Aufgabe 1

a) Methoden sollten als Klassenmethoden (static) definiert werden, wenn sie keine Instanzvariablen verändert oder benutzt. Insbesondere, wenn sie nur auf ihren Parametern agieren, wenn die Klasse keine Instanzvariablen hat oder wenn sie nur auf Klassenvariablen agiert.

b) Lokale Variablen sind automatisch privat, ihr Skope ist immer nur die Methode bzw. der Block in dem sie deklariert werden. Es macht Sinn Klassenmethoden als privat zu deklarieren, wenn sie nur als Hilfsmethoden für andere Methoden der Klasse agieren und wenn sie private Variablen exposen könnten.

c) Wenn in einer Klassendefinition kein Konstruktor existiert, wird automatisch ein leerer Konstruktor aufgerufen, welcher die Instanzvariablen mit den Defaultwerten bzw. ohne Werte initialisiert.
Konstruktoren können nur explizit vererbet werden, indem `super` im Konstruktor der Unterklasse aufgerufen wird.
Konstruktoren können privat sein, dies macht Sinn für Klassen, welche nur Klassenmethoden und oder Konstanten sammeln und, wenn meherer Konstruktoren oder Factory Methoden verwendet werden.

d) Die Main Methode muss eine Klassenmethode sein, da sie direkt in der JVM aufgerufen wird, ohne das zuvor ein Objekt erzeugt werden musste. Schließlich können nur in der Main Methode Objekte erzeugt werden, also nicht bevor sie aufgerufen wurde. Wenn die Main Methode nicht statisch ist, gibt das Programm einen Fehler aus.

e) Als `protected` deklariert Instanzvariablen sind sichtbar in der Klasse, in Unterklassen und im Package. Konstruktoren können als `protected` deklariert werden, wenn Objekte des entsprechenden Typs nicht außerhalb des Packages erzeugt werden sollen.

f) Abstrakte Klassen sind Klassen, von denen keine Objekte erstellt werden können. Sie dienen dazu Methoden und Variablen in einer Klasse zusammen zu fassen, welche entweder nur statisch sind oder von Unterklassen geerbt werden sollen. Klassenmethoden und Klassenvariablen von abstrakten Klassen können aufgerufen werden als `Class.var` bzw. `Class.CONST` bzw. `Class.method(pars)`.

g) Objekte abstrakter Klassen können zwar nicht erzeugt werden, es macht aber trotzdem unter Umständen Sinn einen Konstruktor in ihnen zu definieren, wenn dieser von den Unterklassen geerbt werden soll.

h) Schnittstellen definieren Variablen und Methoden, die von bestimmten Klassen implementiert werden sollen. Damit bieten sie eine Möglichkeit gewisse Funktionalitäten von Klassen zusammen zu fassen und sicherzustellen, dass diese für alle Klassen, welche das Interface implementieren aufrufbar sind.

i) Es ist keine gute Idee in einer Elternklasse Instanzvariablen vom Typ von Kinderklassen zu definieren, da dies zu einer Endlosschleife beim Erzeugen von Objekten der Eletern-

klasse führt, denn der Konstruktor ruft den Konstruktor der Kindklasse auf, welche wiederum den Konstruktor der Elternklasse aufruft und so weiter.

j) Methodenüberladung bedeutet, dass eine Methode mehrmals mit demselben Namen und verschiedenen Argumenten definiert wird. Beim aufrufen wird dann automatisch die Implementierung mit den passenden Argumenten aufgerufen. Methodenüberschreibung bedeutet, dass eine Methode der Elternklasse in der Kindklasse neu definiert wird. In diesem Fall kann auf Objekten der Kindklasse nur noch die überschriebene Methode aufgerufen werden.

k) Variablen können in Unterklassen nicht überschrieben werden, nur versteckt. Das heißt, wenn die Variable neu in der Unterklasse definiert wird, so gilt sie zwar in den in der Unterklasse neu definierten Methoden, nicht aber in den Methoden, welche von der Oberklasse geerbt wurden.

l) Da in Java primitive Datentypen keine Objekte sind, manche Strukturen wie zum Beispiel Arrays und Enumeratoren aber nur mit Objekten gebaut werden können, gibt es Wrapper Klassen für die primitiven Datentypen. Die Operationen für primitive Datentypen funktionieren auch noch für Objekte der Wrapper Klasse. Dabei werden die Wrapper zunächst ausgepackt, dann die Operation ausgeführt und dann das Resultat wieder in Wrapper verpackt. Das Auspacken funktioniert allerdings nur bei Objekten, die mit automatischen Einpacken gebildet wurden und nur innerhalb des Wertbereichs -128 bis 127. Dies kann zu großen Verwirrungen führen.

m) Generische Klassen sind Klassen, welche es ermöglichen eine ganze Familie von Klassen auf einmal zu definieren. Sie haben ein Objekttyp als Argument, welcher in der Klasse verwendet wird und können dann mit verschiedene Objekttypen initialisiert werden.

n) Vererbungspolymorphie bedeutet, dass Objekte von einer Klasse auch immer vom Typ aller Oberklassen und Schnittsetellen dieser Klasse sind.

o) Exceptions (Ausnahmefehler) sind unerwartete Fälle, auf welche das Programm reagieren muss. In Java sind Exceptions als Objekte implementiert, welche vom Exception Objekt erben. Methoden können Exception Objekte werfen. Wenn eine Methode Exceptions wirft, so müssen diese aufgefangen werden. Runtime-Exceptions sind Exceptions, welche von der `RuntimeException` Klasse erben. Sie sind insofern besonders, als sie nicht aufgefangen werden müssen.

Aufgabe 2

```
abstract class Vehicle{
  // ATTRIBUTES
  String vehicleClass; // as specified in driver license
  String brand = "no name";
  int tankCapacity; // in l
  int tankFilling; // in l
  int consumption; // in km/l

  // CONSTRUCTOR
  public Vehicle(String vehicleClass, String brand, int tankCapacity, int consumption) {
    this.vehicleClass = vehicleClass;
    this.brand = brand;
    this.tankCapacity = tankCapacity;
    this.tankFilling = 0;
    this.consumption = consumption;
  }

  // METHODS
  public String toString(){
    String type = "";
    if( vehicleClass.equals("A") ){
      type = "Bike";
    } else if( vehicleClass.equals("B") ) {
      type = "Car";
    }
    return String.format("%s from %s\n  Tank: %s/%s\n", type, brand, tankFilling, tankCapacity);
  }
  abstract String sound();
}


public class Bike extends Vehicle{
  public Bike(String brand, int tankCapacity, int consumption){
    super("A", brand, tankCapacity, consumption);
  }
  public String sound(){
    return "Aaar Aaar";
  }
}


public class Car extends Vehicle{
  public Car(String brand, int tankCapacity, int consumption){
    super("B", brand, tankCapacity, consumption);
  }
  public String sound(){
    return "Wrrr Wrrr";
  }
}


import java.sql.Time;

public class Driver<V extends Vehicle>{
  // ATTRIBUTES
  V vehicle;
  int position = 0; // in km

  // CONSTRUCTOR
  public Driver(V vehicle){
    this.vehicle = vehicle;
  }

  // METHODS
  public Time drive(int km, int speed) throws NotEnoughFuelException {
    if( this.vehicle.consumption*this.vehicle.tankFilling >= km ){
      this.position += km;
      this.vehicle.tankFilling -= (int)((double)km/(double)this.vehicle.consumption);
```

```
      Time time = new Time( (long)((double)km/(double)speed*3600-3600)*1000 );
      return time;
    } else {
      int maximalDistance = this.vehicle.consumption*this.vehicle.tankFilling;
      throw new NotEnoughFuelException("Tried to drive further than possible.", maximalDistance);
    }
  }

  public void tank(int liter) throws NotEnoughCapacityException {
    if( this.vehicle.tankFilling + liter <= this.vehicle.tankCapacity ){
      this.vehicle.tankFilling += liter;
    } else {
      int capacityOverflow = this.vehicle.tankFilling + liter - this.vehicle.tankCapacity;
      throw new NotEnoughCapacityException("Tried to tank more than possible.", capacityOverflow);
    }
  }
  public int tankUp(){
    int tanking = this.vehicle.tankCapacity - this.vehicle.tankFilling;
    this.vehicle.tankFilling += tanking;
    return tanking;
  }

  public boolean isEmpty(){
    return this.vehicle.tankFilling == 0;
  }
  public boolean isFull(){
    return this.vehicle.tankFilling == this.vehicle.tankCapacity;
  }

  public String toString(){
    return String.format("| Driver at %s km on:\n  %s", position, vehicle);
  }
}


public class NotEnoughFuelException extends Exception {
  // ATTRIBUTES
  public int maximalDistance;

  // CONSTRUCTORS
  public NotEnoughFuelException(){
    super();
  }
  public NotEnoughFuelException(String reason){
    super(reason);
  }
  public NotEnoughFuelException(String reason, int maximalDistance){
    super(reason);
    this.maximalDistance = maximalDistance;
  }
}


public class NotEnoughCapacityException extends Exception {
  // ATTRIBUTES
  int capacityOverflow;

  // CONSTRUCTORS
  public NotEnoughCapacityException(){
    super();
  }
  public NotEnoughCapacityException(String reason){
    super(reason);
  }
  public NotEnoughCapacityException(String reason, int capacityOverflow){
    super(reason);
    this.capacityOverflow = capacityOverflow;
  }
}


public class TestVehicle{
  public static void drive(Driver driver, int km, int speed){
```

```
    try {
      System.out.println("Drove " + km + "km in: " + driver.drive(km, speed));
    }
    catch( NotEnoughFuelException e ) {
      if( e.maximalDistance > 0 ){
        System.out.println("Tried to drive " + (km-e.maximalDistance) + "km further than possible
                           with current tank filling. Drove as far as possible instead.");
        drive(driver, e.maximalDistance, speed);
      } else {
        System.out.println("Could not drive any further with current tank filling.");
      }
    }
  }
  public static void tank(Driver driver, int liter){
    try
    {
      driver.tank(liter);
    }
    catch( NotEnoughCapacityException e )
    {
      System.out.println("Tried to tank " + e.capacityOverflow + "l more than the tank could
                         hold. Tanked full instead");
      driver.tankUp();
    }
  }

  public static void main(String[] args) {
    // CREATE SOME OBJECTS
    Car audi = new Car("Audi", 70, 17);
    Car fiat = new Car("Fiat", 45, 35);
    Bike bike = new Bike("Yamaha", 30, 30);
    Driver audiDriver = new Driver<Car>(audi);
    Driver fiatDriver = new Driver<Car>(fiat);
    Driver biker = new Driver<Bike>(bike);
    System.out.println(audiDriver);
    System.out.println(fiatDriver);
    System.out.println(biker);

    // DRIVE AROUND
    audiDriver.tankUp();
    fiatDriver.tankUp();
    biker.tankUp();
    drive(audiDriver, 2000, 150);
    drive(fiatDriver, 2000, 100);
    drive(biker, 2000, 200);
    System.out.println();
    System.out.println(audiDriver);
    System.out.println(fiatDriver);
    System.out.println(biker);

    // System.out.println();
    drive(audiDriver, 10, 50);
    System.out.println("Empty tank? " + audiDriver.isEmpty());
    System.out.println(audiDriver);
    tank(audiDriver,100);
    System.out.println("Full tank? " + audiDriver.isFull());
    System.out.println(audiDriver);

    // MAKE SOUNDS
    System.out.println();
    System.out.println(biker.vehicle.sound());
    System.out.println(audiDriver.vehicle.sound());
    System.out.println(fiatDriver.vehicle.sound());
  }
}
```