Programmieren in Java

Vorlesung 01: Einfache Klassen

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2015

Inhalt

Einführung

Java

- Eine Programmiersprache, die zusammengesetzte Daten in Form von Objekten unterstützt.
- Objekte werden hierarchisch in Klassen organisiert.
- Neben Daten enthalten Objekte Methoden, die Operationen auf den Objekten implementieren.

Einführung

Java

- Eine Programmiersprache, die zusammengesetzte Daten in Form von Objekten unterstützt.
- Objekte werden hierarchisch in Klassen organisiert.
- Neben Daten enthalten Objekte Methoden, die Operationen auf den Objekten implementieren.

Programmieren in Java

- Erstellen von Klassen
- Zuordnen von Attributen (Daten) und Operationen (Methoden)
- Entwurf von Operationen
- Kodieren in Java

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.

Fahrschein

Spezifikation

Ein Verkehrsunternehmen möchte Einzelfahrscheine ausgeben. Der Einzelfahrschein hat eine Preisstufe (1, 2, 3), er ist entweder für Erwachsene oder für Kinder verwendbar und er kann entwertet werden. Der Entwerterstempel enthält Uhrzeit, Datum und Ort der Entwertung. Der Fahrgast kann den Fahrschein entwerten und auf seine Verwendbarkeit prüfen. Der Kontrolleur kann die Gültigkeit des Fahrscheins kontrollieren.

Fahrschein

Spezifikation

Ein Verkehrsunternehmen möchte Einzelfahrscheine ausgeben. Der Einzelfahrschein hat eine Preisstufe (1, 2, 3), er ist entweder für Erwachsene oder für Kinder verwendbar und er kann entwertet werden. Der Entwerterstempel enthält Uhrzeit, Datum und Ort der Entwertung. Der Fahrgast kann den Fahrschein entwerten und auf seine Verwendbarkeit prüfen. Der Kontrolleur kann die Gültigkeit des Fahrscheins kontrollieren.

- Substantive liefern Kandidaten für Klassen oder Attribute

Fahrschein

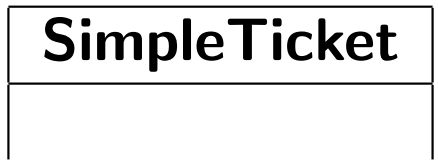
Spezifikation

Ein Verkehrsunternehmen möchte Einzelfahrscheine ausgeben. Der Einzelfahrschein hat eine Preisstufe (1, 2, 3), er ist entweder für Erwachsene oder für Kinder verwendbar und er kann entwertet werden. Der Entwerterstempel enthält Uhrzeit, Datum und Ort der Entwertung. Der Fahrgast kann den Fahrschein entwerten und auf seine Verwendbarkeit prüfen. Der Kontrolleur kann die Gültigkeit des Fahrscheins kontrollieren.

- Substantive liefern Kandidaten für Klassen oder Attribute

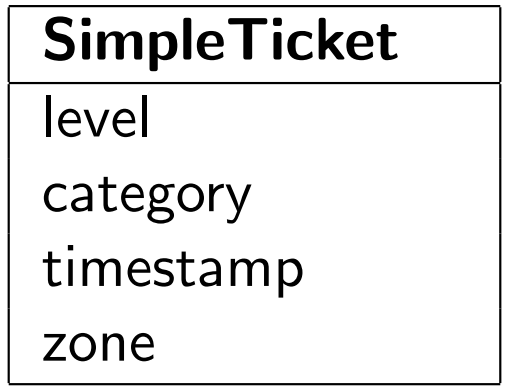
Provider | **SimpleTicket** | **Passenger** | **Conductor**

Klassendiagramm



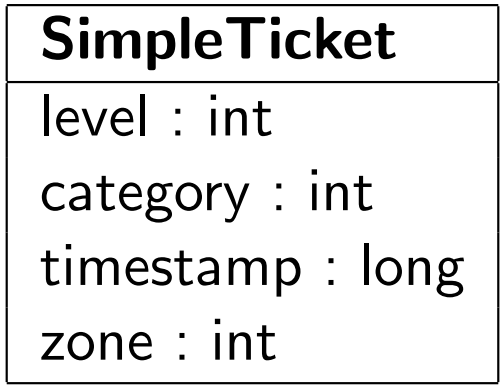
- Eine Klasse kann durch ein *Klassendiagramm* spezifiziert werden.
- Klassendiagramme dienen hauptsächlich der Datenmodellierung. Sie sind im UML (Unified Modeling Language) Standard definiert.
- Verpflichtend: *Name* der Klasse

Klassendiagramm



- Eine Klasse kann durch ein *Klassendiagramm* spezifiziert werden.
- Klassendiagramme dienen hauptsächlich der Datenmodellierung. Sie sind im UML (Unified Modeling Language) Standard definiert.
- Verpflichtend: *Name* der Klasse
- Untere Abteilung: *Attribute* der Klasse

Klassendiagramm



- Eine Klasse kann durch ein *Klassendiagramm* spezifiziert werden.
- Klassendiagramme dienen hauptsächlich der Datenmodellierung. Sie sind im UML (Unified Modeling Language) Standard definiert.
- Verpflichtend: *Name* der Klasse
- Untere Abteilung: *Attribute* der Klasse
- Attribute können mit Java *Typen* versehen werden

Klassendiagramm → Java: Klassen

```
1  package lesson_01;
2  /**
3   * Representation of a single ride ticket.
4   * @author thiemann
5   */
6  public class SimpleTicket {
68 }
```

- Paketdeklaration **package** lesson_01;
  Die Klasse **SimpleTicket** gehört zum Paket **lesson_01**.
- Klassenkommentar
  - Kurze Erläuterung der Klasse.
  - Metadaten (*Javadoc*)
- Klassendeklaration **public class** SimpleTicket
  - *Sichtbarkeit* **public**: Klasse überall verwendbar
  - Name der Klasse: Bezeichner, **immer** groß, CamelCase
- Dateiname = Klassenname: `SimpleTicket.java`

Klassendiagramm → Java: Attribute → Felder

```
9     // Preisstufe 1, 2, 3
10    private int level;
11    // Kind = 0, Erwachsener = 1
12    private int category;
13    // Zeitstempel der Entwertung (in Millisekunden seit 1.1.1970)
14    // nicht entwertet=0, ungültig=1
15    private long timestamp;
16    // Ort der Entwertung: Zone A=1, B=2, C=3
17    private int zone;
```

- Attribute → *Instanzvariable* bzw. *Felder*
- Felddeklaration
  - Sichtbarkeit: normalerweise **private**
    d.h. nur Objekte der gleichen Klasse dürfen direkt zugreifen
  - Typ
  - Bezeichner, **immer** klein, CamelCase, substantivisch
- Kommentar (darüber): Erläuterung, Einschränkung des Wertebereichs

Klassendiagramm → Java: Konstruktor

```
1  /**
2   * @param level Preisstufe 1, 2 oder 3.
3   * @param category Kind = 0, Erwachsener = 1.
4   */
5  public SimpleTicket(int level, int category) {
6    this.level = level;
7    this.category = category;
8  }
```

- Konstruktorkommentar: Erläuterung der Parameter (Javadoc)
- Konstruktormethode
  - Sichtbarkeit
  - Name = Klassenname
  - Parameterliste
  - Rumpf: Java Anweisungen; Ziel: Initialisierung der Felder
- Ausführung
  - wird nach Erzeugen eines neuen **SimpleTicket** Objekts aufgerufen
  - **this** bezieht sich auf das neue Objekt
  - alle Felder werden vorab auf Null (passend zum Typ) initialisiert

Einfache Klassen

- **SimpleTicket** ist eine *einfache Klasse*
- d.h., jedes Feld hat primitiven Datentyp

Einfache Klassen

- **SimpleTicket** ist eine *einfache Klasse*
- d.h., jedes Feld hat primitiven Datentyp

Primitive Datentypen in Java

- boolean, char, byte, short, int, long, float, double
- Einzelheiten siehe Tutorial über primitive Datentypen
  `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html`

Operationen

Operationen (Fahrschein)

Spezifikation

Ein Verkehrsunternehmen möchte Einzelfahrscheine ausgeben. Der Einzelfahrschein hat eine Preisstufe (1, 2, 3), er ist entweder für Erwachsene oder für Kinder verwendbar und er kann entwertet werden. Der Entwerterstempel enthält Uhrzeit, Datum und Ort der Entwertung. Der Fahrgast kann den Fahrschein entwerten und auf seine Verwendbarkeit prüfen. Der Kontrolleur kann die Gültigkeit des Fahrscheins kontrollieren.

Operationen (Fahrschein)

Spezifikation

Ein Verkehrsunternehmen möchte Einzelfahrscheine ausgeben. Der Einzelfahrschein hat eine Preisstufe (1, 2, 3), er ist entweder für Erwachsene oder für Kinder verwendbar und er kann entwertet werden. Der Entwerterstempel enthält Uhrzeit, Datum und Ort der Entwertung. Der Fahrgast kann den Fahrschein entwerten und auf seine Verwendbarkeit prüfen. Der Kontrolleur kann die Gültigkeit des Fahrscheins kontrollieren.

- Verben liefern Kandidaten für Operationen

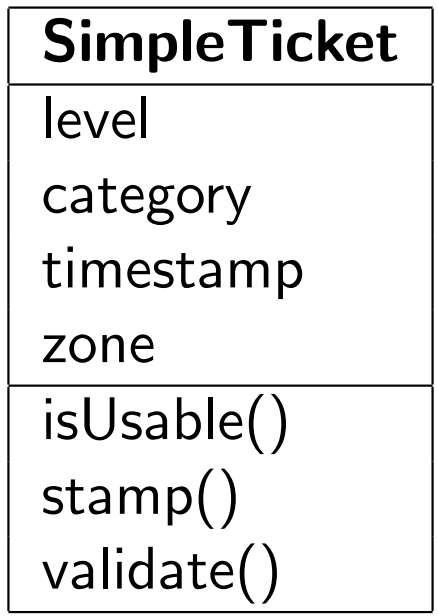
Operationen (Fahrschein)

Spezifikation

Ein Verkehrsunternehmen möchte Einzelfahrscheine ausgeben. Der Einzelfahrschein hat eine Preisstufe (1, 2, 3), er ist entweder für Erwachsene oder für Kinder verwendbar und er kann entwertet werden. Der Entwerterstempel enthält Uhrzeit, Datum und Ort der Entwertung. Der Fahrgast kann den Fahrschein entwerten und auf seine Verwendbarkeit prüfen. Der Kontrolleur kann die Gültigkeit des Fahrscheins kontrollieren.

- Verben liefern Kandidaten für Operationen
- Betrachte zunächst
  - Verwendbarkeit prüfen
  - entwerten
  - Gültigkeit kontrollieren

Klassendiagramm mit Operationen



- Operationen: dritte Abteilung der Klassenbox
- (außer dem Namen der Klasse ist alles optional)

Klassendiagramm mit Operationen und Typen

SimpleTicket
level : int category : int timestamp : long zone : int
isUsable() : boolean stamp(when : long, where : int) : void validate(int category, when : long, where : int) : boolean

- Operationen: dritte Abteilung der Klassenbox
- (außer dem Namen der Klasse ist alles optional)

Klassendiagramm → Java: Operationen → Methoden

isUsable()

```
/**
 * Check if this ticket is good for a ride.
 * @return true if the ticket can still be used
 */
public boolean isUsable() {
  // TODO: fill in method body
}
```

- Methodenkommentar (Javadoc)
  - Erläuterung der Funktion der Methode
  - Erklärung des Rückgabewerts
- Methodensignatur (**public** boolean isUsable())
  - Sichtbarkeit
  - Typ des Rückgabewertes
  - Bezeichner, **immer** klein, CamelCase, Tätigkeit
  - Parameterliste (hier: leer)

Klassendiagramm → Java: Operationen → Methoden

stamp()

```
/**
 * Stamp this ticket.
 * @param when time of validation (in millisec since 1.1.1970)
 * @param where location of validation (zone)
 */
public void stamp(long when, int where) {
  // TODO: fill in method body
}
```

- Methodenkommentar
  - Erläuterung der Parameter (Javadoc)
- Methodensignatur
  - Sichtbarkeit
  - Typ **void**: **kein** Rückgabewert
  - Parameterliste vgl. Konstruktor

Methodenentwurf für einfache Klassen

Ausfüllen der Methodenrümpfe

Rumpf der Methode

- Java Anweisungen
- verwendet werden dürfen
  - **alle** Felder der eigenen Klasse (ggf. qualifiziert durch **this**)
  - **alle** Methoden der eigenen Klasse
  - **public** Methoden von **public** Klassen
  - **alle** Konstruktoren der eigenen Klasse
- **return** definiert den Rückgabewert und beendet die Methode

Methodenentwurf

Beispiel: isUsable()

Spezifikation isUsable()

Fahrschein ist verwendbar, wenn er noch nicht abgestempelt worden ist.

Methodenentwurf

Beispiel: isUsable()

Spezifikation isUsable()

Fahrschein ist verwendbar, wenn er noch nicht abgestempelt worden ist.

Betroffene Felder

```
// Zeitstempel der Entwertung (in Millisekunden seit 1.1.1970)
// nicht entwertet=0, ungültig=1
private long timestamp;
```

Methodenentwurf

Beispiel: isUsable()

Spezifikation isUsable()

Fahrschein ist verwendbar, wenn er noch nicht abgestempelt worden ist.

Betroffene Felder

```
// Zeitstempel der Entwertung (in Millisekunden seit 1.1.1970)
// nicht entwertet=0, ungültig=1
private long timestamp;
```

Implementierung

```
/**
 * Check if this ticket is good for a ride.
 * @return true if the ticket can still be used
 */
public boolean isUsable() {
  return this.timestamp == 0;
}
```

Methodenentwurf — stamp()

Spezifikation stamp()

Stemple den Fahrschein mit aktueller Zeit und aktuellem Ort, die als Parameter übergeben werden. Mehrfaches Stempeln macht den Fahrschein ungültig.

Methodenentwurf — stamp()

Spezifikation stamp()

Stemple den Fahrschein mit aktueller Zeit und aktuellem Ort, die als Parameter übergeben werden. Mehrfaches Stempeln macht den Fahrschein ungültig.

Methodenhülse

```
/**
 * Stamp this ticket.
 * @param when time of validation (in millisec since 1.1.1970)
 * @param where location of validation (zone)
 */
public void stamp(long when, int where) {
  // TODO: fill in method body
}
```

Methodenentwurf

Betroffene Felder

```
// Zeitstempel der Entwertung (in Millisekunden seit 1.1.1970)
// nicht entwertet=0, ungültig=1
private long timestamp;
// Ort der Entwertung: Zone A=1, B=2, C=3
private int zone;
```

Methodenentwurf

Betroffene Felder

```
// Zeitstempel der Entwertung (in Millisekunden seit 1.1.1970)
// nicht entwertet=0, ungültig=1
private long timestamp;
// Ort der Entwertung: Zone A=1, B=2, C=3
private int zone;
```

Implementierung, Schritt 1

```
public void stamp(long when, int where) {
  if (this.isUsable()) {
    // remember stamp
  } else {
    // invalidate ticket
  }
}
```

Implementierung, Schritt 2

```
/**
 * Stamp this ticket.
 * @param when time of validation (in millisec since 1.1.1970)
 * @param where location of validation (zone)
 */
public void stamp(long when, int where) {
  if (this.isUsable()) {
    // remember stamp
    this.timestamp = when;
    this.zone = where;
  } else {
    // invalidate ticket
    this.timestamp = 1;
  }
}
```

- Bei **void** Methoden darf **return** weggelassen werden.

Methodenentwurf — validate()

Spezifikation validate()

Prüfe ob alle folgenden Bedingungen zutreffen.

1. der Fahrschein ist einmal gestempelt,
2. der Fahrschein ist für den Benutzer zulässig (ein Erwachsener sollte nicht mit einem Kinderfahrschein fahren),
3. die Benutzungsdauer des Fahrscheins ist nicht überschritten,
4. die Preisstufe passt zum Ort des Abstempelns und zum Ort der Kontrolle.

- Zu Punkt 3 siehe `http://www.vag-freiburg.de/tickets-tarife/hin-und-wieder-fahrer/einzelfahrschein.html`
- Zu Punkt 4 siehe `http://www.rvf.de/Tarifzonenplan.php`

Methodenentwurf — validate()

```
/**
 * Check validity of this ticket.
 * @param c category of passenger (child or adult)
 * @param t time of ticket check (millisec)
 * @param z location of ticket check (zone)
 * @return true iff the ticket is valid
 */
public boolean validate(int c, long t, int z) {
  // 1. stamped exactly once?
  boolean result = (this.timestamp != 0) && (this.timestamp != 1);
  // 2. passenger category less than ticket category?
  result = result && (c <= category);
  // 3. ticket expired?
  long timediff = t - timestamp;
  result = result && (timediff <= level * 60 * 60 * 1000);
  // 4. ticket used in valid zone?
  int leveldiff = Math.abs(zone - z);
  result = result && (leveldiff < level);
  return result;
}
```

Testen

- Ansatz: *Unit Testing*
- separate Testsuite für jede Programmkomponente (e.g., Klasse)
- separate Tests für jede Methode
- zwei Arten von Tests
  - synthetische Tests abgeleitet von der Spezifikation
    - Demonstration der normalen Funktion
    - Randfälle
  - Tests, die Fehler dokumentieren
    - Bug report

    → fehlschlagender Testfall

    → Bug fix

    → funktionierender Testfall

Testen mit Werkzeugen

- Werkzeug zum Unit Test: JUnit mit Eclipse Integration
- Zu jeder “normalen” Klasse erstelle eine Testklasse
- Für jede signifikante Methode erstelle mindestens eine synthetische Testmethode
- Für jeden nachvollziehbaren Bug erstelle eine Testmethode

Testklasse zu **SimpleTicket**

```
package lesson_01;

import static org.junit.Assert.*;
import org.junit.Test;

public class SimpleTicketTest {
  // Testmethoden
}
```

- Klasse im gleichen Paket
- **import** macht **public** Klassen und Methoden den Testframeworks JUnit sichtbar
  - org.junit ist ein Paket, das zu JUnit gehört
  - org.junit.Test und org.junit.Assert sind die Klassen Test und Assert in diesem Paket
  - Erklärung von **static** folgt später
- Konvention für den Namen der Testklasse: `<IhreKlasse>Test`

Testen von Methoden: isUsable()

Spezifikation isUsable()

Fahrschein ist verwendbar, wenn er noch nicht abgestempelt worden ist.

Testen von Methoden: isUsable()

Spezifikation isUsable()

Fahrschein ist verwendbar, wenn er noch nicht abgestempelt worden ist.

Synthetischer Test

```
@Test
public void testIsUsable() {
  SimpleTicket st = new SimpleTicket(1, 1);
  assertTrue("Ticket should be usable", st.isUsable());
}
```

Testen von Methoden: isUsable()

Spezifikation isUsable()

Fahrschein ist verwendbar, wenn er noch nicht abgestempelt worden ist.

Synthetischer Test

```
@Test
public void testIsUsable() {
  SimpleTicket st = new SimpleTicket(1, 1);
  assertTrue("Ticket should be usable", st.isUsable());
}
```

- @Test: Markierung (Annotation) für eine Testmethode
- SimpleTicket st: Deklaration einer lokalen Variable mit Typ
- **new** SimpleTicket (1, 1): Erzeuge ein neues Objekt der Klasse **SimpleTicket** und rufe den Konstruktor mit den Parameter (1, 1) auf
- st.isUsable(): Aufruf der Method auf dem Objekt st
- assertTrue(message, condition): Prüft, ob condition erfüllt ist. Sonst Exception.

Vermeiden von sinnlosen Objekten

Erinnerung (Felder und Konstruktor von SimpleTicket)

```
// Preisstufe 1, 2, 3
private int level;
// Kind = 0, Erwachsener = 1
private int category;
// ...
public SimpleTicket(int level, int category) {
  this.level = level;
  this.category = category;
}
```

Vermeiden von sinnlosen Objekten

Erinnerung (Felder und Konstruktor von SimpleTicket)

```
// Preisstufe 1, 2, 3
private int level;
// Kind = 0, Erwachsener = 1
private int category;
// ...
public SimpleTicket(int level, int category) {
  this.level = level;
  this.category = category;
}
```

Sinnvolle und sinnlose Objekte

- Keine Restriktion beim Aufruf des Konstruktors
- Sinnvoll: **new** SimpleTicket(1, 1); **new** SimpleTicket(0, 3)
- Sinnlos: **new** SimpleTicket(1, 0); **new** SimpleTicket(−1, 42); **new** SimpleTicket(−999, −999)

Vermeiden der Konstruktion von sinnlosen Objekten

Zwei Strategien

1. Fehlermeldung beim Erzeugen (Factory Method)
2. Einschränkung des Wertebereichs

Strategie: Absicherung durch Factory Method

- Eine Java Methode kann einen Fehlerzustand durch eine *Exception* melden.
- Eine Exception wird durch die **throw** Anweisung ausgelöst und kann durch die **catch** Anweisung abgefangen werden.
- Die **throw** Anweisung nimmt ein Exception-Objekt als Parameter
- Konvention: vermeide Exceptions in Konstruktoren
- Lösung: verwende eine *factory method*

Strategie: Absicherung durch Factory Method

Implementierung Teil 1

```
private SimpleTicket(int level, int category) {
  this.level = level;
  this.category = category;
}
// next up: constructor method create()
```

- Sichtbarkeit des Konstruktors wird **private**.
- Anstelle von **new** SimpleTicket(l, c) verwende Aufrufe der Konstruktormethode create (nächste Folie):

  SimpleTicket.create (l, c)
- Die Konstruktormethode hat die gleichen Parameter wie der Konstruktor, aber testet sie bevor das Objekt erzeugt wird.

Strategie: Absicherung durch Factory Method

Implementierung Teil 2

```
/**
 * Safe constructor method for simple tickets.
 * @param level must be 1,2,3
 * @param category must be 0,1
 * @return a new legal SimpleTicket
 */
public static SimpleTicket create(int level, int category) {
  if (level < 1 || level > 3 || category < 0 || category > 1 ) {
    throw new IllegalArgumentException("Illegal level or category");
  }
  return new SimpleTicket(level, category);
}
```

- Eine **static** Methode hängt nicht an einem Objekt, sondern kann direkt über die Klasse als SimpleTicket.create() aufgerufen werden.
- IllegalArgumentException ist ein vordefinierter Konstruktor für ein Exception Objekt. Das String-Argument ist Teil der Fehlermeldung.