TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Lernziele

- Klassen in Java deklarieren
  - Attribute
  - Konstruktor
  - (Instanz-)Methoden
- Delegation als Programmierpattern anwenden
- Aus einer Problembeschreibung ein objektorientiertes Modell und Implementierung ableiten

Objekte und Klassen in Java

Klassen in Java

Eine einfache **Klassendeklaration** in Java hat folgende Bestandteile:

```
class Person {                          Klassenname
  String name;                          Attribut
  Person(String n) {                    }
    this.name = n;                      } Konstruktor
  }
  String getName() {                    }
    return this.name;                   } Methode
  }
}
```

- In der Regel speichern wir den Code für die Deklaration einer Klasse in einer gleichnamigen `.java` - Datei ab.
  Im Beispiel: `Person.java`
- Klassennamen beginnen in Java nach Konvention mit einem Großbuchstaben.

Deklaration von Klassen: Klassenname

- Der Klassenname wird gleichzeitig als *Typname* für die Objekte dieser Klasse verwendet (*Klassentyp*).
- Er kann im Programm dann wie elementare Typen (`int`, `double`, usw.) für die Deklaration von lokalen Variablen, Parametern und Rückgabewerten verwendet werden.

Beispiel:

```
Person m = new Person("Klaus Meyer");
```

Deklaration von Klassen: Attribute

Innerhalb einer Klasse K können Attribute deklariert werden.

Syntax in Java:

```
AttributDeklaration →
      Typ « Bezeichner » ;
    | Typ « Bezeichner » = Ausdruck ;
```

Für jedes in K deklarierte Attribut vom Typ T besitzen die Objekte der Klasse K eine **objektlokale** Variable vom Typ T. Diese objektlokalen Variablen nennt man häufig auch **Instanzvariablen**.

Die Lebensdauer der Instanzvariablen entspricht der Lebensdauer des Objekts.

Beispiel: Attribute

```
class Mensch {
  String name;
  Mensch vater;
  Mensch mutter;
  boolean lebt;
}
```

Deklaration von Klassen: Methoden I

- Ein Java-Objekt kann genau auf die Nachrichten reagieren, für die Methoden in seiner Klasse deklariert sind oder für die es Methoden geerbt hat ($\Rightarrow$ Vorlesung zu "Vererbung").
- Methodendeklarationen bestehen aus einer Signatur und einem Methodenrumpf. Syntaktisch sind sie wie Prozedurdeklarationen aufgebaut.
- Innerhalb der Klassendeklaration können beliebig viele *Methoden* deklariert werden.

Deklaration von Klassen: Methoden II

Syntax in Java:

MethodenDeklaration →
  TypOderVoid ≪ *Bezeichner* ≫ (FormaleParameter) Anweisungsblock

```
// Zugriff auf impliziten Parameter in Methoden:
Ausdruck → this
```

Beispiel:

```
class Mensch {
  // ...
  String getName() {
    return this.name;
  }
}
```

Außer den deklarierten Parametern besitzt jede Methode m einen weiteren, sogenannten **impliziten Parameter** vom Typ der Klasse, in der m deklariert wurde. Dieser Parameter wird im Methodenrumpf mit `this` bezeichnet.

Konstruktoren I

Konstruktoren erzeugen und initialisieren Objekte.

Syntax in Java:

KonstruktorDeklaration →
  ≪ *Bezeichner* ≫ (FormaleParameter) Anweisungsblock

Beispiele:

```
class Farbe {
  // Attribute:
  int rot;
  int gruen;
  int blau;
  // Konstruktor:
  Farbe(int rot, int gruen, int blau) {
    this.rot = rot;
    this.gruen = gruen;
    this.blau = blau;
  }
}
```

Konstruktoren II

- Konstruktoren haben den gleichen Namen wie die Klasse, in der sie deklariert sind.
- Konstruktoren liefern als Ergebnis das neu erzeugte Objekt zurück, genauer: eine Referenz auf dieses Objekt.
- Beim Start der Ausführung eines Konstruktors ist das zugehörige Objekt bereits erzeugt, seine Attribute jedoch nur mit Standardwerten initialisiert.

Initialisierung I

Attribute (wie auch lokale Variablen) können direkt an ihrer Deklarationsstelle initialisiert werden.

```
class C {
  int a;
  C() {
    a = 78;
  }
}
```

```
class C {
  int a = 78;
  C() {}
}
```

```
class C {
  int a = 78;
}
```

- Die Initialisierung von Attributen erfolgt vor dem Eintritt in den Konstruktor*rumpf*.
- Falls kein Konstruktor deklariert ist, erhält die Klasse einen leeren Standard-Konstruktor.

Initialisierung II

Falls **ein** Konstruktor definiert ist, gibt es **keinen** Standard-Konstruktor.

```
class Person {
  String name;
  Person(String n) {
    this.name = n;
  }
}

...
new Person("Hugo")

// nicht erlaubt!
new Person()
```

```
class Hund {
  // Alle Hunde heissen
  Bello!
  String name = "Bello";
}

...
new Hund()

// nicht erlaubt!
new Hund("Walid")
```

Initialisierung III

In Java können Attribute und Variablen durch das Schlüsselwort `final` als **unveränderlich** deklariert werden.

In diesem Fall *muss* die Initialisierung an der Deklarationsstelle erfolgen oder in geeigneter Weise im Konstruktor.

```
class Hund {
  final String name;
  Hund() {
    this.name = "Bello";
  }
}
```

```
class Hund {
  final String name = "Bello";
}
```

Attributzugriff I

Syntax in Java:

Auf Instanzvariablen von Objekten kann mit Ausdrücken folgender Form zugegriffen werden:

`Ausdruck` → `Ausdruck` . $\ll Bezeichner \gg$

`Zuweisung` → `Ausdruck` . $\ll Bezeichner \gg$ = `Ausdruck`;

Semantik:

Werte den Ausdruck aus; dieser muss eine Referenz liefern.

Liefert dieser `null`, löse eine `NullPointerException` aus.

Andernfalls liefert er die Referenz auf ein Objekt X; in dem Fall liefert der gesamte Ausdruck die Instanzvariable von X zum angegebenen Attribut (L-Wert) oder deren Wert (R-Wert).

Attributzugriff II

Der implizite Methodenparameter `this` kann beim Zugriff auf ein Attribut `a` weggelassen werden, d.h.

```
a
```

ist gleichbedeutend mit

```
this.a
```

innerhalb von Klassen, in denen `a` deklariert ist.

```
class Person {
  String name;
  Person(String n) {
    this.name = n;
  }
  String getName() {
    return this.name;
  }
}
```

```
class Person {
  String name;
  Person(String n) {
    name = n;
  }
  String getName() {
    return name;
  }
}
```

Kaskadierende Attributzugriffe

```
class Autor {
  String name;
  int    geburtsjahr;
}


class Buch {
  Autor  autor;
  String titel;

   void printInfo() {
      StdOut.println("Titel:" + this.titel);
      StdOut.println("Autor:" + this.autor.name);
   }
}
```

Bemerkung

Abkürzende Notation:

Wie beim Attributzugriff kann auch beim Methodenaufruf der implizite Methodenparameter `this` weggelassen werden, also `m(...)` statt `this.m(...)`.

Beispiel: Methodenaufrufe I

```
class Mensch {
  Mensch vater;
  Mensch mutter;
  String name;

  Mensch getOpa (boolean mutterseits) {
    if (mutterseits) {
      return mutter.vater;
    } else {
      return vater.vater;
    }
  }

  Mensch(String name, Mensch vater, Mensch mutter) {
    this.name = name;
    this.vater = vater;
    this.mutter = mutter;
  }
}
```

Beispiel: Methodenaufrufe II

```
// Beispiel zur Verwendung:
public class MenschTest {
  @Test
  public void ermittleOpa() {
    Mensch gundula = new Mensch("Gundula", null, null);
    Mensch fritz = new Mensch("Fritz", null, null);
    Mensch erika = new Mensch("Erika", null, null);
    Mensch daniel = new Mensch("Daniel", null, null);
    Mensch christine = new Mensch("Christine", fritz, gundula);
    Mensch bob = new Mensch("Bob", daniel, erika);
    Mensch alice = new Mensch("Alice", bob, christine);

    Mensch opaV = alice.getOpa(false);
    assertEquals(daniel, opaV);

    String opaMName = alice.getOpa(true).name;
    assertEquals("Fritz", opaMName);
  }
}
```

Klassendiagramme

Eine Klasse kann durch ein **Klassendiagramm** spezifiziert werden.

- Klassendiagramme dienen hauptsächlich der Datenmodellierung.
- Sie sind im UML (Unified Modeling Language) Standard definiert.
- Sie enthalten den *Name* der Klasse, *Attribute* mit Typ und *Methoden* mit Signaturen.

Klasse:

Mensch
vater : Mensch mutter : Mensch name : String
getOpa(boolean mutterseits) : Mensch

Objektdiagramme

Objekt:

: Mensch
vater = null mutter = null name = "Gundula"

- Enthalten (aktuelle) Werte der Attribute
- Methoden werden nicht gelistet, da diese für alle Objekte einer Klasse identisch sind

Fallstudie: Code-Tagebuch

Erstellen einer Klasse in 4 Schritten

1. Studiere die Problembeschreibung. Identifiziere die darin beschriebenen Objekte und ihre Attribute und Methoden.
2. Erstelle entsprechende Klassendiagramme.
3. Übersetze die Klassendiagramm in eine Klassendefinition. Füge einen Kommentar hinzu, der den Zweck der Klasse erklärt.
4. Repräsentiere einige Beispiele durch Objekte. Erstelle Objekte und stelle fest, ob sie Beispielobjekten entsprechen.

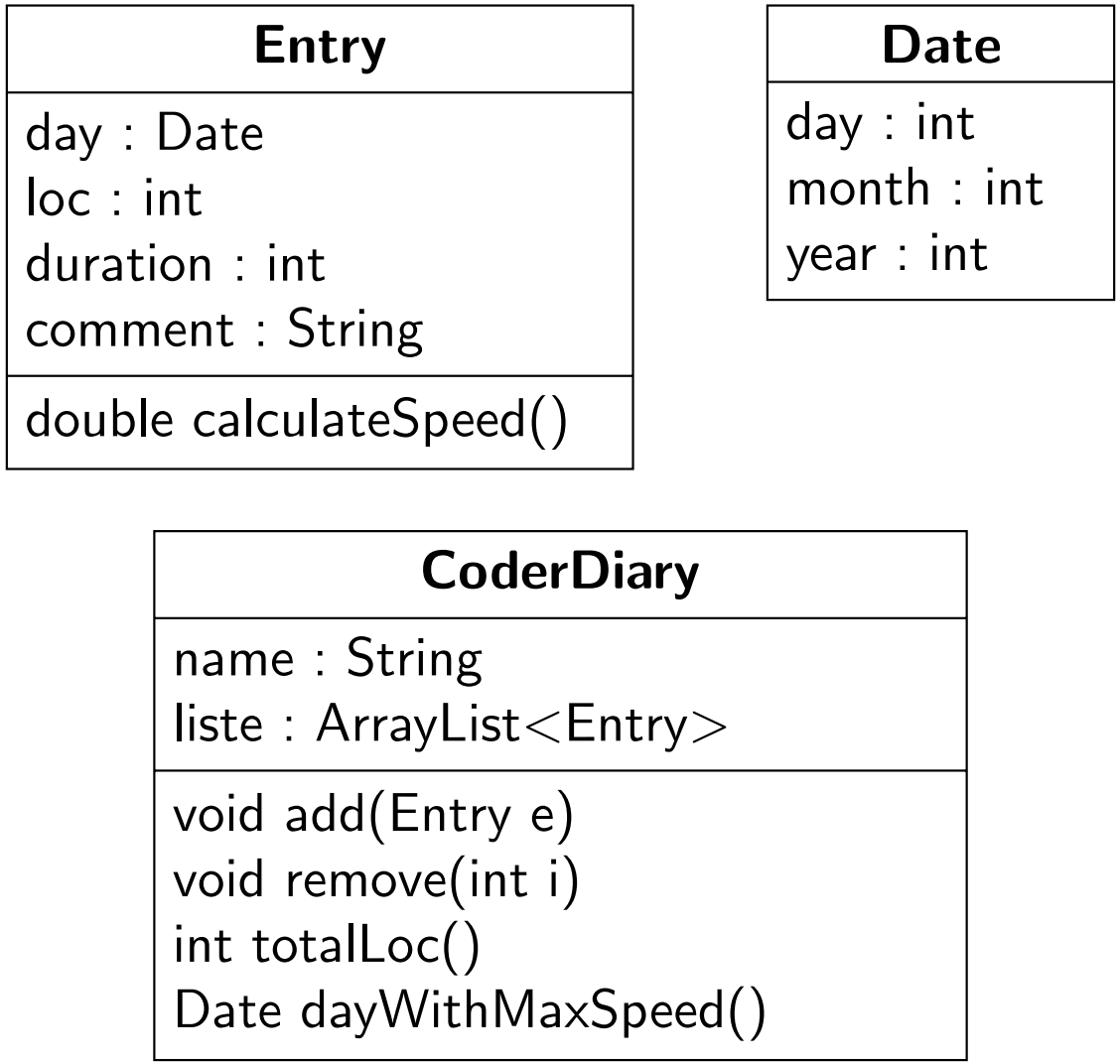
Aufgabe

Entwickle ein Programm, mit dem man ein persönliches Code-Tagebuch führen kann.

- *Es soll möglich sein, Einträge im Code-Tagebuch hinzuzufügen und zu entfernen, sich alle Einträge anzeigen zu lassen, die Gesamtzahl der geschriebenen Codezeilen zu ermitteln und den Tag mit der höchsten Geschwindigkeit bei der Codegenerierung zu ermitteln.*
- *Ein Eintrag besteht aus dem Datum, der Anzahl an geschriebenen Zeilen Code, der Dauer und einem Kommentar. Für einen Eintrag soll man außerdem die durchschnittliche Anzahl an Zeilen Code pro Session ermitteln.*
- *Ein Datum besteht aus Tag, Monat und Jahr.*

- Substantive liefern Hinweise auf Klassen oder Attribute
- Verben liefern Hinweise für Methoden

Klassendiagramme



Implementierung: Eintrag

```
// Repraesentiert einen Eintrag im Lauftagebuch
class Entry{
    Date day;
    int loc;          // lines of code
    int duration;     // in min
    String comment;

    Entry(Date day, int loc, int duration, String comment) {
        this.day = day;
        this.loc = loc;
        this.duration = duration;
        this.comment = comment;
    }

    // Durchschnittsgeschwindigkeit LOC/h bei der Codeerzeugung
    double calculateSpeed() {
      return 0.0; // TODO
    }
}
```

Implementierung: Datum

```
// Datum mit Tag, Monat, Jahr
class Date {
    int day;
    int month;
    int year;

    Date(int day, int month, int year) {
      this.day = day;
      this.month = month;
      this.year = year;
    }
}
```

Beispielobjekte

- Beispieleinträge
  - am 2. November 2017, 756 LOC in 210 Minuten, Initiales Design
  - am 8. November 2017, 140 LOC in 20 Minuten, Tests
  - am 11. November 2017, 8 LOC in 75 Minuten, Debugging
- ...als Objekte in einem Ausdruck

```
diary.add(new Entry(new Date(2,11,2017),765,210,"
  Initiales Design"));
diary.add(new Entry(new Date(8,11,2017),140,20,"Tests"));
diary.add(new Entry(new Date(10,11,2017),8,75,"Debugging"
  ));
```

- ...in zwei Schritten mit Hilfsdefinition

```
Date d1 = new Date(2,11,2017);
Entry e1 = new Entry(d1,765,210,"Initiales Design"));
```

Implementierung: Geschwindigkeit

```
class Entry {
    Date d;
    int loc; // lines of code
    int duration;    // in min
    String comment;
    ...
    // Durchschnittsgeschwindigkeit LOC/h bei der
    Codeerzeugung
    double calculateSpeed() {
        double h = duration / 60.0;
        return loc / h;
    }
}
```

Darstellung als String I

```
class Date {
    int day;
    int month;
    int year;
    ...
    public String toString() {
        return day + "." + month + "." + year;
    }
}
```

- Alle Referenztypen in Java haben eine Methode `toString()`.
- Man kann eigene String-Repräsentationen definieren. Diese müssen als `public` deklariert werden (siehe Kapitel "Vererbung").
- Die String-Repräsentation enthält üblicherweise Informationen zu den Attributwerten.

Darstellung als String II

- `toString()` wird (automatisch) aufgerufen, wenn das Objekt in einem Kontext verwendet wird, das einen String erwartet.

```
Date d = new Date (5,6,2015);
StdOut.println(d.toString());
StdOut.println(d); //alternativ
```

Delegation

```
class Entry {
  Date d;
  int loc; // lines of code
  int duration;  // in min
  String comment;
  ...
  public String toString() {
        return d.toString() + ": " + loc + " LOC in "
            + duration + " min; " + comment;
  }
}
```

- Die Implementierung von `toString()` in `Entry` verwendet die Implementierung dieser Methode in `Date`.
- Dieses Muster ist typisch für Klassen, deren Objekte (u.a.) aus Objekten anderer Klassen zusammengesetzt sind (sogenannte **Aggregate** oder **Kompositionen**).

Die Klasse CoderDiary

- Noch offen: Verwaltung **beliebiger Anzahl** von `Entry`-Objekten
- Variante 1: Array mit `Entry`-Objekten
- Variante 2: **Liste** von `Entry`-Objekten

	API von `java.util.ArrayList`	(Auszug)
	`ArrayList<E>()`	Konstruktor für Liste mit Elementen vom Typ `E`
`int`	`size()`	Anzahl der Elemente
`boolean`	`add(E elem)`	Fügt das Element am Ende der Liste an und liefert true zurück
`E`	`get(int i)`	Liefert das Element an Position `i`
`E`	`remove(int i)`	Entfernt das Element an Position `i` und gibt dieses zurück

Erstellen von List-Objekten und Verwaltung von Elementen

```
class CoderDiary{
    String name;
    ArrayList<Entry> liste;

    CoderDiary(String name) {
        this.name = name;
        this.liste = new ArrayList<Entry>();
    }
    // Fuegt einen Eintrag am Ende der Liste hinzu
    void add(Entry e) {
      this.liste.add(e);
    }

    // Entfernt den Eintrag an Position i
    void remove(int i) {
      this.liste.remove(i);
    }
```

Iterieren über Elemente einer Liste

```
// Variante: Zaehlschleife
public String toString() {
    String result = "Code-Tagebuch von " + name + "\n";

    for (int i = 0; i < this.liste.size(); i++) {
        Entry e = this.liste.get(i);
        result += i + ". " + e.toString() + "\n";
    }

    return result;
}

// Variante: foreach-Schleife
  int totalLoc() {
    int sum = 0;
    for(Entry e : this.liste) {
      sum += e.loc;
    }
    return sum;
}
```

Die Anwendung CoderDiaryApp I

App-Steuerung über Kommandozeile

exit	Beende die Ausführung der App.
add	Füge einen Eintrag hinzu; dazu muss der Nutzer Informationen zu Datum, LOC, Dauer und Kommentar eingeben.
remove i	Entferne den Eintrag an Position i.
show	Zeige alle Einträge an.
speed	Zeige den Tag mit der schnellsten durchschnittlichen Code-Erzeugungsrate.
loc	Zeige die Gesamtzahl der Codezeilen für alle Einträge an.

Die Anwendung CoderDiaryApp II

```
public class CoderDiaryApp {
  public static void main(String[] args) {
    // Abfrage des Namens
    StdOut.print("Name: ");
    String name = StdIn.readLine();
    // Erstellen eines CoderDiaries
    CoderDiary d = new CoderDiary(name);

    // Interaktionsschleife
    boolean exit = false;
    do {
      StdOut.print("Aktion (exit, add, remove, show, loc, speed): ");
      String action = StdIn.readString();
      switch (action) {
        case "exit":    exit = true; break;
        case "add" :    Entry e = readEntry(); d.add(e); break;
        case "remove" : int i = StdIn.readInt(); d.remove(i); break;
        case "show":    StdOut.println(d);  break;
        case "speed" :  StdOut.println(d.dayWithMaxSpeed()); break;
        case "loc" :    StdOut.println(d.totalLoc()); break;
        default:        StdOut.println("Aktion nicht bekannt!");
      }
    } while(!exit);
  } //...
```

Einlesen von Daten I

```
// Einlesen der Daten eines neuen Eintrags
public static Entry readEntry() {
    StdOut.print("Datum (dd.mm.yyyy): ");
    String date = StdIn.readString();
    StdOut.print("LOC:                ");
    int loc = StdIn.readInt();
    StdOut.print("Zeit:               ");
    int time = StdIn.readInt();
    StdOut.print("Kommentar:          ");
    String comment = StdIn.readString();
    return
      new Entry(new Date(date), loc, time, comment);
}
```

Einlesen von Daten II

Ergänzen der Klasse `Date` um weiteren Konstruktor:

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

  Date(int day, int month, int year) {
    this.day = day;
    this.month = month;
    this.year = year;
  }

  // Alternativer Konstruktor aus String (dd.mm.yyyy)
  Date(String s){
    this.day   = Integer.valueOf(s.substring(0,2));
    this.month = Integer.valueOf(s.substring(3,5));
    this.year  = Integer.valueOf(s.substring(6,10));
  }
```

Ausblick

Die Anwendung dient uns hier als Fallstudie zum objekt-orientierten Modellieren. Sie hat aber noch einige Schwächen:

- Da die Einträge nicht persistent in eine Datei o.ä. geschrieben werden, stehen die Einträge nach Beenden der Anwendung nicht mehr zur Verfügung.
- Bei Datumswerte wird nicht überprüft, ob sie tatsächlich ein gültiges Datum repräsentieren.