```java
public class BMW extends Auto{
    private boolean hatXDrive;

    public BMW(int leistung, boolean hatXDrive) {
        super(leistung);
        this.hatXDrive = hatXDrive;
    }
}

```

```java
public abstract class Auto {
    private int leistung;

    public Auto(int leistung) {
        this.leistung = leistung;
    }

    public int getLeistung() {
        return leistung;
    }

    public String toString(){
        return "Auto mit " + this.leistung+ " PS";
    }
}

```

```java
public class Test {
    // Diese Methode verwendet keinen generischen Datentyp für die Übergabe des
    // Parameters. Da bei generischen Datentypen die Vererbung allerdings
    // irrelevant ist, kann diese Methode niemals aufgerufen werden. Das liegt
    // daran, dass es keine Werkstatt vom Typ "Auto" geben kann, da "Auto"
    // abstrakt ist. Es kann also nur "BMW" oder "Mercedes" sein. Hätte die
    // Methode "Werkstatt<BMW> werkstatt" als Parameter könnte die Methode
    // aufgerufen werden. Allerdings nicht mit einer Werkstatt vom Typ "Mercedes".
    // Um das zu lösen gibt es zwei Möglichkeiten:
    // 1. Methode mit generischem Datentyp
    // 2. Methode mit einer Wildcard
    public static boolean gleicheLeistung(Werkstatt<Auto> werkstatt) {
        return werkstatt.getHebeBuehne1().getLeistung() == werkstatt.getHebeBuehne2().
getLeistung();
    }

    // Auch Methoden können generische Datentypen verwenden. Diese Methode kann
    // sowohl eine Werkstatt von BMW, als auch eine von Mercedes verarbeiten.
    public static <T extends Auto> void printHebebuehne1(Werkstatt<T> werkstatt){
        System.out.println(werkstatt.getHebeBuehne1().toString());
    }

    // Alternativ kan auch mit einer Wildcard gearbeitet werden. Damit wird die
    // doppelte Prüfung des korrekten generischen Datentyps vermieden. Es wird
    // nur bei der Instanziierung der generischen Klasse geprüft.
    public static void printHebebuehne2(Werkstatt<?> werkstatt){
        System.out.println(werkstatt.getHebeBuehne2().toString());
    }

    public static void main(String[] args) {

        // Erstellen einer Werkstatt. Es wird angenommen, dass nur Autos vom
        // gleichen Typen in einer Werkstatt behandelt werden
        BMW a = new BMW(340, true);
        BMW b = new BMW(130, false);

        // In den </> Klammern wird der generischen Klasse der Datentyp übergeben
        Werkstatt<BMW> w = new Werkstatt<>(a,b);

        // Da nur Autos des gleichen Typs in eine Werkstatt dürfen, wäre folgender
        // Aufruf nicht möglich.
        //  Mercedes a = new Mercedes(44, false);
        //  BMW b = new BMW(22, true);
        //  Werkstatt<BMW> w = new Werkstatt<>(a, b);

        w.printWerkstatt();
        // Konsole:
        // Werkstatt
        // Hebebuehne 1: Auto mit 340 PS
        // Hebebuehne 2: Auto mit 130 PS

        printHebebuehne1(w); // Konsole: Auto mit 340 PS
        printHebebuehne2(w); // Konsole: Auto mit 130 PS
    }
}

```

```java
public class Mercedes extends Auto{
    public boolean hat4Matic;

    public Mercedes(int leistung, boolean hat4Matic) {
        super(leistung);
        this.hat4Matic = hat4Matic;
    }
}

```

```java
// Generische Klassen ermöglichen es allgemeine Klassendefinitionen zu erstellen.
// Dazu wird innerhalb der Klasse kein konkreter Datentyp verwendet, sondern ein
// Platzhalter. In diesem Fall ist der Platzhalter "T". Bei der Instanziierung
// wird der gewünschte Datentyp dann als Parameter in </>-Klammern übergeben. In
// diesem Fall soll eine Werkstatt als Klasse definiert werden. In eine Werkstatt
// dürfen nur Autos des gleichen Typs. Außerdem wird durch das "extends Auto"
// festgelegt, dass der übergebene Datentyp "Auto" oder eine Kindklasse davon sein
// muss. Da "Auto" abstract ist, muss es eine Kindklasse sein.
public class Werkstatt<T extends Auto> {

    private T hebeBuehne1, hebeBuehne2;

    public Werkstatt(T hebeBuehne1, T hebeBuehne2) {
        this.hebeBuehne1 = hebeBuehne1;
        this.hebeBuehne2 = hebeBuehne2;
    }

    public T getHebeBuehne1(){
        return this.hebeBuehne1;
    }

    public T getHebeBuehne2(){
        return this.hebeBuehne2;
    }

    public void printWerkstatt(){
        System.out.println("Werkstatt");
        System.out.println("Hebebuehne 1: " + this.hebeBuehne1);
        System.out.println("Hebebuehne 2: " + this.hebeBuehne2);
    }
}

```

Java Collection-Typen Übersicht

Typ	Sortiert	Duplikate erlaubt	Einfügereihenfolge	Synchronisiert
ArrayList	Nein	Ja	Ja	Nein
LinkedList	Nein	Ja	Ja	Nein
HashSet	Nein	Nein	Nein	Nein
LinkedHashSet	Nein	Nein	Ja	Nein
TreeSet	Ja	Nein	automatisch sortiert	Nein
HashMap	Nein	Schlüssel: Nein	Nein	Nein
LinkedHashMap	Nein	Schlüssel: Nein	Ja	Nein
TreeMap	Ja	Schlüssel: Nein	automatisch sortiert	Nein
Hashtable	Nein	Schlüssel: Nein	Nein	Ja (veraltet)

```java

// Auto implementiert das Interface Comparable<T>, um es in einem SortedSet zu verwenden
// Comparable<T> zwingt dann die Klasse dazu eine Methode zu definieren,
// die es ermöglicht zwei Objekte von dieser Klasse miteinander zu vergleichen
public class Auto implements Comparable<Auto>{
    private String marke;
    private int leistung;

    public Auto(String marke, int leistung) {
        this.marke = marke;
        this.leistung = leistung;
    }

    public int compareTo(Auto a){
        // Hier wird einfach auf die compare Methode von "Integer" zurückgegriffen.
        // Zurückgegeben wird 1, 0 oder -1
        return Integer.compare(this.leistung, a.leistung);

        // Alternativ kann auch nach mit einem String verglichen werden
        // return this.marke.compareTo(a.marke);
    }
}

```

```java
import java.util.*;

// Alternativ zu einem Array können mehrere Referenzobjekte in einer Collection
// gespeichert werden. Collections können dynamisch wachsen und schrumpfen und
// benötigen daher keine vordefinierte Größe.
// Die Größe kann sich während der Laufzeit verändern.
// Die Collection kann dabei entweder eine "List" oder ein "Set" sein.
// Beide Varianten haben eigene Eigenschaften.
// List: Geordnete Folge von Elementen
// Set: Sammlung von Objekten, in der jedes Objekt nur einmal vorkommen darf
// -> Mit einem SortedSet (Kindklasse zu Set) ist auch ein Set in einer
// geordneten Reihenfolge

class CollectionsDemo {
    public static void main(String[] args){
        // --- LISTS ---
        List<Integer> arrayList = new ArrayList<>();
        arrayList.add(30);
        arrayList.add(10);
        arrayList.add(20);
        Collections.sort(arrayList); // funktioniert jetzt
        System.out.println("ArrayList (sortiert): " + arrayList);

        // Abstammung von LinkedList: Collection -> List -> LinkedList
        // Die List fügt Elemente immer am Ende an.
        List<Integer> linkedList = new LinkedList<>();
        linkedList.add(5);
        linkedList.add(15);
        linkedList.add(10);
        Collections.sort(linkedList);
        System.out.println("LinkedList (sortiert): " + linkedList);

        // --- SETS ---
        Set<Integer> hashSet = new HashSet<>();
        hashSet.add(3);
        hashSet.add(1);
        hashSet.add(2);
        List<Integer> sortedHashSet = new ArrayList<>(hashSet);
        Collections.sort(sortedHashSet);
        System.out.println("HashSet (sortiert): " + sortedHashSet);

        Set<Integer> linkedHashSet = new LinkedHashSet<>();
        linkedHashSet.add(9);
        linkedHashSet.add(7);
        linkedHashSet.add(8);
        List<Integer> sortedLinkedHashSet = new ArrayList<>(linkedHashSet);
        Collections.sort(sortedLinkedHashSet);
        System.out.println("LinkedHashSet (sortiert): " + sortedLinkedHashSet);


        // Abstammung von TreeSet: Collection -> Set -> SortedSet -> TreeSet
        // Da TreeSet also ein SortedSet ist muss die Klasse, aus der ein TreeSet
        // erstellt werden soll, das Interface "Comparable<T>" implementieren
        // (siehe Auto)
        // Das TreeSet fügt Elemente automatisch in der richtigen sortierten
        // Reihenfolge hinzu.
        Set<Integer> treeSet = new TreeSet<>();
        treeSet.add(100);
        treeSet.add(50);
        treeSet.add(75);
        System.out.println("TreeSet (automatisch sortiert): " + treeSet);

        // --- MAPS ---
        Map<Integer, String> hashMap = new HashMap<>();
        hashMap.put(3, "Drei");
        hashMap.put(1, "Eins");
        hashMap.put(2, "Zwei");
```

```
68          System.out.println("HashMap (unsortiert): " + hashMap);
69
70          Map<Integer, String> linkedHashMap = new LinkedHashMap<>();
71          linkedHashMap.put(20, "Zwanzig");
72          linkedHashMap.put(10, "Zehn");
73          linkedHashMap.put(30, "Dreißig");
74          System.out.println("LinkedHashMap (Einfügereihenfolge): " + linkedHashMap);
75
76          Map<Integer, String> treeMap = new TreeMap<>();
77          treeMap.put(300, "CCC");
78          treeMap.put(100, "AAA");
79          treeMap.put(200, "BBB");
80          System.out.println("TreeMap (automatisch nach Schlüssel sortiert): " + treeMap);
81
82          Map<Integer, String> hashtable = new Hashtable<>();
83          hashtable.put(9, "Neun");
84          hashtable.put(7, "Sieben");
85          hashtable.put(8, "Acht");
86          System.out.println("Hashtable (unsortiert): " + hashtable);
87
88
89          // --- ITERATOR inkl. Custom-Class ---
90          // Die Klasse Auto hat das Interface Comparable<T> implementiert -> Siehe Auto
91          TreeSet<Auto> treeSet2 = new TreeSet<>();
92          treeSet2.add(new Auto("BMW", 340));
93          treeSet2.add(new Auto("Mercedes", 224));
94          treeSet2.add(new Auto("Volvo",355));
95
96          // Jede Collection enthält einen Iterator, um über die Collection zu iterieren
97          // und alle Elemente zu erreichen
98          Iterator<Auto> it = treeSet2.iterator();
99
100         // Jeder Iterator hat die Methoden "next", "hasNext" und "remove".
101         // Damit kann über jedes Element iteriert werden
102         // While-Schleife
103         while(it.hasNext()){
104             System.out.println(it.next());
105         }
106         // For-Schleife (ausführlich)
107         for(Iterator<Auto> i = treeSet2.iterator(); i.hasNext();){
108             System.out.println(i.next());
109         }
110         // For-Schleife (simpel/kurz)
111         for(Auto a : treeSet2){
112             System.out.println(a);
113         }
114     }
115 }
116
117
118
119 // Für jeden elementaren Datentyp gibt es eine entsprechende Wrapper-Klasse im Paket
120 // java.lang. Diese Klassen ermöglichen es, primitive Werte als Objekte zu behandeln.
121 // (Sie erben somit ebenfalls von "Object") Dies ist notwendig, wenn man primitive
122 // Werte in Kontexten verwenden möchte, die Objekte erfordern (z.B. in Collections
123 // wie ArrayList oder wenn man Methoden auf den Werten aufrufen möchte).
124 //
125 // Elementarer Datentyp     |   Wrapper-Klasse
126 //      boolean         |       Boolean
127 //      byte                |       Byte
128 //      short               |       Short
129 //      int                 |       Integer
130 //      long                |       Long
131 //      float               |       Float
132 //      double              |       Double
133 //      char                |       Character
134 //
```

```
135 // Boxing - Beispiel:
136 // public int getPatientNr(){
137 //         // Um an den elementaren Datentyp einer Wrapper-Klasse zu gelangen muss man
138 //         // diesen theoretisch "unboxen". Das passiert mittlerweile allerdings
139 //         // automatisch.
140 //         // return this.patientNr.intValue(); // Nicht notwendig
141 //         return this.patientNr; // Auto-Unboxing
142 //     }
143 //
144 // Unboxing - Beispiel:
145 // public void setPatientNr(){
146 //         // Um den elementaren Datentyp einer Wrapper-Klasse zu setzen, muss man diesen
147 //         // theoretisch "boxen". Das passiert mittlerweile allerdings automatisch.
148 //         // this.patientNr = Integer.valueOf(42); // Nicht notwendig
149 //         this.patientNr = 42; // Autoboxing
150 //     }
151
```

```java
public class Test {
    public static void main(String[] args) {
        // Der 'try'-Block umschließt Code, der potenziell
        // eine Ausnahme (Exception) auslösen kann. In
        // diesem Fall kann der Konstruktor der Klasse
        // "BankAccount" Exceptions werfen. Wenn eine Exception
        // einer Elternklasse in den Kindklassen nicht behandelt
        // wird, muss sie weiter geworfen werden, damit sie später
        // behandelt werden kann.
        try {
            // Der folgende Block wirft eine Exception vom Typ
            // "ExceedWithdrawl", da mehr abgehoben wird als
            // vorhanden ist.
            BankAccount ba = new BankAccount(20.0);
            ba.withdraw(50.0);

            // Die folgende Zeile wirft eine Exception vom Typ
            // "NoNegativeInitialBalance", da ein negativer
            // Wert dem Konstruktor übergeben wird
            BankAccount ba2 = new BankAccount(-20.0);
        }
        // Die Reihenfolge der 'catch'-Blöcke ist wichtig:
        // Spezifischere Ausnahmen sollten zuerst gefangen
        // werden, gefolgt von allgemeineren Ausnahmen.
        // In diesem Fall sind beide Ausnahmen spezifisch und
        // werden einzeln abgefangen. Alternativ könnte man auch
        // beide Exceptions allgemein abfangen, indem man mit der
        // Elternklasse "Exception" arbeitet.

        // Der erste 'catch'-Block fängt spezifisch die
        // 'NoNegativeInitialBalance' ab. Wenn eine
        // 'NoNegativeInitialBalance' im 'try'-Block geworfen
        // wird, wird der Code in diesem 'catch'-Block ausgeführt.
        catch (NoNegativeInitialBalance e) {
            System.out.println(e);
            // Konsole: NoNegativeInitialBalance:
            // Ein Konto muss zur Erstellung gedeckt sein.
        }
        catch (ExceedWithdrawl e) {
            System.out.println(e.getMessage());
            // Konsole: Fehler: Nicht genügend Guthaben vorhanden!
            // Versucht: 50.0, Verfügbar: 20.0
        }
        // Allgemeines Abfangen jedes Fehlers
        catch (Exception e) {
            e.printStackTrace();
        }

        // Nach dem 'try-catch'-Block wird die Programmausführung fortgesetzt,
        // unabhängig davon, ob eine Ausnahme aufgetreten ist oder
        // nicht (solange sie gefangen wurde). Wenn eine Ausnahme auftritt,
        // die keine "RuntimeException" ist, und gefangen wird, wird der Rest
        // des 'try'-Blocks übersprungen, aber der Code nach dem
        // 'catch'-Block wird ausgeführt.

    }
}

```

```java
public class BankAccount {
    private double balance;

    // NoNegativeInitialBalance ist eine Kindklasse von
    // Exception und NICHT von RuntimeException. Das bedeutet,
    // dass diese Ausnahme explizit in der Methodensignatur
    // deklariert werden muss und vom aufrufenden Code entweder
    // gefangen oder weiter geworfen werden muss.
    public BankAccount(double balance) throws NoNegativeInitialBalance {
        if(balance < 0) {
            // Werfen der Exception
            throw new NoNegativeInitialBalance("Ein Konto muss zur Erstellung gedeckt sein.");
        }
        this.balance = balance;
    }

    public double getBalance() {
        return this.balance;
    }

    public void printBalance() {
        System.out.println("Aktuelles Guthaben:" + this.balance);
    }

    // ExceedWithdrawl ist eine Kindklasse von RuntimeException.
    // Das bedeutet, dass diese Ausnahme eine
    // "unchecked exception" ist und nicht explizit in der
    // Methodensignatur deklariert werden muss, aber sie kann
    // dennoch gefangen werden.
    public void withdraw(double amount) {
        try{

            if (amount > this.balance) {
                // Werfen der Exception
                throw new ExceedWithdrawl("Fehler: Nicht genügend Guthaben vorhanden! Versucht: " + amount + ", Verfügbar: " + this.balance);
            } else {
                this.balance -= amount;
            }
        } catch (ExceedWithdrawl ce){
            System.out.println(ce.getMessage());
        }
    }
}
```

```java
// Diese Klasse ist eine benutzerdefinierte Ausnahme
// (Exception). Sie wird verwendet, um einen bestimmten
// Fehlerzustand in der Anwendung zu signalisieren: Dass
// der abzuhebende Betrag den Kontostand übersteigt.
// Sie erbt von der Standard-Java-Klasse `RuntimeException`,
// was bedeutet, dass es sich um eine nicht-überprüfte
// Ausnahme handelt (unchecked exception). Das bedeutet,
// dass Methoden, die diese Ausnahme werfen könnten, dies
// nicht explizit in ihrer Signatur deklarieren müssen
// (`throws ExceedWithdrawl`). Aufrufende Methoden sind
// nicht gezwungen, diese Ausnahme abzufangen oder
// weiterzuwerfen, obwohl sie es tun können, wenn sie
// den Fehler behandeln möchten. RuntimeExceptions werden
// oft für Programmierfehler oder unerwartete Zustände
// verwendet, die zur Laufzeit auftreten können. Eine
// Runtime Exception bricht einen try-Block nicht ab.
public class ExceedWithdrawl extends RuntimeException {
    public ExceedWithdrawl(String message) {
        super(message);
    }
}

```

```java
// Diese Klasse ist eine benutzerdefinierte Ausnahme (Exception).
// Sie wird verwendet, um einen bestimmten Fehlerzustand in
// der Anwendung zu signalisieren: Dass ein negativer
// Startbetrag übergeben wurde. Sie erbt von der
// Standard-Java-Klasse `Exception`, was bedeutet,
// dass es sich um eine überprüfte Ausnahme handelt
// (checked exception). Das bedeutet, dass Methoden, die diese
// Ausnahme werfen könnten, dies in ihrer Signatur deklarieren
// müssen (`throws NoNegativeInitialBalance`) und aufrufende
// Methoden diese Ausnahme entweder abfangen (`catch`) oder
// ebenfalls weiterwerfen müssen. Wird eine Checked Exception
// geworfen, wird der try-Block an dieser Stelle beendet.
public class NoNegativeInitialBalance extends Exception {
    public NoNegativeInitialBalance(String message) {
        super(message);
    }
}

```

ENUM - Fahrzeug.java

```java
public class Fahrzeug {
    private final String Kennzeichen;
    private final Fahrzeugtyp fahrzeugtyp;
    private final int maxGeschwindigkeit;

    public Fahrzeug(String kennzeichen, Fahrzeugtyp fahrzeugtyp) {
        this.Kennzeichen = kennzeichen;
        this.fahrzeugtyp = fahrzeugtyp;
        this.maxGeschwindigkeit = fahrzeugtyp.getMaxGeschwindigkeit();
    }

    public String getKennzeichen() {
        return Kennzeichen;
    }

    public Fahrzeugtyp getFahrzeugtyp() {
        return fahrzeugtyp;
    }

    public int getMaxGeschwindigkeit() {
        return maxGeschwindigkeit;
    }



}

```

ENUM - Fahrzeuge.java

```java
public class Fahrzeuge {
    public static void main(String[] args) {
        Fahrzeug a = new Fahrzeug("BASN332", Fahrzeugtyp.AUTO);
        Fahrzeug b = new Fahrzeug("ERZZ123", Fahrzeugtyp.LKW);

        Fahrzeugtyp.listeFahrzeugtypen();
    }
}

```

ENUM - Fahrzeugtyp.java

```java
public enum Fahrzeugtyp {
    AUTO(200),
    MOTORRAD(300),
    LKW(80),
    FAHRRAD(25);

    private final int maxGeschwindigkeit;

    Fahrzeugtyp(int max){
        maxGeschwindigkeit = max;
    }

    public int getMaxGeschwindigkeit(){
        return maxGeschwindigkeit;
    }

    //geht alle Enums des Fahrzeugtyps durch → ForEach
    public static void listeFahrzeugtypen() {
        for (Fahrzeugtyp typ : Fahrzeugtyp.values()) {
            System.out.println(typ + " hat eine Höchstgeschwindigkeit von " + typ.getMaxGeschwindigkeit() + " km/h.");
        }
    }
}

```

VERERBUNG - Test.java

```java
public class Test {
    public static void main(String[] args) {
        // Vogel ist eine abstrakte Klasse. Eine Instanziierung wie
        // Vogel v1  = new Vogel();
        // würde also nicht funktionieren. Allerdings lässt sich ein Array der
        // Elternklasse anlegen. In diesen Array können dann alle Subklassen der
        // Elternklasse gespeichert werden. In diesem Beispiel können "Papagei"
        // und "Wellensittich" in einem Array von "Vogel" gespeichert werden,
        // da sie beide von der Klasse "Vogel" abstammen.
        Vogel[] nest = new Vogel[5];

        nest[0] = new Papagei("Peter", 21.4, "gruen");
        nest[1] = new Wellensittich("Ruediger", 25.0, 4.7);

        // Da in der Elternklasse definiert ist, dass es eine Methode "toString"
        // gibt kann diese Methode auch auf den im Array aus "Vogel" gespeicherten
        // Objekten ausgeführt werden. Je nachdem ob das betroffene Objekt dann ein
        // "Papagei" oder ein "Wellensittich" ist, wird die entsprechende Version
        // der Methode "toString" ausgeführt. Der Aufruf der Methode "sprechen",
        // die nur in "Papagei" definiert wurde, würde dementsprechend nicht
        // funktionieren.
        for(int i = 0; i < nest.length; i++){
            // Überspringen von leeren Zellen des Arrays
            if(nest[i]== null) break;
            // Hier steht eigentlich:
            // System.out.println(nest[i].toString());
            // Der Methodenaufruf ".toString()" kann hier allerdings ignoriert werden.
            System.out.println(nest[i]);
            // System.out.println(nest[i].sprechen()); // Funktioniert nicht!
        }

        // Konsole:
        // Peter (21.4 km/h) - Farbe: gruen
        // Ruediger (25.0 km/h)

        System.out.println(nest[0].getAnzahl()); // Konsole: 2

    }
}

```

VERERBUNG - Tier.java

```java
// Ein Interface gibt Methoden vor, die eine Klasse haben muss, die das entsprechende
// Interface implementiert. Der Inhalt der Methoden des Interfaces werden dann erst
// in der Klasse definiert. Da Methoden aus einem Interface immer public und abstract
// sind kann auf diese Schlüsselwörter verzichtet werden. In einem Interface können
// keine Instanzvariablen definiert werden! Definiert Variablen sind immer Konstanten
// und erhalten die Schlüsselwörter "final", "static" und "public".
public interface Tier {
    void fressen();
    void schlafen();

    // In einem Interface können Default-Methoden definiert werden. Damit können
    // Standardimplementierungen bereits im Interface hinterlegt werden. Die Methode
    // muss dazu mit dem Schlüsselwort default versehen werden.
    default void tagesablauf(){
        fressen();
        schlafen();
    }

    // Außerdem können statische Methoden definiert werden. Diese Methoden können
    // ohne ein instanziiertes Objekt aufgerufen werden.
    static void printHello(){
        System.out.println("Hello World!");
    }
}
```

```java
// Implements: Das Interface "Tier" wird auf die Klasse "Vogel" angewandt. Die
// Klasse "Vogel" muss somit alle Methoden aus dem Interface "Tier" enthalten.
// Da "Vogel" abstract ist, würde kein Fehler auftreten, wenn eine Methode aus
// dem Interface nicht implementiert wäre. Dafür wird dann aber in den Subklassen
// erzwungen, dass die entsprechende Methode definiert ist. Vorteil eines Interface
// ist die Mehrfachvererbung. Eine abstrakte Klasse kann zwar mehrere Kindklassen
// haben, aber immer nur eine Elternklasse. Interfaces hingegen können mehrfach auf
// Klassen angewandt werden. Somit ist es hier möglich sowohl das Interface "Tier"
// als auch das Interface "KannFliegen" und "HatFluegel" auf die Klasse "Vogel"
// anzuwenden.

// Auch diese Klasse ist abstrakt, da sie in diesem Fall zu unspezifisch ist.
public abstract class Vogel extends Lebewesen implements Tier, KannFliegen, HatFluegel {

    // Static: Die Variable ist unabhängig von der Instanz und ist für jede Instanz der
    // Klasse Vogel gleich. Das bedeutet egal auf welcher Instanz von "Vogel" die
    // Variable "anzahl" verwendet wird, befindet sich immer der gleiche Wert dahinter.
    // Es wird gespeichert wie viele Vögel instanziiert wurden.
    private static int anzahl;
    private double schnelligkeit;

    public Vogel(String name, double schnelligkeit){
        // Mit "super" wird die Elternklasse angesprochen. In diesem Fall der Konstruktor
        // der Klasse "Lebewesen"
        super(name);
        this.schnelligkeit = schnelligkeit;
        anzahl++;
    }

    // Überladung einer Methode: Die Methode "bewegen" existiert bereits in der
    // Elternklasse "Lebewesen". Dort allerdings mit einem anderen Methodenkopf.
    // In der Elternklasse nimmt die Methode keine Parameter an.
    // Hier schon. Somit wird die Methode hier nicht überschrieben, sondern überladen.
    // Ein Objekt von "Vogel" kann dadurch beide Varianten aufrufen und verwenden.
    // Siehe PatientTest.
    public void bewegen(String richtung) {
        System.out.println("Das Lebewesen bewegt sich " + richtung + ".");
    }

    public void fliegen() {
        System.out.println("Der Vogel fliegt durch die Luft.");
    }

    public void ausbreiten(){
        System.out.println("Fluegel werden ausgebreitet.");
    }

    // Hier wird die Methode "toString" aus der Klasse "Lebewesen" überschrieben.
    public String toString(){
        // Hier wird mit "super" die Methode von der Elternklasse aufgerufen.
        return super.toString() + " (" + schnelligkeit + " km/h)";
    }

    public void sterben() {
        System.out.println("Der Vogel fällt vom Himmel");
    }

    public int getAnzahl(){
        return anzahl;
    }

}

```

```java
// Durch das "final" wird festgelegt, dass die Klasse "Papagei" nicht weiter vererbt
// werden kann. Eine Klassendefinition wie
// public class GruenSchnabelPapagei extends Papagei {...}
// wäre also nicht möglich.
// Ebenso können auch Methoden vom überschreiben geschützt werden, indem sie in der
// Elternklasse mit einem "final" versehen werden, können sie in Subklassen nicht
// überschrieben werden.
// Zur Info: "final" sorgt bei Variablen dafür, dass die Werte zu Konstanten werden.
public final class Papagei extends Vogel {
    private String farbe;

    public Papagei(String name, double schnelligkeit, String farbe) {
        super(name, schnelligkeit);
        this.farbe = farbe;
    }

    public void fressen() {
        System.out.println("Der Papagei frisst Körner.");
    }

    public void schlafen() {
        System.out.println("Der Papagei schläft im Nest.");
    }


    // Methodenüberladung
    public void sprechen() {
        System.out.println("Der Papagei spricht!");
    }

    public void sprechen(String satz) {
        System.out.println("Der Papagei sagt: " + satz);
    }

    public String toString() {
        return super.toString() + " - Farbe: " + farbe;
    }
}

```

VERERBUNG - Lebewesen.java

```java
// Abstract: Es kann keine Instanz der Klasse "Lebewesen" erstellt werden.
// Lediglich von der Klasse "Lebewesen" abstammende Klassen wie "Vogel" können
// instanziiert werden. "Lebewesen" legt lediglich die Grundstruktur fest.
// Das macht Sinn, weil ein Vogel immer ein Lebewesen ist.
// Damit die nicht instanziiert werden kann, macht man sie abstrakt.
public abstract class Lebewesen {
    private String name;

    public Lebewesen(String name) {
        this.name = name;
    }

    public void bewegen() {
        System.out.println("Das Lebewesen bewegt sich.");
    }

    // Jede Klasse stammt von der Klasse "Object" ab,
    // hat also prinzipiell immer ein "extends Object" an der Klassendefinition.
    // Die Klasse "Object" hat eine Methode "toString", die hier in diesem Fall
    // überschrieben wird. Durch das Überschreiben wird beim Aufruf
    // "Lebewesen.toString()" nicht mehr die ursprüngliche Methode der Klasse
    // "Object" verwendet, sondern die hier überschriebene Variante. Genauso kann
    // diese und alle anderen Methoden auch in Klassen, die von Lebewesen abstammen
    // überschrieben werden. Es gilt prinzipiell,
    // dass die erste verfügbare Variante der Methode verwendet wird.
    public String toString(){
        return name;
    }

    // Auch Methoden können abstract sein. Damit wird nur vorgegeben, dass die
    // Methode in der Subklasse existieren muss. Es gibt also in "Vogel" bzw.
    // da "Vogel" ebenfalls abstract ist spätestens in "Papagei" zwingend eine
    // Methode "sterben". Dort muss für diese Methode dann auch Syntax definiert
    // sein. Damit eine Methode abstract sein kann, muss die Klasse auch abstract
    // sein.
    public abstract void sterben();


}

```

VERERBUNG - HatFluegel.java

```java
public interface HatFluegel {
    void ausbreiten();
}

```

VERERBUNG - KannFliegen.java

```java
// Siehe Tier
public interface KannFliegen {
    void fliegen();
}

```

VERERBUNG - Wellensittich.java

```java
public class Wellensittich extends Vogel {
    private double schnabellaenge;

    public Wellensittich(String name, double schnelligkeit, double schnabellaenge) {
        super(name, schnelligkeit);
        this.schnabellaenge = schnabellaenge;
    }

    public void fressen() {
        System.out.println("Wellensittich frisst Beeren.");
    }

    public void schlafen() {
        System.out.println("Wellensittich schlaeft.");
    }
}

```

COMBINED - Mensch.java

```java
// Siehe Person Interface
public interface Mensch {
    public int getAlter();
}

```

```java
// Ein Interface gibt Methoden vor, die eine Klasse haben muss, die das entsprechende
// Interface implementiert. Der Inhalt der Methoden des Interfaces werden dann erst
// in der Klasse definiert. Da Methoden aus einem Interface immer public und abstract
// sind kann auf diese Schlüsselwörter verzichtet werden. In einem Interface können
// keine Instanzvariablen definiert werden! Definiert Variablen sind immer Konstanten
// und erhalten die Schlüsselwörter "final", "static" und "public".

public interface Person {
    String getName();
    String getVorname();
    // In einem Interface können Default-Methoden definiert werden. Damit können
    // Standardimplementierungen bereits im Interface hinterlegt werden. Die Methode
    // muss dazu mit dem Schlüsselwort default versehen werden.
    default String getIdentitaet(){
        return getName() + ", " + getVorname();
    }

    // Außerdem können statische Methoden definiert werden. Diese Methoden können ohne
    // ein instanziiertes Objekt aufgerufen werden.
    static void printHello(){
        System.out.println("Hello World!");
    }
}

```

```java
// Abstract: Es kann keine Instanz der Klasse Patient erstellt werden.
// Lediglich von der Klasse Patient abstammende Klassen wie "Kassenpatient" oder
// "Privatpatient" können instanziiert werden. Patient legt lediglich die Grundstruktur
// fest. Das macht Sinn, weil ein Patient immer entweder privatversichert oder
// kassenversichert ist. Allerdings teilen sowohl private als auch Kassenpatienten viele
// Eigenschaften. Deshalb sammelt man diese Eigenschaften in der Grundklasse Patient.
// Damit die nicht instanziiert werden kann, macht man sie abstrakt.

// Implements: Das Interface "Person" wird auf die Klasse Patient angewandt. Die Klasse
// "Patient" muss somit alle Methoden aus dem Interface "Person" enthalten. Da "Patient"
// abstract ist, würde kein Fehler auftreten, wenn eine Methode aus dem Interface nicht
// implementiert wäre. Dafür wird dann aber in den Subklassen erzwungen, dass die
// entsprechende Methode definiert ist. Vorteil eines Interface ist die
// Mehrfachvererbung. Eine abstrakte Klasse kann zwar mehrere Kindklassen haben, aber
// immer nur eine Elternklasse. Interfaces hingegen können mehrfach auf Klassen
// angewandt werden. Somit ist es hier möglich sowohl das Interface "Person" als auch
// das Interface "Mensch" auf die Klasse "Patient" anzuwenden.
// Außerdem wird das Interface "Comparable" angewandt um die Klasse "Patient" in einem
// Set verwenden zu können

// PraxisVollException ist eine Kindklasse von Exception und NICHT von RuntimeException.
// Das bedeutet, dass diese Ausnahme explizit in der Methodensignatur deklariert werden
// muss und vom aufrufenden Code entweder gefangen oder weiter geworfen werden muss.

// UngueltigerNameException ist eine Kindklasse von RuntimeException.
// Das bedeutet, dass diese Ausnahme eine "unchecked exception" ist und nicht explizit
// in der Methodensignatur deklariert werden muss, aber sie kann dennoch gefangen werden.

abstract class Patient implements Person, Mensch, Comparable<Patient> {
    private String name;
    private String vorname;
    // "int" ist ein elementarer Datentyp
    private int alter;
    // "Integer" ist der dazugehörige Wrapper-Datentyp.
    // Wrapper-Datentypen sind Referenz-Datentypen und erben somit auch von der
    // Klasse "Object".
    // Dadurch können Datentypen wie "String" und "Integer" in einem Array vom Typ
    // "Object" kombiniert werden, was mit "String" und "int" nicht möglich wäre.
    private Integer patientNr;

    // Static: Die Variable ist unabhängig von der Instanz und ist für jede Instanz der
    // Klasse Patient gleich. Das bedeutet egal auf welcher Instanz von "Patient" die
    // Variable "anzahl" verwendet wird, befindet sich immer der gleiche Wert dahinter.
    // Hier macht das Sinn, da hier eine allgemeine Information gespeichert werden soll,
    // die unabhängig vom eigentlichen Patienten ist. Es wird gespeichert wie viele
    // Patienten instanziiert wurden.
    private static int anzahl;

    // Dies ist der Konstruktor der abstrakten Klasse Patient.
    // Er ist für die Initialisierung der grundlegenden Patientendaten zuständig.

    // WICHTIG: Dieser Konstruktor deklariert explizit, dass er eine
    // 'PraxisVollException' werfen kann. Da 'PraxisVollException' eine checked
    // exception ist, MUSS sie hier deklariert werden, damit der Compiler weiß, dass
    // diese Ausnahme auftreten kann. Die 'UngueltigerNameException' ist eine
    // unchecked exception, daher muss sie nicht explizit mit 'throws' deklariert
    // werden. Zur Klarheit wird es hier dennoch gemacht.

    // Deklaration mit 'throws' ist hier eine gute Praxis, um den Aufrufer zu
    // informieren, dass diese Fehler auftreten können.
    public Patient(String name, String vorname, int alter) throws PraxisVollException,
UngueltigerNameException {
        // Dieser Block prüft, ob die maximale Anzahl von Patienten (hier 8)
        // erreicht ist.
        // Wenn ja, wird eine 'PraxisVollException' geworfen.
        // Da 'PraxisVollException' eine checked exception ist, MUSS der Code, der
        // diesen Konstruktor aufruft, diese Ausnahme behandeln (entweder abfangen oder
```

```
        // weiterwerfen). Das Werfen einer Ausnahme unterbricht die normale Ausführung
        // des Konstruktors. Der Code nach dem 'throw' wird nicht ausgeführt, wenn die
        // Bedingung erfüllt ist.
        if(anzahl >= 8){
            throw new PraxisVollException("Maximale Patientenanzahl erreicht");
        } else {
            // Dieser Block prüft die Gültigkeit des übergebenen Namens.
            // Wenn der Name null oder leer ist, wird eine 'UngueltigerNameException'
            // geworfen. Da 'UngueltigerNameException' eine unchecked exception ist,
            // MUSS der Aufrufer sie nicht behandeln, aber er KANN es tun, wie im
            // 'main'-Methodenbeispiel gezeigt.
            if (name == null || name.trim().isEmpty()) {
                throw new UngueltigerNameException("Der Name darf nicht leer sein.");
            }
            // Dieser Block prüft, ob der Name Zahlen enthält.
            // Wenn ja, wird ebenfalls eine 'UngueltigerNameException' geworfen.
            else if (name.matches(".*\\d+.*")) {
                throw new UngueltigerNameException("Der Name darf keine Zahlen enthalten
."");
            }
            // Wenn keine der Fehlerbedingungen erfüllt ist, wird der Name zugewiesen.
            else {
                this.name = name;
            }
            // Wenn keine Ausnahmen geworfen wurden, wird der Rest des
            // Konstruktors ausgeführt.
            this.vorname = vorname;
            this.alter = alter;

            // Die Anzahl der Patienten wird nur erhöht, wenn der Konstruktor
            // erfolgreich durchläuft (keine Ausnahme geworfen wurde).
            anzahl++;
            setPatientNr();
        }
    }

    public String getName(){
        return this.name;
    }

    public String getVorname(){
        return this.vorname;
    }

    public int getAlter(){
        return this.alter;
    }

    public int getPatientNr(){
        // Um an den elementaren Datentyp einer Wrapper-Klasse zu gelangen muss man
        // diesen theoretisch "unboxen". Das passiert mittlerweile allerdings
        // automatisch.
        // return this.patientNr.intValue(); // Nicht notwendig
        return this.patientNr; // Auto-Unboxing
    }

    public static int getAnzahl(){
        return anzahl;
    }

    public void setPatientNr(){
        // Um den elementaren Datentyp einer Wrapper-Klasse zu setzen, muss man diesen
        // theoretisch "boxen". Das passiert mittlerweile allerdings automatisch.
        // Nicht notwendig:
        // this.patientNr = Integer.valueOf((int)((Math.random() * 100) + 1)*42);
        this.patientNr = (int)((Math.random() * 100) + 1)*42; // Autoboxing
    }
```

```java

    // Jede Klasse stammt von der Klasse "Object" ab,
    // hat also prinzipiell immer ein "extends Object" an der Klassendefinition.
    // Die Klasse "Object" hat eine Methode "toString", die hier in diesem Fall
    // überschrieben wird. Durch das Überschreiben wird beim Aufruf
    // "Patient.toString()" nicht mehr die ursprüngliche Methode der Klasse
    // "Object" verwendet, sondern die hier überschriebene Variante. Genauso
    // kann diese und alle anderen Methoden auch in Klassen, die von Patient
    // abstammen überschrieben werden. Es gilt prinzipiell, dass die erste
    // verfügbare Variante der Methode verwendet wird.
    public String toString(){
        return this.name + ", " + this.vorname + ", " + this.alter + ", " + this.
patientNr;
    }

    // Auch Methoden können abstract sein. Damit wird nur vorgegeben, dass die Methode
    // in der Subklasse existieren muss. Es gibt also in "Kassenpatient" und
    // "Privatpatient" zwingend eine Methode "printAbrechnung". Dort muss für diese
    // Methode dann auch Syntax definiert sein. Damit eine Methode abstract sein kann,
    // muss die Klasse auch abstract sein.
    public abstract void printAbrechnung();

    public int compareTo(Patient p){
        return Integer.compare(p.getPatientNr(), this.getPatientNr());
    }
}


```

```java
import java.util.Iterator;
import java.util.LinkedList;
import java.util.TreeSet;

public class PatientTest {
    public static void main(String[] args) {
        // Patient ist eine abstrakte Klasse. Eine Instanziierung wie
        // Patient p1  = new Patient();
        // würde also nicht funktionieren. Allerdings lässt sich ein Array der
        // Elternklasse anlegen. In diesen Array können dann alle Subklassen der
        // Elternklasse gespeichert werden.
        // In diesem Beispiel können "Kassenpatient" und "Privatpatient" in einem
        // Array von Patienten gespeichert werden, da sie beide von der Klasse "Patient"
        // abstammen.
        Patient[] patienten = new Patient[8];

        // Der 'try'-Block umschließt Code, der potenziell eine Ausnahme (Exception)
        // auslösen kann. In diesem Fall kann der Konstruktor der Klasse "Patient"
        // Exceptions werfen. Da die Exceptions in den Klassen "Kassenpatient" und
        // "Privatpatient" nicht behandelt werden, werden die Exceptions weiter geworfen
        // und müssen teilweise an dieser Stelle nun behandelt werden. Nur teilweise ein
        // Muss da eine Exception von RuntimeException abstammt. Siehe Patient.
        try {
            patienten[0] = new Privatpatient("Meier", "Hans", 45);
            patienten[1] = new Kassenpatient("Müller", "Peter", 23, true);

            // Wenn die folgenden Zeilen ausgeführt werden würden, würde das eine
            // "PraxisVollException" werfen,
            // da in "Patient" definiert ist, dass ab 6 Patienten die Praxis voll ist.
            // patienten[2] = new Kassenpatient("Müller", "Daniela", 51, false);
            // patienten[3] = new Kassenpatient("Müller", "Daniela", 51, false);
            // patienten[4] = new Kassenpatient("Müller", "Daniela", 51, false);
            // patienten[5] = new Kassenpatient("Müller", "Daniela", 51, false);
            // patienten[6] = new Kassenpatient("Müller", "Daniela", 51, false);
            // patienten[7] = new Kassenpatient("Müller", "Daniela", 51, false);

            // Wenn die folgende Zeile ausgeführt werden würde, würde das eine
            // "UngueltigerNameException" werfen, da in "Patient" definiert ist, dass
            // ein Name keine Zahl enthalten darf
            // patienten[2] = new Kassenpatient("Müller02", "Daniela", 51, false);
        }

        // Die Reihenfolge der 'catch'-Blöcke ist wichtig: Spezifischere Ausnahmen
        // sollten zuerst gefangen werden, gefolgt von allgemeineren Ausnahmen. In diesem
        // Fall sind beide Ausnahmen spezifisch und werden einzeln abgefangen.
        // Alternativ könnte man auch beide Exceptions allgemein abfangen, indem man mit
        // der Elternklasse "Exception" arbeitet.

        // Der erste 'catch'-Block fängt spezifisch die 'UngueltigerNameException' ab.
        // Wenn eine 'UngueltigerNameException' im 'try'-Block geworfen wird, wird der
        // Code in diesem 'catch'-Block ausgeführt.
        catch (UngueltigerNameException e) {
            // Hier wird die Fehlermeldung der Ausnahme auf der Konsole ausgegeben.
            System.out.println("Fehler: " + e.getMessage());
        }

        // Der zweite 'catch'-Block fängt spezifisch die 'PraxisVollException' ab.
        // Wenn eine 'PraxisVollException' im 'try'-Block geworfen wird (und keine
        // 'UngueltigerNameException' zuvor gefangen wurde), wird der Code in diesem
        // 'catch'-Block ausgeführt.
        catch (PraxisVollException e) {
            // Auch hier wird die Fehlermeldung der Ausnahme auf der Konsole ausgegeben.
            e.printStackTrace();
        }

        // Allgemeines Abfangen jedes Fehlers
        catch (Exception e) {
```

```java
            e.printStackTrace();
        }

        // Nach dem 'try-catch'-Block wird die Programmausführung fortgesetzt,
        // unabhängig davon, ob eine Ausnahme aufgetreten ist oder nicht (solange sie
        // gefangen wurde). Wenn eine Ausnahme auftritt und gefangen wird, wird der Rest
        // des 'try'-Blocks übersprungen, aber der Code nach dem 'catch'-Block wird
        // ausgeführt.


        // Da in der Elternklasse definiert ist, dass es eine Methode "toString" gibt
        // kann diese Methode auch auf den im Array aus Patienten gespeicherten Objekten
        // ausgeführt werden. Je nachdem ob das betroffene Objekt dann ein
        // "Kassenpatient" oder ein "Privatpatient" ist, wird die entsprechende Version
        // der Methode "toString" ausgeführt. Der Aufruf der Methode
        // "getFamilienversichert", die nur in "Kassenpatient" definiert wurde, würde
        // dementsprechend nicht funktionieren.
        for(int i = 0; i < patienten.length; i++){
            // Überspringen von leeren Zellen des Arrays
            if(patienten[i]== null) break;
            // Hier steht eigentlich:
            // System.out.println(patienten[i].toString());
            // Der Methodenaufruf ".toString()" kann hier allerdings ignoriert werden.
            System.out.println(patienten[i]);

            // Funktioniert nicht!
            // System.out.println(patienten[i].getFamilienversichert());
        }

        // Konsole:
        // Meier, Hans, 45, 0
        // Müller, Peter, 23, 0, familienversichert

        System.out.println(patienten[0].getAnzahl()); // Konsole: 2

        try {
            Kassenpatient kp1 = new Kassenpatient("Mustermann", "Max", 34, false);

            // Ausführen der Methode "getName" aus der Elternklasse
            System.out.println(kp1.getName()); // Konsole: Mustermann
            // Ausführen der, in "Kassenpatient" überladenen, Methode "getName"
            System.out.println(kp1.getName("Herr")); // Konsole: Name: Mustermann

            // Aufruf der Default-Methode aus dem Interface "Person"
            System.out.println(kp1.getIdentitaet()); // Konsole: Mustermann, Max

        } catch (UngueltigerNameException e) {
            System.out.println("Fehler: " + e.getMessage());
        } catch (PraxisVollException e) {
            e.printStackTrace();
        }


        // Aufruf der statischen Methode aus dem Interface "Person"
        Person.printHello(); // Konsole: Hello World!


        try {
            // Erstellen eines Zweibettzimmers. Es wird angenommen, dass nur gleiche
            // Patiententypen gemeinsam in ein Zimmer dürfen
            Kassenpatient kpA = new Kassenpatient("Schmidt", "Andreas", 44, false);
            Kassenpatient kpB = new Kassenpatient("Hartmann", "Fixi", 22, true);

            // In den </> Klammern wird der generischen Klasse der Datentyp übergeben
            ZweiBettZimmer<Kassenpatient> zimmer1 = new ZweiBettZimmer<>(kpA, kpB);

            // Da nur Patienten des gleichen Typs in ein Zimmer dürfen, wäre folgender
```

```
135             // Aufruf nicht möglich
136             //  Kassenpatient kpA = new Kassenpatient("Schmidt", "Andreas", 44, false);
137             //  Privatpatient ppA = new Kassenpatient("Hartmann", "Fixi", 22);
138             //  ZweiBettZimmer<Kassenpatient> zimmer1 = new ZweiBettZimmer<>(kpA, ppA);
139
140             zimmer1.printBelegung();
141             // Konsole:
142             // Zimmerbelegung:
143             // Fensterseite: Schmidt, Andreas, 44, 3528, nicht familienversichert
144             // Wandseite: Hartmann, Fixi, 22, 294, familienversichert
145
146             printFensterbett(zimmer1);
147             // Konsole: Schmidt, Andreas, 44, 2394, nicht familienversichert
148
149             printWandbett(zimmer1);
150             // Konsole: Hartmann, Fixi, 22, 630, familienversichert
151         }
152         catch (PraxisVollException e){
153             e.printStackTrace();
154         }
155
156         // Alternativ zu einem Array können mehrere Referenzobjekte in einer Collection
157         // gespeichert werden. Collections können dynamisch wachsen und schrumpfen und
158         // benötigen daher keine vordefinierte Größe.
159         // Die Größe kann sich während der Laufzeit verändern.
160         // Die Collection kann dabei entweder eine "List" oder ein "Set" sein.
161         // Beide Varianten haben eigene Eigenschaften.
162         // List: Geordnete Folge von Elementen
163         // Set: Sammlung von Objekten, in der jedes Objekt nur einmal vorkommen darf
164         // -> Mit einem SortedSet (Kindklasse zu Set) ist auch ein Set in einer
165         // geordneten Reihenfolge
166         try {
167             Privatpatient pp1 = new Privatpatient("Meier", "Michael", 45);
168             Kassenpatient kp1 = new Kassenpatient("Meier", "Max", 34, false);
169
170             // Abstammung von TreeSet: Collection -> Set -> SortedSet -> TreeSet
171             // Da TreeSet also ein SortedSet ist muss die Klasse, aus der ein TreeSet
172             // erstellt werden soll, das Interface "Comparable<T>" implementieren
173             // (Siehe Patient)
174             // Das TreeSet fügt Elemente automatisch in der richtigen, sortierten
175             // Reihenfolge aus.
176             TreeSet<Patient> patientSet = new TreeSet<>();
177             patientSet.add(pp1);
178             patientSet.add(kp1);
179
180             // Jede Collection enthält einen Iterator, um über die Collection zu
181             // iterieren
182             // und alle Elemente zu erreichen
183             Iterator<Patient> it = patientSet.iterator();
184
185             // Jeder Iterator hat die Methoden "next", "hasNext" und "remove".
186             // Damit kann über jedes Element iteriert werden
187             // While-Schleife
188             while(it.hasNext()){
189                 System.out.println(it.next());
190             }
191             // For-Schleife (ausführlich)
192             for(Iterator<Patient> i = patientSet.iterator(); i.hasNext();){
193                 System.out.println(i.next());
194             }
195             // For-Schleife (simpel/kurz)
196             for(Patient p : patientSet){
197                 System.out.println(p);
198             }
199
200             // Konsole bei allen drei Varianten:
201             // Meier, Michael, 45, 3612
```

```
202             // Meier, Max, 34, 882, nicht familienversichert
203
204             // Mit "size" lässt sich die Größe einer Collection ausgeben
205             System.out.println(patientSet.size()); // Konsole: 2
206
207             // Da ein Set keine Duplikate enthalten darf, hat die folgende Zeile
208             // keine Auswirkung auf die Collection. Es bleiben weiterhin 2 Elemente.
209             patientSet.add(pp1);
210             System.out.println(patientSet.size()); // Konsole: 2
211
212
213             // Gleicher Programmablauf mit einer LinkedList:
214
215             // Abstammung von LinkedList: Collection -> List -> LinkedList
216             // Die List fügt Elemente immer am Ende an.
217             LinkedList<Patient> patientList = new LinkedList<>();
218             patientList.add(pp1);
219             patientList.add(kp1);
220
221             // Jede Collection enthält einen Iterator, um über die Collection zu
222             // iterieren
223             // und alle Elemente zu erreichen
224             Iterator<Patient> it2 = patientList.iterator();
225
226             // Jeder Iterator hat die Methoden "next", "hasNext" und "remove".
227             // Damit kann über jedes Element iteriert werden
228             // While-Schleife
229             while(it.hasNext()){
230                 System.out.println(it2.next());
231             }
232             // For-Schleife (ausführlich)
233             for(Iterator<Patient> i = patientList.iterator(); i.hasNext();){
234                 System.out.println(i.next());
235             }
236             // For-Schleife (simpel/kurz)
237             for(Patient p : patientList){
238                 System.out.println(p);
239             }
240
241             // Konsole bei allen drei Varianten:
242             // Meier, Michael, 45, 3612
243             // Meier, Max, 34, 882, nicht familienversichert
244
245             // Mit "size" lässt sich die Größe einer Collection ausgeben
246             System.out.println(patientList.size()); // Konsole: 2
247
248             // Anders als bei einem Set wird bei einer List nicht auf Duplikate
249             // überprüft
250             patientList.add(pp1);
251             System.out.println(patientList.size()); // Konsole: 3
252
253         }
254         catch (PraxisVollException e){
255             e.printStackTrace();
256         }
257
258     }
259
260     // Diese Methode verwendet keinen generischen Datentyp für die Übergabe des
261     // Parameters. Da bei generischen Datentypen die Vererbung allerdings irrelevant
262     // ist, kann diese Methode niemals aufgerufen werden. Das liegt daran, dass es
263     // kein ZweiBettZimmer vom Typ "Patient" geben kann, da "Patient" abstrakt ist.
264     // Es kann also nur "Kassenpatient" oder "Privatpatient" sein. Hätte die Methode
265     // "ZweiBettZimmer<Privatpatient> bett" als Parameter könnte die Methode aufgerufen
266     // awerden. Allerdings nicht mit einem ZweiBettZimmer vom Typ "Kassenpatient".
267     // Um das zu lösen gibt es zwei Möglichkeiten:
268     // 1. Methode mit generischem Datentyp
```

```java
    // 2. Methode mit einer Wildcard
    public static boolean gleicherName(ZweiBettZimmer<Patient> bett){
        return bett.getFensterSeite().getName().equals(bett.getWandSeite().getName());
    }

    // Auch Methoden können generische Datentypen verwenden. Diese Methode kann sowohl
    // ein Zweibettzimmer mit Privatpatienten, als auch eins mit Kassenpatienten
    // verarbeiten.
    public static <T extends Patient> void printFensterbett(ZweiBettZimmer<T> bett){
        System.out.println(bett.getFensterSeite().toString());
    }

    // Alternativ kan auch mit einer Wildcard gearbeitet werden. Damit wird die doppelte
    // Prüfung des korrekten generischen Datentyps vermieden. Es wird nur bei der
    // Instanziierung der generischen Klasse geprüft.
    public static void printWandbett(ZweiBettZimmer<?> bett){
        System.out.println(bett.getWandSeite().toString());
    }
}

```

```java
public class Kassenpatient extends Patient {
    private boolean familienversichert;

    /**
     * Konstruktor für die Klasse Kassenpatient.
     * Initialisiert einen neuen Kassenpatienten.
     *
     * WICHTIG: Dieser Konstruktor MUSS "throws PraxisVollException" deklarieren,
     * weil der Aufruf des Elternkonstruktors mit "super(name, vorname, alter)"
     * eine PraxisVollException werfen kann. PraxisVollException MUSS behandelt werden:
     * Siehe Patient
     *
     * Der Kassenpatient-Konstruktor wirft die Ausnahme nicht selbst direkt,
     * sondern er muss deklarieren, dass die Ausnahme, die vom "super()"-Aufruf
     * kommen kann, an den Aufrufer des Kassenpatient-Konstruktors weitergegeben wird.
     *
     * @param name Der Name des Patienten.
     * @param vorname Der Vorname des Patienten.
     * @param alter Das Alter des Patienten.
     * @param familienversichert Gibt an, ob der Patient familienversichert ist.
     * @throws PraxisVollException Wird vom Elternkonstruktor geworfen und hier
     *                             weitergegeben, wenn die Praxis voll ist.
     */
    public Kassenpatient(String name, String vorname, int alter, boolean
familienversichert) throws PraxisVollException {
        // Mit "super(...)" wird der Konstruktor der Elternklasse aufgerufen.
        // Dieser Aufruf kann eine PraxisVollException werfen, da der
        // Elternkonstruktor dies deklariert. Deshalb muss auch dieser
        // Konstruktor dieAusnahme deklarieren.
        super(name, vorname, alter);

        this.familienversichert = familienversichert;
    }

    // Hier wird erneut die Methode "toString",
    // die bereits in "Patient" die Version aus der Klasse "Object" überschreibt,
    // überschrieben.

    public String toString(){
        String familie;
        if (familienversichert){
            familie = "familienversichert";
        } else {
            familie = "nicht familienversichert";
        }

        // Mit dem Aufruf "super.toString()" wird innerhalb der neuen Version der Methode
        // "toString" die Version aus "Patient" ausgeführt. So müssen gleichbleibende
        // Inhalte nicht erneut geschrieben werden.
        return super.toString() + ", " +  familie;
    }

    public boolean getFamilienversichert(){
        return familienversichert;
    }

    public void printAbrechnung(){
        System.out.println("Abrechnung: " + "XYZ");
    }

    // Überladung einer Methode: Die Methode "getName" existiert bereits in der
    // Elternklasse "Patient". Dort allerdings mit einem anderen Methodenkopf. In der
    // Elternklasse nimmt die Methode keine Parameter an.
    // Hier schon. Somit wird die Methode hier nicht überschrieben, sondern
    // überladen. Ein Objekt von "Kassenpatient" kann dadurch beide Varianten
    // aufrufen und verwenden. Siehe PatientTest.
    public String getName(String prefix){
```

```
67          return prefix + " " + super.getName();
68      }
69  }
```

```java
// Durch das "final" wird festgelegt, dass die Klasse "Privatpatient" nicht weiter
// vererbt werden kann.
// Eine Klassendefinition wie
// public class VIPPatient extends Privatpatient {...}
// wäre also nicht möglich.
// Ebenso können auch Methoden vom überschreiben geschützt werden, indem sie in der
// Elternklasse mit einem "final" versehen werden, können sie in Subklassen nicht
// überschrieben werden.
// Zur Info: "final" sorgt bei Variablen dafür, dass die Werte zu Konstanten werden.
public final class Privatpatient extends Patient {
    /**
     * Konstruktor für Privatpatient.
     * Muss "throws PraxisVollException" deklarieren, weil der Aufruf von super()
     * im Elternkonstruktor eine solche checked exception werfen kann und diese hier
     * weitergegeben wird. Siehe Patient
     */
    public Privatpatient(String name, String vorname, int alter) throws
PraxisVollException{
        // super: Siehe Kassenpatient
        super(name, vorname, alter);
    }

    public void printAbrechnung(){
        System.out.println("Abrechnung: " + "ABC");
    }
}

```

COMBINED - ZweiBettZimmer.java

```java
// Generische Klassen ermöglichen es allgemeine Klassendefinitionen zu erstellen. Dazu
// wird innerhalb der Klasse kein konkreter Datentyp verwendet, sondern ein Platzhalter.
// In diesem Fall ist der Platzhalter "T". Bei der Instanziierung wird der gewünschte
// Datentyp dann als Parameter in </>-Klammern übergeben. In diesem Fall soll ein
// ZweiBettZimmer als Klasse definiert werden. In ein Zimmer dürfen nur Patient des
// gleichen Typs. Außerdem wird durch das "extends Patient" festgelegt,
// dass der übergebene Datentyp "Patient" oder eine Kindklasse davon sein muss.
public class  ZweiBettZimmer<T extends Patient> {
    private T fensterSeite, wandSeite;

    public ZweiBettZimmer(T fensterSeite, T wandSeite) {
        this.fensterSeite = fensterSeite;
        this.wandSeite = wandSeite;
    }

    public T getFensterSeite() {
        return fensterSeite;
    }

    public T getWandSeite() {
        return wandSeite;
    }

    public void printBelegung(){
        System.out.println("Zimmerbelegung:");
        System.out.println("Fensterseite: " + fensterSeite.toString());
        System.out.println("Wandseite: " + wandSeite.toString());
    }
}

```

```java
// Diese Klasse ist eine benutzerdefinierte Ausnahme (Exception).
// Sie wird verwendet, um einen bestimmten Fehlerzustand in der Anwendung zu
// signalisieren:
// Dass die maximale Anzahl von Patienten in der Praxis erreicht wurde.
// Sie erbt von der Standard-Java-Klasse `Exception`, was bedeutet, dass es sich um eine
// überprüfte Ausnahme handelt (checked exception). Das bedeutet, dass Methoden, die
// diese Ausnahme werfen könnten, dies in ihrer Signatur deklarieren müssen
// (`throws PraxisVollException`)
// und aufrufende Methoden diese Ausnahme entweder abfangen (`catch`) oder ebenfalls
// weiterwerfen müssen.
public class PraxisVollException extends Exception{

    // Dies ist der Standardkonstruktor. Er wird verwendet, wenn die Ausnahme ohne
    // eine spezifische Fehlermeldung erstellt wird.
    public PraxisVollException(){}

    // Dies ist ein überladener Konstruktor, der eine formatierte Fehlermeldung
    // entgegennimmt. Er verwendet die Methode `String.format()` um die übergebenen
    // Argumente in die Nachricht einzufügen. Die formatierte Nachricht wird dann an den
    // Konstruktor der übergeordneten Klasse `Exception` weitergegeben, wo sie
    // gespeichert und über die Methode `getMessage()` abgerufen werden kann.
    public PraxisVollException( String format, Object... args ) {
        super( String.format( format, args ) );
    }
}
```

```java
// Diese Klasse ist eine benutzerdefinierte Ausnahme (Exception).
// Sie wird verwendet, um einen bestimmten Fehlerzustand in der Anwendung zu
// signalisieren:
// Dass ein ungültiger Name für einen Patienten eingegeben wurde.
// Sie erbt von der Standard-Java-Klasse `RuntimeException`, was bedeutet, dass es sich
// um eine nicht-überprüfte Ausnahme handelt (unchecked exception). Das bedeutet, dass
// Methoden, die diese Ausnahme werfen könnten, dies nicht explizit in ihrer Signatur
// deklarieren müssen (`throws UngueltigerNameException`). Aufrufende Methoden sind
// nicht gezwungen, diese Ausnahme abzufangen oder weiterzuwerfen, obwohl sie es tun
// können, wenn sie den Fehler behandeln möchten.
// RuntimeExceptions werden oft für Programmierfehler oder unerwartete Zustände
// verwendet, die zur Laufzeit auftreten können.
public class UngueltigerNameException extends RuntimeException{

    // Dies ist der Konstruktor, der eine Fehlermeldung entgegennimmt.
    // Die übergebene Nachricht wird an den Konstruktor der übergeordneten Klasse
    // `RuntimeException` weitergegeben, wo sie gespeichert und über die Methode
    // `getMessage()` abgerufen werden kann. Diese Nachricht beschreibt den spezifischen
    // Grund für den ungültigen Namen.
    public UngueltigerNameException(String message){
        super(message);
    }

}
```

File - J:\Repositories\IT-Techniker-Erlangen\24-25\PRG\Cheat-Sheet-SA1\src\Elektro.java

```java
public class Elektro extends Fahrzeug {
    // Definition privater Variablen
    private int engineCount;
    private int batterySize;

    // Benutzerdefinierter Konstruktor mit Parametern
    public Elektro(String make, String model, int horsepower, int engineCount, int batterySize) {
        // Parameterloser Vater-Konstruktor -> siehe Konstruktor in Verbrenner.java
        super();

        super.setMake(make);
        super.setModel(model);
        super.setHorsepower(horsepower);

        this.engineCount = engineCount;
        this.batterySize = batterySize;
    }

    // Get-Methoden -> siehe Fahrzeug.java
    public int getEngineCount() {
        return engineCount;
    }

    public int getBatterySize() {
        return batterySize;
    }

    // Set-Methoden -> siehe Fahrzeug.java
    public void setEngineCount(int engineCount) {
        this.engineCount = engineCount;
    }

    public void setBatterySize(int batterySize) {
        this.batterySize = batterySize;
    }

    // Überschreibung einer Methode der Vater-Klasse -> siehe Verbrenner.java
    public void printDataSheet(){
        super.printDataSheet();
        System.out.println("Motorenanzahl: " + engineCount);
        System.out.println("Batteriegröße: " + batterySize);
    }
}

```

File - J:\Repositories\IT-Techniker-Erlangen\24-25\PRG\Cheat-Sheet-SA1\src\Katalog.java

```java
// Import für das Anlegen des Scanners
import java.util.Scanner;

public class Katalog {
    public static void main(String[] args) {
        // Einlesen wie groß der Katalog sein soll
        // Scanner-Objekt anlegen -> Scanner vorher importieren!
        Scanner scanner = new Scanner(System.in);
        System.out.println("Wie gross soll der Katalog sein?");
        int catalogSize = scanner.nextInt(); // -> Als Beispiel wurde hier 5 eingegeben
        // Zum Einlesen eines Strings:
        // String inputText = scanner.nextLine();

        // Anlage eines Array, der die vorher definierte Größe hat
        Benziner [] benzinerKatalog = new Benziner[catalogSize];

        // Anlage der Benziner im Array
        benzinerKatalog[0] = new Benziner("BMW", "435i", 340, 3000, 6, "Super-Plus", 98);
        benzinerKatalog[1] = new Benziner("Audi", "S3", 310, 2000, 4, "Super", 95);
        benzinerKatalog[2] = new Benziner("Mercedes-Benz", "C43 AMG", 390, 3000, 6, "
Super-Plus", 98);
        benzinerKatalog[3] = new Benziner("Volkswagen", "Golf GTI", 245, 2000, 4, "Super"
, 95);
        benzinerKatalog[4] = new Benziner("Porsche", "911 Carrera", 385, 3000, 6, "Super-
Plus", 98);

        // Data-Sheet für alle Elemente im Array ausgeben
        for(int i = 0; i < benzinerKatalog.length; i++){
            benzinerKatalog[i].printDataSheet();
        }

        // String Funktionen
        System.out.println("------ String-Funktionen ------");

        String mercedes = benzinerKatalog[2].getMake();
        String lowerCaseMercedes = mercedes.toLowerCase();
        String volkswagen = benzinerKatalog[3].getMake();

        System.out.println(" 1: " + lowerCaseMercedes);
        // "mercedes-benz"
        System.out.println(" 2: " + mercedes.charAt(3));
        // "c" -> Der Character an der 4. Position
        System.out.println(" 3: " + mercedes.compareTo(volkswagen));
        // "-9" -> "V" olkswagen ist 9 Stellen unter "M" ercedes-Benz
        System.out.println(" 4: " + volkswagen.endsWith("wagen"));
        // "true"
        System.out.println(" 5: " + mercedes.equals(lowerCaseMercedes));
        // "false" -> da Case-Sensitive
        System.out.println(" 6: " + mercedes.equalsIgnoreCase(lowerCaseMercedes));
        // "true" -> nicht Case-Sensitive
        System.out.println(" 7: " + mercedes.indexOf("-"));
        // "8" -> Bindestrich an 8. Stelle
        System.out.println(" 8: " + mercedes.indexOf("Ben"));
        // "9" -> beginn des substrings an 9. Stelle
        System.out.println(" 9: " + mercedes.indexOf("BMW"));
        // "-1" -> Substring ist nicht in String enthalten
        System.out.println("10: " + mercedes.length());
        // "13" -> "Mercedes-Benz" hat 13 Zeichen
        System.out.println("11: " + mercedes.replace('e', 'x'));
        // "Mxrcxdxs-Bxnz" -> alle 'e' werden durch 'x' ersetzt
        System.out.println("12: " + mercedes.repeat(2));
        // "Mercedes-BenzMercedes-Benz" -> String wird zwei mal hintereinander gehängt
        System.out.println("13: " + mercedes.startsWith("Schoen"));
        // "false" -> String beginnt nicht mit "Schoen"
        System.out.println("14: " + mercedes.substring(4));
        // "edes-Benz" -> String beginnt an 4. Stelle
        System.out.println("15: " + mercedes.substring(5,12));
```

```java
        // "des-Ben" -> string beginnt an 5. Stelle und endet VOR 12.
        System.out.println("16: " + volkswagen.toLowerCase());
        // "volkswagen" -> alle Großbuchstaben werden in Kleinbuchstaben umgewandelt
        System.out.println("17: " + volkswagen.toUpperCase());
        // "VOLKSWAGEN" -> alle Kleinbuchstaben werden in Großbuchstaben umgewandelt
        System.out.println("18: " + String.valueOf(1.5e2));
        // "150.0" -> wird in lesbaren String umgewandelt


        // Datentypen in JAVA

        // Primitive Datentypen
        // -> Werden im Speicher direkt in der Zelle gespeichert
        boolean isCar = true;       // Wahr/Falsch-Wert
        char character = 'a';       // Ein einzelnes Zeichen
        int number = 0;             // Ganze Zahl
        float floatNumber = 1.5F;   // Kommazahl mit einfacher Genauigkeit
        double doubleNumber = 1.5D; // Kommazahl mit doppelter Genauigkeit

        byte smallNumber = 127;     // Kleine Ganzzahl (-128 bis 127)
        short mediumNumber = 32000; // Mittelgroße Ganzzahl (-32,768 bis 32,767)
        long largeNumber = 123456789L; // Große Ganzzahl (bis ±9 Quintillionen)

        // Non-Primitive/Komplexe Datentypen
        // -> Speicherzelle enthält einen Pointer auf ein Objekt im Heap sobald es erstellt wurde.
        // Vor der Initialisierung enthält die Speicherzelle einen Pointer auf null
        int [] numbers = new int[10]; // Array jeglichen Datentyps
        Fahrzeug testFahrzeug = new Fahrzeug(); // Jede Klasse ist ein komplexer Datentyp
        String text = "Das ist ein String"; // Ein String ist ein Objekt
    }
}

```

File - J:\Repositories\IT-Techniker-Erlangen\24-25\PRG\Cheat-Sheet-SA1\src\Benziner.java

```java
public class Benziner extends Verbrenner  {
    // Definition privater Variablen
    private String fuelSpecification;
    private int octaneRating;

    // Benutzerdefinierter Konstruktor mit Parametern -> siehe Verbrenner.java
    public Benziner(String make, String model, int horsepower, int displacement, int cylinderCount, String fuelSpecification, int octaneRating) {
        // Es wird nun der Konstruktor der Vater-Klasse "Verbrenner" aufgerufen. In dessen Konstruktor werden allerdings
        // auch Attribute der, aus aktueller Sicht, Großvaterklasse "Fahrzeug" gesetzt. Daher benötigt der Konstruktor
        // der Klasse "Benziner" als Parameter sowohl die Werte für die Attribute der Klasse "Verbrenner" als auch der Klasse "Fahrzeug"
        super(make, model, horsepower, displacement, cylinderCount);

        // Da ein Benzinmotor immer Benzin als Treibstoff braucht, kann das entsprechende Attribut der Vater-Klasse "Verbrenner"
        // im Konstruktor gesetzt werden
        super.setFuelType("Benzin");

        this.fuelSpecification = fuelSpecification;
        this.octaneRating = octaneRating;
    }

    // Get-Methoden -> siehe Fahrzeug.java
    public String getFuelSpecification() {
        return fuelSpecification;
    }

    public int getOctaneRating() {
        return octaneRating;
    }

    // Set-Methoden -> siehe Fahrzeug.java
    public void setFuelSpecification(String fuelSpecification) {
        this.fuelSpecification = fuelSpecification;
    }

    public void setOctaneRating(int octaneRating) {
        this.octaneRating = octaneRating;
    }

    // Erneutes Überschreiben einer Methode der Vater-Klasse: Die Methode "printDataSheet" wurde in der Klasse "Fahrzeug" definiert
    // und anschließend in "Verbrenner" überschrieben. Beim Überschreiben wurde allerdings die Methode der Vater-Klasse mit verwendet
    // um nicht unnötig Code zu duplizieren. Das Gleiche kann in der Tochter-Klasse "Benziner" der Tochter-Klasse "Verbrenner" gemacht
    // werden. Dabei wird nicht die Methode der Klasse "Fahrzeug" überschrieben, sondern die Methode der Klasse "Verbrenner". Beim
    // Überschreiben kann mittels "super" die überschriebene Methode aus "Verbrenner" verwendet werden.
    public void printDataSheet(){
        // Aufruf der Methode aus der Vater-Klasse "Verbrenner"
        super.printDataSheet();
        System.out.println("Kraftstoff Spezifikation: " + fuelSpecification);
        System.out.println("Oktanzahl : " + octaneRating);
    }
}

```

File - J:\Repositories\IT-Techniker-Erlangen\24-25\PRG\Cheat-Sheet-SA1\src\Fahrzeug.java

```java
public class Fahrzeug {
    // private Variablen erstellen
    private String make;
    private String model;
    private int horsepower;

    // An dieser Stelle könnte ein Konstruktor stehen. Der Konstruktor wird ausgeführt, sobald eine neue Instanz eines
    // Objekts erstellt wird. Wenn kein Konstruktor definiert ist, gibt es einen parameterlosen Konstruktor. Das Objekt
    // wird also initialisiert, aber es passiert nichts weiter.

    // Da private Variablen nicht per Punktreferenz (Objekt.attribut) werden Get-Methoden benötigt,
    // die public sind. Dadurch kann ein privater Parameter trotzdem per Punktreferenz abgerufen werden
    // (Objekt.getAttribut())
    public String getMake() {
        return make;
    }

    public String getModel() {
        return model;
    }

    public int getHorsepower() {
        return horsepower;
    }

    // Set-Methoden sind das Gegenstück von Get-Methoden. Damit lassen sich private Variablen außerhalb des
    // Objekts setzen
    public void setMake(String make) {
        this.make = make;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public void setHorsepower(int horsepower) {
        this.horsepower = horsepower;
    }

    // Weitere Methoden
    public void printDataSheet(){
        System.out.println("----- Fahrzeug-Info-Blatt -----");
        System.out.println("Marke: " + make);
        System.out.println("Modell: " + model);
        System.out.println("Leistung: " + horsepower);
    }
}

```

File - J:\Repositories\IT-Techniker-Erlangen\24-25\PRG\Cheat-Sheet-SA1\src\Verbrenner.java

```java
public class Verbrenner extends Fahrzeug{
    // Definition von privaten Variablen
    private int displacement;
    private int cylinderCount;
    private String fuelType;

    // Konstruktor der Verbrenner-Klasse
    public Verbrenner(String make, String model, int horsepower, int displacement, int cylinderCount) {
        // Zuerst muss IMMER der Konstruktor der Vater-Klasse ausgeführt werden! Ein parameterloser Konstruktor muss
        // nicht zwingend genannt werden. Er wird automatisch aufgerufen, sollte nichts anderes in der Tochter-Klasse
        // definiert sein,
        super();

        // Da der Konstruktor der Tochter-Klasse "Verbrenner" auch Parameter annimmt, die der Vater-Klasse zugehörig
        // sind, sollten diese auch entsprechend gesetzt werden. Da die Variablen der Vater-Klasse allerdings private
        // sind, können sie nicht direkt (super.make = make) gesetzt werden, sondern müssen über die Set-Methoden
        // gesetzt werden
        super.setMake(make);
        super.setModel(model);
        super.setHorsepower(horsepower);

        // Anschließend werden die privaten Variablen der Tochter-Klasse gesetzt
        this.displacement = displacement;
        this.cylinderCount = cylinderCount;
    }

    // Get-Methoden → siehe Fahrzeug.java
    public int getDisplacement() {
        return displacement;
    }

    public int getCylinderCount() {
        return cylinderCount;
    }

    public String getFuelType() {
        return fuelType;
    }

    public int getDisplacementPerCylinder() {
        return displacement / cylinderCount;
    }

    // Set-Methoden → siehe Fahrzeug.java
    public void setDisplacement(int displacement) {
        this.displacement = displacement;
    }

    public void setCylinderCount(int cylinderCount) {
        this.cylinderCount = cylinderCount;
    }

    public void setFuelType(String fuelType) {
        this.fuelType = fuelType;
    }

    // Überschreibung einer Methode der Vater-Klasse:
    // Wenn ein Vater-Objekt eine Methode besitzt, die in einer Tochter-Klasse anders behandelt werden muss,
    // kann diese überschrieben werden. Dazu muss eine Methode mit dem gleichen Methoden-Kopf erstellt werden.
```

```java

    public void printDataSheet(){
        // Innerhalb einer Methode kann die Methode des Vater-Objekts mit super.<Methode>() aufgerufen werden.
        // Das beschränkt sich nicht auf überschriebene Methoden. In diesem Fall macht das allerdings hier Sinn:
        // Die Methode erweitert das Basis-Data-Sheet aus dem Vater-Objekt um Informationen aus dem Tochter-Objekt.

        // Ohne den Super-Aufruf müsste der gesamte Inhalt der gleichnamigen Methode in Fahrzeug.java eingefügt werden.
        // Man spart sich also Code-Zeilen
        super.printDataSheet();
        System.out.println("Hubraum: "+ displacement + " ccm");
        System.out.println("Zylinder: "+ cylinderCount + " (" + getDisplacementPerCylinder() + " ccm)");
        System.out.println("Kraftstoff: "+ fuelType);
    }
}

```