$,

- exakt, alle Rechenoperationen modulo 2^{32}
- Wertebereich: zwischen -2147483648 und 2147483647
- Literale: `42` `0` `-16384` `+911`

double Gleitkommazahlen mit doppelter Genauigkeit (64 Bit)
Vgl. Standard IEEE 754-1985, Vorlesung Technische Informatik

- inexakt, alle Rechenoperationen werden gerundet
- Wertebereich: etwa $\pm 1.7976931348623157 \times 10^{308}$
- Literale: `3.14159265` `-.14142`
  `+6.02214179E+23` `8.854E-12`

Wahrheitswerte und Zeichenfolgen

boolean Wahrheitswerte

- Wertebereich: nur `true` und `false`
- Literale: `true` `false`

String Zeichenfolgen (Strings)

- Wertebereich: alle endlichen Folgen von Zeichen (bis zu einer unspezifizierten Maximallänge)
- Literale: `""` `"Sushi"` `"küçük"` `"#§$&???"`

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
  eObject`.`*methode*`(`*eArg*`, ...)`
  - *eObject* Ausdruck, dessen Wert ein Objekt ist
  - *methode* Name einer Methode dieses Objektes
  - *eArg* Argumentausdruck für die Methode
- Schachtelung möglich (Auswertung von links nach rechts)
  `"mai".concat("karenda").length()`

Einige String Methoden

Methode	Parameter	Ergebnis	Beispiel
`length`	()	int	`"xy".length()`
`concat`	(String)	String	`"xy".concat ("zw")`
`toLowerCase`	()	String	`"XyZ".toLowerCase()`
`toUpperCase`	()	String	`"XyZ".toUpperCase()`
`equals`	(String)	boolean	`"XyZ".equals ("xYz")`
`endsWith`	(String)	boolean	`"XyZ".endsWith("yZ")`
`startsWith`	(String)	boolean	`"XyZ".startsWith("Xy")`

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

Einfache Klassen

- Primitive Datentypen reichen nicht für alle Anwendungen aus
- Beispiel:

  . . . Das Programm soll die Buchhaltung für einen Teegroßhändler unterstützen. Die Quittung für eine Lieferung beinhaltet die Teesorte, den Preis (in Euro pro kg) und das Gewicht der Lieferung (in kg). . . .

- Beispielquittungen
  - 100kg Darjeeling zu 40.10 EUR
  - 150kg Assam zu 27.90 EUR
  - 140kg Ceylon zu 27.90 EUR

Modellierung einer Teelieferung

```
2  // Repräsentation einer Rechnung für eine Teelieferung
3  class Tea {
6      String kind; // Teesorte
7      int price; // in Eurocent pro kg
8      int weight; // in kg
11     Tea(String kind, int price, int weight) {
12         this.kind = kind;
13         this.price = price;
14         this.weight = weight;
15     }
34 }
```

- Vollständige *Klassendefinition*

Grundgerüst einer Klassendefinition

```
2 // Repräsentation einer Rechnung für eine Teelieferung
3 class Tea {
```

- Benennt den Klassentyp Tea
- Rumpf der Klasse spezifiziert
  - die Komponenten der *Objekte* vom Klassentyp
  - den *Konstruktor* des Klassentyps
  - das Verhalten der Objekten (später)

```
34 }
```

Felddeklarationen

```
6     String kind; // Teesorte
7     int price; // in Eurocent pro kg
8     int weight; // in kg
```

- Beschreibt die Komponenten: *Instanzvariable*, *Felder*, *Attribute*
- Beschreibung eines Felds
  - Typ des Feldes (String, int)
  - Name des Feldes (`kind`, `price`, `weight`)
- Kommentare: `//` bis Zeilenende

Konstruktordeklaration

```
11    Tea(String kind, int price, int weight) {
12        this.kind = kind;
13        this.price = price;
14        this.weight = weight;
15    }
```

- Beschreibt den *Konstruktor*: Funktion, die aus den Werten der Komponenten ein neues Objekt initialisiert
- Argumente des Konstruktors entsprechen den Feldern
- Rumpf des Konstruktors enthält Zuweisungen der Form

$$\textbf{this}.\mathit{feldname} = \mathit{feldname}$$

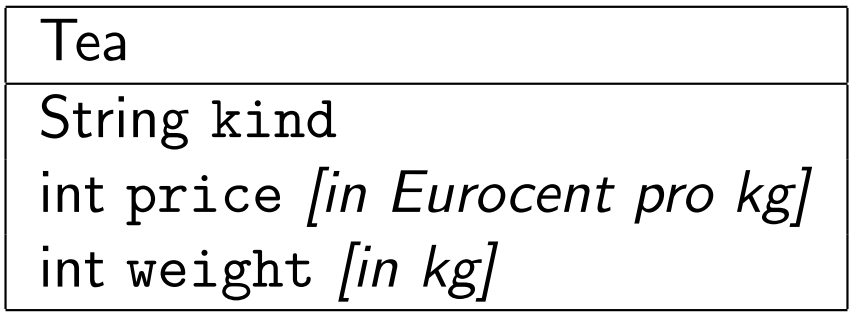
- **this** ist das Objekt, das gerade konstruiert wird
- **this**.*feldname* bezeichnet das entsprechende Feld des Objekts
- *feldname* bezeichnet den Wert des entsprechenden Konstruktorarguments

Beispiel für Teelieferungen

- 100kg Darjeeling zu 40.10 EUR
- 150kg Assam zu 27.90 EUR
- 140kg Ceylon zu 27.90 EUR

```
new Tea("Darjeeling", 4010, 100)
new Tea("Assam", 2790, 150)
new Tea("Ceylon", 2790, 140)
```

Klassendiagramm

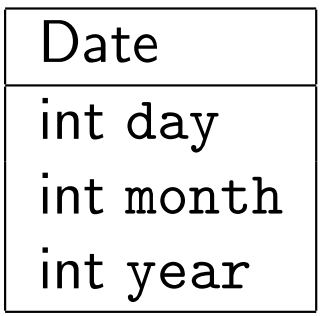


- Die Spezifikation einer Klasse kann auch als *Klassendiagramm* angegeben werden.
- Obere Abteilung: Name der Klasse
- Untere Abteilung: Felddeklarationen
- Anmerkung: Klassendiagramme werden in der Softwaretechnik verwendet. Sie sind im UML (Unified Modeling Language) Standard definiert. Sie sind nützliche Werkzeuge für die Datenmodellierung.

Beispiel: Datumsklasse

Ein Datumswert besteht aus Tag, Monat und Jahr.

- Drei Komponenten
- Jede Komponente kann durch int repräsentiert werden.
- Klassendiagramm dazu



Beispiel: Implementierung der Datumsklasse

```
1  // Ein Datum
2  class Date {
3    int day;
4    int month;
5    int year;
6
7    Date(int day, int month, int year) {
8      this.day = day;
9      this.month = month;
10     this.year = year;
11   }
12 }
```

Beispiel: Verwendung der Datumsklasse

- Korrekte Beispiele

```
new Date (30, 9, 2007) // 30. September 2007
new Date (13, 4, 2003) // 13. April 2003
new Date ( 1, 10, 1999) // 1. Oktober 1999
```

- Aber auch sinnlose Date Objekte sind möglich

```
new Date (33, 88, 1600) // ???
new Date (−1, −4, 0) // ???
```

- Anmerkung: Das wird noch ausgeschlossen.

Beispiel: Billardkugeln

Die Position einer Billardkugel auf dem Tisch wird durch ihre X- und Y-Koordinaten beschrieben. Jede Billardkugel besitzt einen Radius, der durch die Spielregel vorgeschrieben ist.

- Drei Komponenten
- Jede Komponente kann durch int repräsentiert werden.
- Klassendiagramm dazu

Ball
int `x` int `y` int `RADIUS` = 5 *[durch Spielregel vorgegeben]*

Beispiel: Implementierung von Billardkugeln

```
1  // eine Billardkugel
2  class Ball {
3    int x;
4    int y;
5    int RADIUS = 5; // durch Spielregel vorgegeben
6
7    Ball(int x, int y) {
8      this.x = x;
9      this.y = y;
10   }
11 }
```

- Zeile 4 *initialisiert* das Feld `RADIUS` auf den Wert 5.
- Der Konstruktor (Zeile 6-9) nimmt kein `RADIUS`-Argument.
- Der Konstruktor darf das `RADIUS`-Feld nicht setzen.

$\Rightarrow$ Das `RADIUS`-Feld eines jeden Ball-Objekts hat den Wert 5 und kann vom Konstruktor nicht anders gesetzt werden.

Beispiel: Verwendung von Billardkugeln

new Ball (36, 45) // ==> *Ball(x = 36, y = 45, RADIUS = 5)*
new Ball (100, 3) // ==> *Ball(x = 100, y = 3, RADIUS = 5)*

- Auch sinnlose Werte sind möglich:
  - außerhalb des Tisches
  - negative Koordinaten

Zusammenfassung

- Eine Klasse spezifiziert einen zusammengesetzten Datentyp, den *Klassentyp*.
- Die zum Klassentyp gehörigen Werte sind die *Instanzen* bzw. *Objekte* der Klasse.
- Ein Objekt enthält die Werte der Komponenten in den *Instanzvariablen*.
- Werte vom Klassentyp C werden durch den Konstruktoraufruf

$$\textbf{new } \mathrm{C}(v_1, \ldots, v_n)$$

gebildet, wobei $v_1, \ldots, v_n$ die Werte der Instanzvariablen sind.

Erstellen einer Klasse

1. Studiere die Problembeschreibung. Identifiziere die darin beschriebenen Objekte und ihre Attribute und schreibe sie in Form eines Klassendiagramms.
2. Übersetze das Klassendiagramm in eine Klassendefinition. Füge einen Kommentar hinzu, der den Zweck der Klasse erklärt. (Mechanisch, außer für Felder mit fest vorgegebenen Werten)
3. Repräsentiere einige Beispiele durch Objekte. Erstelle Objekte und stelle fest, ob sie Beispielobjekten entsprechen. Notiere auftretende Probleme als Kommentare in der Klassendefinition.

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

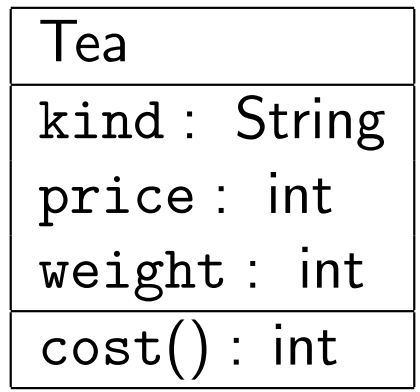
Methodendefinition

```
// 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



- Dritte Abteilung enthält die Kopfzeilen der Methoden
  Signaturen von Methoden

Entwicklung der Methode `cost`

- Jede Methode kann auf ihr zugehöriges Objekt über die Variable `this` zugreifen

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

Entwicklung der Methode `cost`

- Jede Methode kann auf ihr zugehöriges Objekt über die Variable `this` zugreifen

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

- Zugriff auf die Felder des Objekts 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 zugehöriges Objekt über die Variable `this` zugreifen

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

- Zugriff auf die Felder des Objekts 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  import static org.junit.Assert.*;
2  import org.junit.Test;
3  // test cases for Tea
4  public class TeaTests {
5    @Test
6    public void testCost() {
7      assertEquals(150*1590, new Tea("Assam", 1590, 150).cost());
8      assertEquals(220*2790, new Tea("Darjeeling", 2790, 220).cost());
9      assertEquals(130*1590, new Tea("Ceylon", 1590, 130).cost());
10   }
11 }
```

- Separate Testklasse, erzeugt mit *New* → *JUnit Test Case*
- Die `import` Statements integrieren den Testrahmen
- Testmethoden werden mit `@Test` annotiert
- Die `assert` Funktion vergleicht den erwarteten Wert eines Ausdrucks mit dem tatsächlichen Wert.

Methoden mit Argumenten

Primitive Datentypen

Der Teelieferant sucht nach billigen Angeboten, 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:

```
@Test
public void testCheaperThan() {
  assertFalse(new Tea ("Earl Grey", 3945, 75).cheaperThan (2000));
  assertTrue(new Tea ("Ceylon", 1590, 400).cheaperThan (2000));
}
```

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);
@Test
public void testLighterThan() {
  assertFalse(t1.lighterThan (new Tea ("Earl Grey", 3945, 25)));
  assertTrue(t2.lighterThan (new Tea ("Assam", 1590, 500)));
}
```

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

```
static final double DELTA = 1e-5;
@Test
public void testInterest() {
  assertEquals(5880.0, new Deposit ("Dieter", 120000).interest(), DELTA);
  assertEquals(25000.0, new Deposit ("Verona", 500000).interest(), DELTA);
  assertEquals(56100.0, new Deposit ("Franjo", 1100000).interest(), DELTA);
}
```

Bemerkung

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

Bedingte Anweisung

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

Bedingte Anweisung

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

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

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

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

Bedingte Anweisung geschachtelt

- Die bedingte Anweisung ist selbst eine Anweisung, also ist auch Schachtelung möglich.

```
if (bedindung1) {
  // ausgeführt, falls bedingung1 wahr
  return ausdruck1;
} else {
  if (bedindung2) {
    // ausgeführt, falls bedingung1 falsch und bedingung2 wahr
    return ausdruck2;
  } else {
    // ausgeführt, falls bedingung1 und bedingung2 beide falsch
    return ausdruck3;
  }
}
```

$\Rightarrow$ Passt genau zu den Anforderungen an `interest()`!

Bedingte Anweisung für Zinsberechnung

- Einsetzen der Bedingungen und Auflisten der verfügbaren Variablen:

```
// 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: Kapsele die Zinsformel und die 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.