Lernziele
- Klassen, Klasseneigenschaften, Vererbung
- überschreiben von Methoden, Kaskadierung von Methoden mittels `super`
- Zugriffsschutz, Zugriffsmethoden und Initialisierung mit Konstruktoren
- Garbage Collection und finalize()
- Verhalten von Objekten
- Erstellung eines Programmgerüsts für weitere Übungen

Aufgabe 1) Eine einfache Klasse

Erstellen Sie eine neue namens `Person.java` Datei und erstellen Sie folgende Klasse

```
public class Person {
}
```

Obwohl diese Klasse augenscheinlich leer ist, hat sie schon eine Menge von Eigenschaften. Dies liegt daran, daß Sie stillschweigend von der Klasse `Object` erbt, die an der Spitze bzw. Wurzel der Klassenhierarchie in Java steht. Etwas ausführlicher hätte man obigen Code auch folgendermaßen schreiben können:

```
public class Person extends Object {
}
```

Wenn Sie die Klasse `Object` in der Online-Dokumentation betrachten, finden Sie eine Vielzahl von Methoden, unter anderem die Instanzmethode `toString()` die einen String zurückliefert, der zur textuellen Beschreibung des jeweiligen Objekts dient. Dies ist deswegen interessant, weil genau diese Methode von der `println`-Variante für Objekte verwendet wird. Mit anderen Worten, `println(Object)` funktioniert für alle Objekte, weil jede Klasse die Methode `toString()` von Objekt erbt.

Um dies zu überprüfen erstellen wir als nächstes eine weitere Datei namens `PersonTest.java` mit folgender Testklasse:

```
public class PersonTest {
  public static void main(String[] args) {
    Person p = new Person(); // Person Objekt anlegen

    // Person Objekt ausgeben
    System.out.println(p);
    // Beschreibung des Person Objekts ausgeben
    System.out.println(p.toString());
  }
}
```

Bei der Ausführung dieser Testklasse wird zweimal die gleiche Zeichenkette ausgegeben. Diese besteht aus dem Namen der Klasse und einer Hexadezimalzahl, welche die Adresse des Objekt im Hauptspeicher angibt. Daraus folgt, daß sich println(Object) der Methode `toString()` bedient.

Dies können wir einfach überprüfen, indem wir die Klasse Person um eine Instanzvariable ergänzen und anschließend das Verhalten von `toString()` anpassen, indem wir eine eigene Implementierung dieser Methode angeben. Dadurch wird die von `Object` geerbte Methode **überschrieben** bzw. **overridden** (vgl. Kap. 10.5).

```
public class Person {
  public String name; // Instanzvariable

  public String toString() {
    return "Person, name=" + name;
  }
}
```

Nun wird der Name der Person mit ausgegeben, vorausgesetzt er wurde initialisiert.

```
public class PersonTest {
  public static void main(String[] args) {
    Person p = new Person(); // Person Objekt anlegen

    p.name = "Mustermann"; // Name der Person initialisieren
    // Person Objekt ausgeben
    System.out.println(p);
    // Beschreibung des Person Objekts ausgeben
    System.out.println(p.toString());
  }
}
```

Die Methode `toString()` ist eine sogenannte Hook-Methode. Hook Methoden sind teil eines vordefinierten Ablaufs (hier: Ausgabe einer Beschreibung des Objekts). Durch überschreiben der jeweiligen Hook-Methode kann man sich in diesen Ablauf "einklinken" (daher der Name), um ihn für eigene Klassen anzupassen.

Achtung: Die Anpassung des Ausgabeformats über toString setzt den Zugriff auf die Klassendefinition voraus, ist also nur für eigene Klassen möglich. Um die Ausgabe von Objekten vorhandener (fremder) Klassen, z.B. Arrays, zu beeinflussen, muss eine eigene Hilfsmethode, z.B. arrayToString() geschrieben werden.

Zur einfacheren Fehlersuche (Debugging) sollte fast jede Klasse eine Implementierung von `toString()` besitzen, die eine aussagekräftige Beschreibung liefert. Diese zu implementieren sollte einer der ersten Schritte bei der Klassendefinition sein.

Aufgabe 2) Vererbung

Als nächstes definieren wir die Klasse `Student`. Diese soll von der Klasse `Person` erben. Erstellen Sie dafür folgenden Code in einer neuen Datei namens `Student.java`

```
public class Student extends Person {
```

```
  public long matrikelnr; // Instanzvariable
}
```

Ein Student soll also außer einem (geerbten) Namen auch eine Matrikelnummer besitzen. Zum Test passen Sie bitte die Testklasse wie folgt an, indem Sie dort ein neues Objekt der Klasse Student anlegen und dieses am Bildschirm ausgeben.

```
public class PersonTest {
  public static void main(String[] args) {
    Person p = new Person(); // Person Objekt anlegen
    Student s = new Student(); // Student Objekt anlegen

    p.name = "Mustermann";
    s.name = "Musterfrau";
    s.matrikelnr = 651276;

    // Person Objekt ausgeben
    System.out.println(p);
    // Student Objekt ausgeben
    System.out.println(s);
  }
}
```

Beim Ablauf dieses Programms stellen wir fest, daß auch der `Student` am Bildschirm ausgegeben werden kann, da er die `toString()` Implementierung der Klasse `Person` erbt. Allerdings gibt er sich als Person aus und zeigt nur seinen Namen an. Dies könnten wir durch Einfügen folgender Methode in die Klasse `Student` beheben:

```
  public String toString() {
    return "Student, name=" + name + ", matrikelnr=" + matrikelnr;
  }
```

Dies hätte jedoch zwei gravierende Nachteile:
- Ein Teil der Arbeit ist singgemäß schon bei Person vorhanden. Es wäre schöner, wenn wir diesen Teil wiederverwenden könnten.
- Bei der nächsten abgeleiteten Klasse (die von Student erbt) stimmt wieder der Klassenname in der Ausgabe nicht.

Um den Teil der Beschreibung aus `Person` wiederzuverwenden kann die dortige Implementierung von `toString()` über einen qualifizierten Zugriff aufgerufen und um den neuen Teil ergänzt werden. Dies geschieht mit Hilfe des Schlüsselworts `super` ähnlich wie bei verdeckten Variablen mit Hilfe des Schlüsselworts `this`.

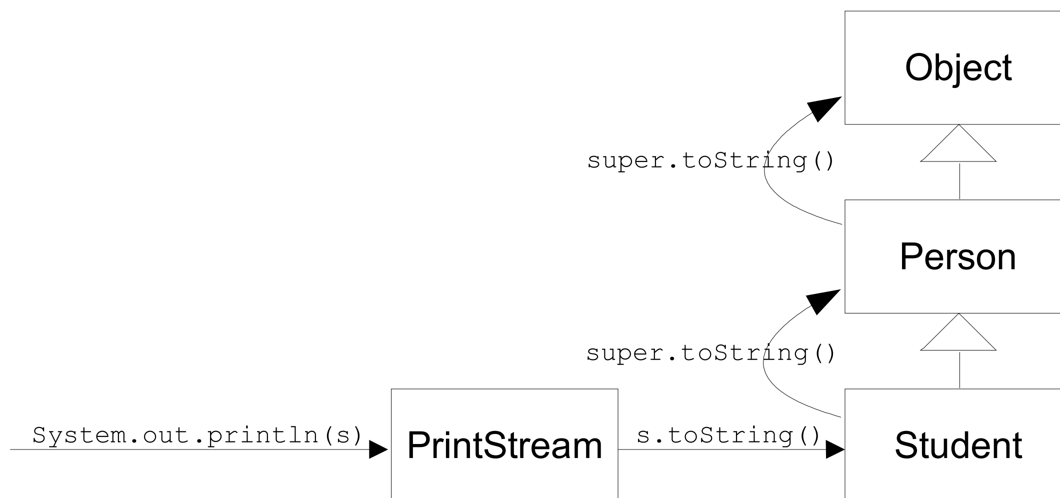
```
  // in der Klasse Student
  public String toString() {
    return super.toString() + ", matrikelnr=" + matrikelnr;
  }
```

Während die Bedeutung von this lautet "*dieses Objekt*" bedeutet `super` "*in der direkten*

Vaterklasse dieses Objekts". Der Aufruf von `super.toString()` liefert die Beschreibung einer `Person` zurück. Diese wird dann um die Matrikelnummer ergänzt und zurückgegeben. Allerdings steht am Anfang der Beschreibung nun immer noch `Person`, obwohl es sich um ein Objekt der Klasse `Student` handelt (Testlauf!). Um dies zu umgehen, kann man nun in der toString() Methode der Klasse Person ebenfalls den Aufruf der geerbten Methode einfügen.

```
// in der Klasse Person
public String toString() {
  return super.toString() + ", name=" + name;
}
```

Damit ergibt sich beim Aufruf von `System.out.println(s)` folgender Ablauf



Die verschiedenen Implementierungen von `toString()` sind also über die gegenseitigen Aufrufe **kaskadiert**. Dabei stellen die unteren Versionen Spezialisierungen im besten Sinne des Vererbungskonzepts dar: Die prinzipielle Funktion der Methode `toString()` "*liefere eine Beschreibung des Objekts mit dem Klassennamen und den (wichtigsten) Attributen*" bleibt auf jeder Stufe erhalten. Mit anderen Worten: obwohl es verschiedene Implementierungen von `toString()` gibt steht der Name konzeptionell immer für dasselbe. Insofern stiftet die Vielfalt gleichnamiger Methoden hier keine Verwirrung sondern trägt zur Vereinfachung bei: gleiche Dinge sind mit dem gleichen Namen belegt.

Will man die Anzeige der Speicheradresse unterdrücken und nur den Klassennamen anzeigen kann man dazu eine weitere Basisfunktion verwenden: Mit Hilfe der Methode

`getClass()` bei `Object` ist es möglich, auf einen Repräsentanten der Klasse jedes Objektes zuzugreifen. Dieser ist seinerseits eine Instanz der Klasse `Class` (nicht verwirren lassen!). Wie man in der Online-Hilfe nachprüfen kann, verfügt dieser Repräsentant über eine Methode `getName()`, die den Namen der Klasse zurückliefert. Somit liefert `obj.getClass().getName()` den Namen der Klasse des Objekts `obj` (`obj.getClass()` liefert den Repräsentanten `rep` und `rep.getName()` den gewünschten Namen).

Aufgabe 3) Initialisierung

Nachdem es überaus lästig ist, ein Objekt in mehreren Schritten erzeugen und anschließend initialisieren zu müssen, wollen wir als nächstes spezielle Initialisierungsfunktionen implementieren, die dies in einem Schritt erledigen. Diese nennt man Konstruktoren (vgl. Kap. 9.4.4). Das folgende Listing zeigt die Konstruktoren für die Klassen `Person` und `Student`.

```
public class Person {
  public String name; // Instanzvariable

  // Konstruktor mit Initialisierung
  public Person(String name) {
    this.name = name;
  }

  public String toString() {
    return this.getClass().getName() + ", name=" + name;
  }
}

public class Student extends Person {
  public long matrikelnr; // Instanzvariable

  // Konstruktor mit Initialisierung
  public Student(String name, long matrikelnr) {
    this.name = name;
    this.matrikelnr = matrikelnr;
  }

  public String toString() {
    return super.toString() + ", matrikelnr=" + matrikelnr;
  }
}
```

Ein Konstruktor hat ausnahmsweise keinen Rückgabewert und heißt immer genauso wie die zugehörige Klasse. Bei näherer Betrachtung fällt auf, daß die erste Zeile in beiden Konstruktoren identisch ist. Anstelle Sie zu wiederholen kann wie oben ein Aufruf der geerbten Methode erfolgen (dies in diesem Fall sogar zwingend, da mit der Definition des Konstruktors in der Klasse Person der Default-Konstruktor entfällt. Dieser wird vom Konstruktor der Subklasse implizit aufgerufen, wenn kein expliziter Aufruf vorhanden ist):

```
  // in der Klasse Student
  public Student(String name, long matrikelnr) {
    super(name);
    this.matrikelnr = matrikelnr;
  }
```

Da hier keine Methode mit einem festen Namen aufgerufen wird gilt hierbei eine spezielle Syntax. Anstelle von super.super() (wäre schlicht doof) oder super.Person() (würde die Bedeutung "*direkte Superklasse*" von super durch den direkten Klassenbezug zerstören) schreibt man einfach nur super(). Der Aufruf des Konstruktors erfolgt direkt beim Erzeugen eines Objekts:

```
public class PersonTest {
  public static void main(String[] args) {
    // Person Objekt anlegen und initialisieren
    Person p = new Person("Mustermann");
    // Student Objekt anlegen und initialisieren
    Student s = new Student("Musterfrau", 651276);

    // Person Objekt ausgeben
    System.out.println(p);
    // Student Objekt ausgeben
    System.out.println(s);
  }
}
```

Versucht man anschließend, ein Objekt ohne initialisierung anzulegen stellt man fest, daß dies nicht mehr geht. Dies liegt daran, daß es keine Konstruktoren ohne Parameter gibt. Vorher ging es, weil der Compiler automatisch einen Konstruktor ohne Parameter zur Verfügung stellt wenn gar kein Konstruktor explizit definiert ist (und nur dann). Um also eine Objekterzeugung ohne Initialisierung wieder zuzulassen muß explizit ein Konstruktor ohne Initialisierung definiert werden. Für die Klasse Student ist dies jedoch gar nicht wünschenswert, da ein Student stets eine Matrikelnummer haben soll. Hier der Konstruktor ohne Initialisierung für die Klasse Person:

```
  // Konstruktor ohne Initialisierung
  public Person() {
    this( "unbekannt"); // Aufruf des vorhandenen Konstruktors!!!
  }
```

Damit gibt es nun zwei Konstruktoren für die Klasse Person. Der Compiler kann diese anhand der unterschiedlichen Parameterlisten unterscheiden und je nach Aufruf die richtige auswählen. Diesen Fall, mehrere Methoden gleichen Namens mit unterschiedlichen Parameterlisten, hatten wir schoneinmal gesehen: bei der Methode println(). Man spricht hier von **überladen** bzw. **overloading** (vgl. Kap. 8.6.8). Wie beim überschreiben von Methoden gilt dabei, daß sich alle Varianten konzeptionell gleich verhalten bzw. den gleichen Effekt haben sollen - sie erreichen ihn nur auf unterschiedliche Art und Weise. Bei der methode `println()` lautet dieser Zweck "*Ausgabe eines Werts*", bei den Konstruktoren "*Initialisierung eines Objekts*".
Mit Hilfe von Konstruktoren ist es also möglich zu kontrollieren, ob und wie Objekte

erzeugt und initialisiert werden können bzw. müssen. Man kann die Erzeugung von Objekten auch für die Allgemeinheit verbieten, indem man einen anderen Zugriffsmodifizierer als `public` verwendet (vgl. Kap. 9.5.2).

Aufgabe 4) Zugriffsmethoden

Nachdem die Möglichkeit zur Initialisierung durch die Konstruktoren sichergestellt ist, kann man nun den willkürlichen Zugriff auf die Instanzvariablen verbieten um wahllose Änderungen zu verhindern und dem Prinzip des Information Hiding bzw. der Kapselung Rechnung zu tragen. Um trotzdem lesenden bzw. kontrollierten Schreibzugriff auf ausgewählte Instanzvariablen zu erlauben, kann man einfache Zugriffsmethoden (auch Accessors bzw Getters and Setters) implementieren.

```
public class Person {
  private String name; // Instanzvariable

  // Konstruktor mit Initialisierung
  public Person(String name) {
    this.name = name;
  }

  // Konstruktor ohne Initialisierung
  public Person() {
    this.name = "unbekannt";
  }

  public String toString() {
    return this.getClass().getName() + ", name=" + name;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;     // vorher ggf. pruefen ob name gueltig
  }
}

public class Student extends Person {
  private long matrikelnr; // Instanzvariable

  // Konstruktor mit Initialisierung
  public Student(String name, long matrikelnr) {
    super(name)
    this.matrikelnr = matrikelnr;
  }
```

```
  public String toString() {
    return super.toString() + ", matrikelnr=" + matrikelnr;
  }

  public long getMatrikelnr() {
    return matrikelnr;
  }
}
```

Wie man sieht, sind sowohl die get-Methoden als auch die set-Methoden sehr einfach. Man beachte, daß die get-Methode nur einen Rückgabewert (aber keine Argumente) und die set-Methode nur ein Argument (aber keinen Rückgabewert) besitzt, und diese jeweils vom Typ der betroffenen Instanzvariable sind. Für die matrikelnr wird sinnvollerweise nur eine get-Methode bereitgestellt, da sie nach der Initialisierung nicht mehr geändert werden soll.

Aufgabe 5) Klassenvariablen

Als nächstes sollen in die beiden Klassen Person und Student Zähler eingeführt werden, um die Anzahl der erzeugten Objekte zu verfolgen. Da nur ein Zähler für alle Objekte notwendig ist, wird dieser als Klassenvariable realisiert und direkt mit 0 initialisiert. Um diesen Zähler jedesmal zu erhöhen, wenn ein Objekt erzeugt wird, werden die Konstruktoren entsprechend ergänzt (bei `Person` beide!).

```
public class Person {
  public static int personen = 0;
  private String name; // Instanzvariable

  // Konstruktor mit Initialisierung
  public Person(String name) {
    this.name = name;
    personen++;
  }

  // Konstruktor ohne Initialisierung
  public Person() {
    this.name = "unbekannt";
    personen++;
  }

  // ...Rest der Klasse
}

public class Student extends Person {
  public static int studenten = 0;
  private long matrikelnr; // Instanzvariable
```

```
  // Konstruktor mit Initialisierung
  public Student(String name, long matrikelnr) {
    super(name)
    this.matrikelnr = matrikelnr;
    studenten++;
  }

  // ...Rest der Klasse
}
```

Die Wirkungsweise dieser Änderungen kann wie folgt demonstriert werden:

```
public class PersonTest {
  public static void main(String[] args) {
    // Person Objekte anlegen und initialisieren
    Person p1 = new Person("Max");
    Person p2 = new Person("Hans");
    Person p3 = new Person("Klaus");
    // Student Objekte anlegen und initialisieren
    Student s1 = new Student("Peter", 651276);
    Student s2 = new Student("Uschi", 651232);
    Student s3 = new Student("Sarah", 653445);

    System.out.println(Person.personen + " Personen");
    System.out.println(Student.studenten + " Studenten");
  }
}
```

Man beachte, daß jeder `Student` auch eine `Person` ist und durch den Aufruf von `super(...)` im Konstruktor von `Student` auch der Zähler für Personen erhöht wird.

Aufgabe 6) Garbage Collection und finalize()

Objekte werden mit Hilfe des new-Operators erzeugt und auf dem Heap gespeichert. Doch wie stellt man sicher, daß der Heap nicht irgendwann überläuft wenn man während des Programmablaufs viele Objekte erzeugt und wieder verwirft? In vielen Programmiersprachen gibt es zu diesem Zweck ein Pendant zum new-Operator um Objekte wieder freizugegeben. Dies setzt jedoch eine sorgfältige Buchführung durch den Programmierer voraus, damit kein Objekt verloren geht, ohne vorher freigegeben zu werden. Selbst bei größter Sorgfalt gehen dabei immer wieder Objekte verloren. Dies führt zu sog. Speicherlecks die selbst bei geringer Größe der verschlampten Objekte zu Problemen in lange laufenden Softwaresystemen, z.B. Steuerungen von Anlagen, zur Folge haben können. In Java gibt es zur Vermeidung derartiger Probleme einen "Müllabfuhr"-Mechanismus bzw. Garbage Collection (vgl. Kap. 9.6)

Ein Objekt geht verloren, wenn der letzte Verweis auf dieses Objekt gelöscht wird. Dies kann durch Löschung der entsprechenden Referenzvariable geschehen (etwa beim verlassen einer Methode wenn es sich um eine lokale Variable handelt), oder wenn diese

mit einem neuen Wert überschrieben wird. Die Garbage Collection, die automatisch in regelmäßigen Abständen angestoßen wird, findet alle diese Objekte und sorgt für deren fachgerechte Entsorgung. Um dem Programmierer die Chance zu geben, im Rahmen dieser Entsorgung noch eigene Aufräumarbeiten zu erledigen, wird unmittelbar vorher die Methode `finalize()` aufgerufen. Dies ist eine weitere Hook-Methode die bereits bei Objekt definiert ist (vgl. Online Hilfe).

Normalerweise merkt man nichts von der Garbage Collection, da diese im Hintergrund läuft. Man muß sie auch nicht explizit anstoßen. Um jedoch vor zeitkritischen Programmteilen (z.B. Steuerungsabläufen) eine Unterbrechung durch die Garbage Collection zu vermeiden kann es Sinn machen, vor diesen Abschnitten explizit die "Müllabfuhr zu bestellen" (dasselbe gilt auch für Demos im Rahmen von Übungen ;-). Dies ist mit Hilfe der Methode `gc()` in der Klasse System möglich (vgl. Online-Hilfe). Um die Funktionsweise der GarbageCollection zu zeigen, fügen ergänzen wir unsere Testklasse wie folgt:

```
public class PersonTest {
  public static void main(String[] args) {
    // Person Objekt anlegen und initialisieren
    Person p = new Person("Mustermann");
    // Student Objekt anlegen und initialisieren
    Student s = new Student("Musterfrau", 651276);

    System.out.println(Person.personen + " Personen");
    System.out.println(Student.studenten + " Studenten");

    s = null;     // Student wegwerfen
    System.gc(); // Garbage Collection bestellen

    System.out.println(Person.personen + " Personen");
    System.out.println(Student.studenten + " Studenten");
  }
}
```

Außerdem fügen wir folgende Methoden in die Klassen `Person` und `Student` ein.

```
  // Klasse Person
  public void finalize() {
    System.out.println("Aaargh!");
    personen--;
  }

  // Klasse Student
  public void finalize() {
    System.out.print("Ooouch! ");
    studenten--;
    personen--;
  }
```

Man beachte, daß die Methode weder Argumente noch Rückgabewert besitzt. Beim

Ablauf des fertigen Programms stellt man fest, daß die Garbage Collection wie erwartet das Objekt `s` entsorgt und dabei die Methode `finalize()` aufruft. Hinterher sind sowohl die beiden Klassenvariablen `Person.personen` und `Student.studenten` jeweils um eins kleiner.

Aufgabe 7) Verhalten von Objekten

Teilaufgabe a) Personen können laufen

Implementieren Sie in der Klasse Person eine parameterlose Methode namens walk. Diese soll den Text "tap, tap" auf dem Bildschirm ausgeben. Testen Sie diese Methode zunächst, indem Sie ein Objekt der Klasse Person erzeugen und die Methode aufrufen. Erzeugen Sie anschließend ein Objekt der Klasse Student und rufen Sie ebenfalls die Methode auf. Was schließen Sie aus dem Ergebnis?

Teilaufgabe b) Studenten laufen anders

Überschreiben Sie nun die Methode walk bei der Klasse Student. Dort soll der Text "schlurf, schlurf" ausgegeben werden. Testen Sie die neue Methode wieder durch Aufruf in beiden Klassen.

Teilaufgabe c) Jeden Tag wird einer geboren

Implementieren Sie nun eine neue Klasse Idiot die von Person erbt. Bei einem Idiot soll die Methode walk den Text "hüpf, tap, tap, hüpf" ausgeben. Verwenden Sie bei der Implementierung der Methode die vorhandene Funktionalität in der Klasse Person mit einem super-Aufruf (Dazu ist evtl. eine Zerlegung der vorhandenen Methoden nötig).

Aufgabe 8) Zusatzaufgaben

Teilaufgabe a) Mars und Venus

Ergänzen Sie die Klasse Person um eine Instanzvariable sex (Geschlecht). Diese soll als Ganzzahl implementiert sein. Implementieren Sie zur besseren Lesbarkeit des Programms zwei öffentliche Konstanten auf Klassenebene für die Werte dieser Variable

Bsp.

```
public static final int MALE = 1;
```

Ergänzen Sie die Konstruktoren und/oder Zugriffsmethoden für dieses Attribut

Teilaufgabe b) Lachen ist gesund

Implementieren Sie nun eine Methode laugh um den Personen das Lachen beizubringen. Bei Männern soll "hahaha", bei Frauen "hihihi" ausgegeben werden.

Teilaufgabe c) Voll der Wahnsinn

Überschreiben Sie die Methode laugh beim Idioten und geben Sie unabhängig vom

Geschlecht "hihihahahohoho" aus.

Teilaufgabe d) Es weihnachtet sehr

Implementieren Sie eine Klasse Weihnachtsmann. Von dieser Klasse soll es nur eine einzige Instanz geben (können). Wenn der Weihnachtsmann läuft, ertönt ein leises "klingel, klongel" und wenn er lacht klingt das so: "Ho, Ho, Ho".

Teilaufgabe e) Weitere Charaktereigenschaften

Denken Sie sich weitere Eigenschaften und Verhaltensmuster für Personen, Studenten usw. aus und implementieren Sie diese.