1 Java Einführung

1.1 Die main-Methode

Der Einstiegspunkt eines Programms in Java ist immer die `main()`-Methode:

```
public class MeinProgramm {
    public static void main(String[] args) {
        System.out.println("Hallo Welt!");
        // Mein Code hier...
    }
}
```

Diese Methode muss in einer Klasse als öffentliche, statische Methode ohne Rückgabewert definiert sein, benötigt einen String-Array als Parameter, und trägt immer den Namen `main`. In einem Projekt sollte immer nur eine `main`-Methode existieren, jedoch kann man in IntelliJ einstellen, welche Klasse als Einstiegspunkt gewählt werden soll.

Java unterstützt verschiedene Datentypen, wie `int`, `double`, und `String`. Zahlenwerte können mit C-Operatoren (`+`, `-`, `*`, `/`, `>`, `==`, usw.) verrechnet und verglichen werden. Strings lassen sich mithilfe von `+` aneinanderreihen. So kann man zum Beispiel zwei Strings zu einem neuen String kombinieren:

```
String halloString = "Hallo";
String halloWeltString = halloString + " Welt!";
```

Wird ein String mit einem Zahlenwert verbunden (z.B. `"Ergebnis: " + zahl`), wird dieser automatisch zu einem String umgewandelt. Eine Ausgabe in die Konsole erfolgt dann mit der Methode `System.out.println(str)`. Alternativ kann auch ein Formatierungsstring verwendet werden, falls mehr Kontrolle über die Zahlendarstellung benötigt wird (siehe https://docs.oracle.com/javase/tutorial/essential/io/formatting.html).

Achtung! Während die Datentypen `double` und `int` primitive Datentypen sind, ist ein String ein Objekt! Strings sollten deshalb immer mit der `equals()`-Methode verglichen werden anstatt mit `==`.

Aufgaben

1. Erstellen Sie ein Projekt in IntelliJ mit dem Namen “HelloWorldProjekt”.
2. Erstellen Sie eine Klasse **`Hauptklasse`** mit einer Main-Methode und geben Sie “Hello World!” in die Konsole aus.
3. Legen Sie in der `main`-Methode eine Variable vom Typ `double` welche den Radius eines Kreises beschreibt. Berechnen Sie dann den Kreisumfang und die Fläche des Kreises und speichern diese in zwei neuen Variablen ab. *Hinweis: Die Kreiszahl Pi erhalten Sie durch die statische Variable* ***`Math.PI`****.*

4. Geben Sie das Ergebnis in die Konsole aus in der Form: “Bei einem Radius von ... beträgt der Umfang ... und die Fläche ...”

Lösung:

```
public class Hauptklasse {

    public static void main(String[] args) {
        System.out.println("Hallo Welt!");

        double radius = 2;
        double umfang = 2 * radius * Math.PI;
        double flaeche = radius * radius * Math.PI;

        System.out.println(
                "Bei einem Radius von " + radius +
                " beträgt der Umfang " + umfang +
                " und die Fläche " + flaeche);
    }
}
```

1.2 Klassen und Objekte

Alle Datentypen in Java, mit Ausnahme der primitiven Datentypen, sind *Klassen*. Klassen sind eine Ansammlung von Attributen und Methoden, welche die Klasse beschreiben. Attribute werden auf Klassenebene wie eine Variable definiert.

Konkrete Instanzen einer Klasse werden *Objekte* genannt. Verschiedene Objekte einer Klasse haben alle die selben Attribute und Methoden, jedoch können die Attribute unter den Objekten unterschiedliche Werte beinhalten:

```java
// MeineKlasse.java
public class MeineKlasse {
    // Wir definieren ein Attribut vom Typ String
    public String meinText;
}
// Hauptklasse.java
public class Hauptklasse {
    public static void main(String[] args) {
        MeineKlasse meinObjekt1 = new MeineKlasse();
        meinObjekt1.meinText = "foo";

        MeineKlasse meinObjekt2 = new MeineKlasse();
        meinObjekt2.meinText = "bar";

        System.out.println("Objekt 1: " + meinObjekt1.meinText); //
        ↪ Objekt 1: foo
        System.out.println("Objekt 2: " + meinObjekt2.meinText); //
        ↪ Objekt 2: bar
    }
}
```

Tipp

Ein Objekt besitzt immer das Attribut `this`, welches das Objekt selber referenziert. Dies kann nützlich sein wenn Attribute von lokalen Variablen oder Parametern überdeckt werden.

1.3 Funktionen und Methoden

Im Gegensatz zu C++ können Funktionen nur auf einem Klassenlevel definiert werden. Funktionen die an eine Klasse gebunden sind (in Java also alle Funktionen), werden auch Methoden genannt. Ähnlich wie in C++ haben Methoden einen Satz an Eingabeparametern, und einen optionalen Rückgabewert (Methoden ohne Rückgabewert haben den Rückgabetypen `void`). Eine Funktion kann wie folgt definiert werden:

```java
public class MeineKlasse {
    public int meineFunktion(double doubleParameter, String
    ↪ stringParameter) {
        // Mein code hier...

        return meinIntWert;
    }
}
```

Dabei wird zwischen Objektmethoden und Klassenmethoden unterschieden. Eine Objektmethode kann auf die Attribute einer Klasse zugreifen und sie verändern. Diese kann aber nur durch ein erstelltes Objekt aufgerufen werden:

```
MeineKlasse meinObjekt = new MeineKlasse();
meinObjekt.meineFunktion();
```

Klassenmethoden werden mit dem Schlüsselwort `static` beschrieben, und haben keinen Zugriff auf die Attribute und Objektmethoden einer Klasse. Sie können auch ohne das erstellen eines Objektes aufgerufen werden:

```
MeineKlasse.meineStatischeFunktion();
```

1.4 Konstruktor

Wenn ein neues Objekt mit `new` initialisiert wird, wird der Konstruktor der Klasse aufgerufen. Eine Klasse kann dabei auch mehrere Konstruktoren mit unterschiedlicher Signatur haben. Hier definieren wir zwei verschiedene Konstruktoren.

```
public class Person {
    String lieblingsfarbe;

    public Person() {
        lieblingsfarbe = "gelb";
    }

    public Person(String lieblingsfarbe) {
        this.lieblingsfarbe = lieblingsfarbe;
    }
}
```

Wird kein Konstruktor definiert, so wird ein Default-Konstruktor angenommen, welcher keine Parameter besitzt. Der Konstruktor sollte in der Regel alle Attribute mit sinnvollen Werten belegen und benötigte Objekte instanziieren. Häufig werden dabei Attribute direkt über Konstruktor-Parameter gesetzt.

1.5 Zugriffsregeln

Java hat ein strenges Zugriffsprinzip, welche bestimmt, welche Klassen, Attribute und Methoden von außen zugänglich sind. Die wichtigsten Zugriffsregeln sind wie folgt:

- `public`: Ein Attribut oder eine Methode die als `public` definiert ist, kann von allen Orten aus aufgerufen, gelesen und geschrieben werden.
- `protected`: Ein Attribut oder eine Methode sind nur von Unterklassen aus zugänglich. Nützlich, wenn eine Klasse erweiterbar gemacht werden soll.
- `private`: Nur Zugriff von innerhalb der aktuellen Klasse, kein Zugriff von außen.

- Keine Zugriffsregel: Ähnlich wie `public`, jedoch nur von Klassen im selben Paket zugänglich.

Generell werden Methoden als `public` deklariert, damit sie von außen aus aufrufbar sind. Nur interne Methoden sollten als `private` definiert werden.

Attribute hingegen werden generell als `private` deklariert. Zugriff auf Attribute sollten hinter sogenannten *Getter- und Setter-Methoden* versteckt werden:

```
public class Person {
    private int alter;

    public void setAlter(int alter) {
        this.alter = alter;
    }

    public int getAlter() {
        return alter;
    }
}
```

Zugriffsregeln helfen bei der Kontrolle über ein Attribut. So kann ein Attribut schreibgeschützt werden indem man die Setter-Methode weglässt oder `protected` deklariert, oder ein Schreibzugriff auf eine Attribut kann geloggt werden.

1.6 Ein erstes Objekt

Es soll eine Java-Klasse erstellt werden, mit der verschiedene Quader modelliert werden. Ein Quader wird durch seine geometrischen Abmessungen (d.h. Länge, Breite und Höhe) beschrieben. Zusätzlich sollen der Rauminhalt und die Oberfläche des Quaders bei Bedarf berechnet werden können.

Wir haben festgelegt, dass jeder Quader durch eine Länge, eine Breite und eine Höhe beschrieben werden kann. Dies sind die Eigenschaften oder Attribute des Quaders. Mehr als diese drei Attribute benötigen wir vorerst nicht. Wir wählen für unsere drei Attribute die Namen laenge, breite und hoehe. Diese Benennung ist gegenüber einer eher kryptischen Kurzbezeichnung wie `l`, `b` und `h` zu bevorzugen, da der Name eines Attributs selbsterklärend sein sollte.

Zusätzlich zu den Attributen müssen wir die Methoden festlegen, die unser Quader bereitstellen soll. Wir wollen zunächst nur eine Methode vorsehen, mit der das Volumen des Quaders berechnet werden soll. Der Name der Methode soll berechneVolumen lauten. Das Ergebnis, das die Methode zurückgibt, soll das Volumen, also eine Kommazahl (double) sein.

Legen Sie ein neues Projekt in IntelliJ an und erstellen Sie in diesem Projekt eine neue Java-Klasse namens Quader.

Um die Klasse testen zu können, benötigen wir noch eine main-Methode. Diese schreiben

wir diesmal in eine neue Klasse **QuaderTest**, die nur zum Testen der Funktionsfähigkeit der Klasse **Quader** gedacht ist.

```
public class QuaderTest {
    public static void main(String[] args) {
        Quader q1 = new Quader();
        q1.laenge = 10;
        q1.breite = 5;
        q1.hoehe = 4;
        System.out.println("Volumen: " + q1.berechneVolumen() +
        ↪  "m³");
    }
}
```

Aufgaben

1. Erstellen Sie die Klasse **Quader** wie im oberen Text beschrieben und testen diese mithilfe der Klasse **QuaderTest**.
2. Erweitern Sie ihre Klasse durch einen Konstruktor für Quader, welche die Dimensionen als Parameter übernimmt. Ändern Sie QuaderTest entsprechend ab.
3. Erzeugen Sie einen zweiten Quader als Würfel mit der Kantenlänge 5 und lassen Sie sich sein Volumen ausgeben.
4. Stellen Sie sicher, dass Ihre Klasse Quader gegen Zugriff von außen geschützt ist. Sehen Sie für alle Attribute sowohl Setter- als auch Getter-Methoden vor.
5. Erzeugen Sie einen dritten Quader, dessen Abmessungen als arithmetisches Mittel aus den Abmessungen der beiden anderen Quader berechnet werden.
6. Erweitern Sie die Quader-Klasse um eine Methode zur Berechnung der Oberfläche des Körpers. Lassen Sie sich die Oberfläche von allen Quadern in der Testklasse berechnen und ausgeben.
7. Erweitern Sie die Quader-Klasse um ein Attribut zur Erfassung der Dichte. Schreiben Sie außerdem eine Methode zur Berechnung des Gewichts eines Quader (verwenden Sie dabei nach Möglichkeit die schon vorhandene Berechnungsmethode für das Volumen). Weisen Sie jedem Quader eine Dichte von 0,5 g/cm³ zu und lassen Sie die Gewichte aller Quader in der Konsole ausgeben.
8. Erweitern Sie Ihre Klasse um einen zweiten Konstruktor, der zusätzlich zur Länge, Breite und Höhe auch die Dichte als Parameter verwendet.
9. Schreiben Sie einen dritten Konstruktor, der nur eine einzelne Länge übergeben bekommt und damit einen Würfel mit der Dichte 1.0 erzeugt.
10. Zeichnen Sie das UML-Klassendiagramm der Klasse Quader mit allen soeben durchgeführten Änderungen.

Lösung:

Quader.java

```
public class Quader {
    private static final double G_CM3_TO_KG_M3 = 100;

    private double laenge, hoehe, breite;
    private double dichte;

    public Quader(double laenge, double hoehe, double breite) {
        this.laenge = laenge;
        this.hoehe = hoehe;
        this.breite = breite;
    }

    public Quader(double laenge, double hoehe, double breite, double
    ↪   dichte) {
        this.laenge = laenge;
        this.hoehe = hoehe;
        this.breite = breite;
        this.dichte = dichte;
    }

    public Quader(double seitenlaenge) {
        this.laenge = seitenlaenge;
        this.hoehe = seitenlaenge;
        this.breite = seitenlaenge;
        this.dichte = 1.0;
    }

    public double berechneVolumen() {
        return laenge * breite * hoehe;
    }

    public double berechneOberfläche() {
        return 2 * laenge * breite
             + 2 * laenge * hoehe
             + 2 * breite * hoehe;
    }

    public double berechneGewicht() {
        return berechneVolumen() * dichte * G_CM3_TO_KG_M3;
    }
```

```
    public double getLaenge() {
        return laenge;
    }

    public void setLaenge(double laenge) {
        this.laenge = laenge;
    }

    public double getHoehe() {
        return hoehe;
    }

    public void setHoehe(double hoehe) {
        this.hoehe = hoehe;
    }

    public double getBreite() {
        return breite;
    }

    public void setBreite(double breite) {
        this.breite = breite;
    }

    public double getDichte() {
        return dichte;
    }

    public void setDichte(double dichte) {
        this.dichte = dichte;
    }
}
```

QuaderTest.java

```
public class QuaderTest {
    public static void main(String[] args) {
        Quader q1 = new Quader(10,5,4);
        System.out.println("Volumen: " + q1.berechneVolumen() + "m³");

        Quader q2 = new Quader(5,5,5);
        System.out.println("Volumen: " + q2.berechneVolumen() + "m³");

        Quader q3 = new Quader(
                (q1.getLaenge() + q2.getLaenge()) / 2,
```

```
                (q1.getHoehe() + q2.getHoehe()) / 2,
                (q1.getBreite() + q2.getBreite()) / 2
            );
        System.out.println("Volumen: " + q3.berechneVolumen() + "m³");

        q1.setDichte(0.5);
        q2.setDichte(0.5);
        q3.setDichte(0.5);

        System.out.println("Gewicht q1: " + q1.berechneGewicht() +
        ↪  "kg");
        System.out.println("Gewicht q2: " + q2.berechneGewicht() +
        ↪  "kg");
        System.out.println("Gewicht q3: " + q3.berechneGewicht() +
        ↪  "kg");
    }
}
```

1.7 Arrays und ArrayList

Arrays werden in Java mit eckigen Klammern denotiert. Ein Array kann wie folgt definiert werden:

```
// Erstellt ein Array mit 10 integer-Werten.
int[] meinIntArray = new int[10];
meinIntArray[0] = 42;
meinIntArray[1] = 256;
// ...
```

Ein Array kann von jedem Typ sein, aber hat immer eine feste Größe welche nicht geändert werden kann. Arrays aus primitiven Datentypen werden immer mit 0 in allen Werten initialisiert. Arrays aus Objekten beinhalten aber an jeder Stelle nur `null`. Der Programmierer muss sich selbst um die Initialisierung der einzelnen Array-Werte kümmern.

Die feste Größe des Arrays zeigt einige Limitationen auf. Betrachten wir das folgende Beispiel:

```
public class ArbeitenMitArrays {
    public static void main(String[] args) {
        // Anlegen und Fuellen von x
        String[] x = new String[4];
        x[0] = "a";
        x[1] = "b";
        x[2] = "c";
        // Aendern und Loeschen eines Eintrags
        x[2] = "d";
        x[1] = null;
        // Alle Eintraege ausgeben
        for (int i = 0; i < x.length; i++)
            System.out.println(x[i]);
        // einen Eintrag suchen
        String suchtext = "d";
        int nr = -1;
        for (int i = 0; i < x.length; i++)
            if (suchtext.equals(x[i]))
                nr = i;
        System.out.println("Position von " + suchtext + ": " +nr);
    }
}
```

Wir legen ein eindimensionales String-Array an und füllen es mit drei Einträgen. Wenn wir einen Eintrag ändern wollen, füllen wir das zugehörige Array-Feld mit einem neuen Wert (hier: d). Zum Löschen eines Array-Eintrags muss an die zu löschende Position der Wert null geschrieben werden. Zum Ausgeben des Arrays müssen wir alle Array-Elemente in einer for-Schleife durchlaufen; die Länge des Arrays können wir über das Attribut length erfragen. Wollen wir wissen, an welcher Position im Array ein bestimmter Eintrag steht,

müssen wir es ebenfalls mit einer for-Schleife durchlaufen. An diesem Beispiel sehen wir bereits einige der Probleme, die beim Arbeiten mit Arrays auftreten können: Die Array-Größe muss immer korrekt gewählt werden. Ist sie wie im Beispiel zu groß, bleiben Teile des Arrays leer, ist sie zu klein, gibt es eine `ArrayIndexOutOfBoundsException`. Das Attribut length gibt nur den zur Verfügung stehenden Platz, nicht aber die tatsächlich momentan vorhandene Menge von Objekten im Array an. Beim Löschen bleiben leere Stellen im Array zurück. Schöner wäre es meist, wenn die entstehenden Löcher durch automatisch nachrückende Array-Elemente geschlossen würden. Das Suchen eines bestimmten Array-Eintrags muss selbst programmiert werden. Schauen wir uns nun an, wie uns die Hilfsklasse ArrayList dabei helfen kann, einige dieser Probleme zu umgehen:

```
import java.util.ArrayList;
public class ArbeitenMitArrayList {
    public static void main(String[] args) {
        // Anlegen und Fuellen von x
        ArrayList<String> x = new ArrayList<String>();
        x.add("a");
        x.add("b");
        x.add("c");
        // Aendern und Loeschen eines Eintrags
        x.set(2, "d");
        x.remove(1);
        // Alle Eintraege ausgeben
        for (int i = 0; i < x.size(); i++)
            System.out.println(x.get(i));
        // einen Eintrag suchen
        String suchtext = "d";
        System.out.println("Position von " + suchtext + ": "+
        ↪  x.indexOf(suchtext));
    }
}
```

Der grundlegende Ablauf beider Programme ist identisch. Bei genauer Betrachtung beider Quelltexte fallen uns jedoch die folgenden Unterschiede auf: Statt eines String-Arrays wird nun eine ArrayList angelegt, in der ebenfalls String-Objekte gespeichert werden. Die spitzen Klammern geben den Typ der Objekte an, die in der Liste gespeichert werden sollen. Beim Erzeugen einer ArrayList braucht keine Größe der Liste angegeben zu werden! Das Hinzufügen von Objekten erfolgt über die add-Methode. Alle neuen Objekte werden an das Ende der Liste angehängt. Über die Methode set lässt sich ein Listeneintrag ändern. Über remove können einzelne Objekte gelöscht werden. Beim Löschen bleibt keine Lücke zurück - die nachfolgenden Elemente rücken automatisch nach. Statt des Array-Attributs length besitzt die ArrayList eine Methode size, die nur die aktuell tatsächlich vorhandene Anzahl der Elemente zurückgibt. Das Suchen von Elementen erfolgt über die Methode indexOf automatisch.

Aufgaben

1. Erstellen Sie eine ArrayList und fügen Sie 5 Quader hinzu. Geben Sie anschließend die Anzahl der vorhanden Elemente innerhalb der erstellten ArrayList in der Konsole aus.
2. Berechnen Sie nun das Gesamtgewicht aller Quader der ArrayList und lassen Sie sich dieses von der Konsole ausgeben.
3. Entfernen Sie zuletzt alle Quader aus der ArrayList, die mehr als 5000kg wiegen. Geben Sie anschließend die Anzahl der restlichen Quader, die sich in der ArrayList befinden, in der Konsole aus.

Lösung:

QuaderTest.java

```
import java.util.ArrayList;

public class QuaderTestArray {

    public static void main(String[] args) {
        //Aufgabe 1
        ArrayList<Quader> quaderList = new ArrayList<Quader>();

        for (int i = 1; i < 6; i++) {
            quaderList.add(new Quader(i));
        }

        System.out.println(quaderList.size());

        //Aufgabe 2
        double gesamtGewicht = 0.0;
        for (int i = 0; i < quaderList.size(); i++) {
            gesamtGewicht = gesamtGewicht +
            ↪  quaderList.get(i).berechneGewicht();
        }

        System.out.println(gesamtGewicht);

        //Aufgabe 3
        for(int i = quaderList.size() - 1; i >= 0; i--) {
            // Liste von hinten durchlaufen, damit entfernte Objekte
            ↪  die Reihenfolge nicht durcheinander bringen
            Quader q = quaderList.get(i);
            if(q.berechneGewicht() >= 5000.0)
                quaderList.remove(i);
```

```
        }

        // Alternative mit Lambda-Ausdrücken
        quaderList.removeIf(quader -> quader.berechneGewicht() >=
        ↪  5000.0);

        System.out.println(quaderList.size());

    }

}
```

1.8 Assoziationen und Multiplizitäten

Eine Assoziation ist grundsätzlich eine Verbindung zwischen zwei Java-Klassen. Nehmen wir dazu als Beispiel die beiden Klassen Student und Computer. Zwischen beiden Klassen herrscht offensichtlich eine Verbindung: Studenten sitzen vor Computern und Computer werden von den Studenten benutzt. Student und Computer sind also miteinander assoziiert.

Um die Assoziation zwischen beiden Klassen umzusetzen, erstellen wir zunächst ein neues IntelliJ-Projekt. Anschließend erzeugen wir zwei neue Java-Klassen namens Student und Computer, die erstmal nur den folgenden kurzen Inhalt haben sollen:

```
public class Student {
    public String name;
    public long matrikel;

    public Student(String n, long m) {
        name = n;
        matrikel = m;
    }
}
```

```
public class Computer {
    public String typ;
    public Computer(String t) {
        typ = t;
    }
}
```

Um für die weiteren Beispiele den Zugriff auf die Attribute zu vereinfachen, wurden alle Attribute als public deklariert. Bei sauberer Programmierung sollten eigentlich alle Attribute nach dem Geheimnisprinzip als private gekennzeichnet und über set- und get-Methoden zugreifbar gemacht werden.

Wie immer schreiben wir zusätzlich eine Test-Klasse, in der wir unser Programm während der Entwicklung kontinuierlich testen können:

```
public class AssoziationsTest {
    public static void main(String[] args) {
        Student s1 = new Student("Peter Petersen", 108089288888L);
        Computer c1 = new Computer("Dell OptiPlex 755 MT");
    }
}
```

Bisher sind die beiden Klassen Student und Computer erstmal vollkommen unabhängig voneinander – es gibt keine Verbindung zwischen ihnen. Wir möchten aber in unserem Programm abbilden, dass jeder Student einen Computer benutzt. Schauen wir uns diese Assoziation zunächst im UML-Klassendiagramm an:

Abbildung 1:

Eine Assoziation wird im Klassendiagramm als einfache Linie dargestellt. Die Zahl 1 am rechten Ende legt fest, dass jeder Student immer an genau einem Rechner arbeiten soll. Der Text benutzt erläutert die Art der Assoziation etwas genauer. Das schwarze Dreieck gibt an, in welche Richtung die Erläuterung zu lesen ist. Insgesamt lässt sich diese Assoziation lesen als: jeder Student benutzt genau einen Computer. Wie wird nun diese Assoziation in Java umgesetzt? Im Prinzip ist jede Assoziation nichts weiter als ein normales Attribut vom Datentyp der Klasse, mit der unsere Klasse assoziiert ist. Wir fügen also in unsere Klasse Student ein neues Attribut ein:

```
s1.computer = c1;
System.out.println(s1.name + " benutzt einen " + s1.computer.typ);
```

Damit ist der Student s1 mit dem Computer c1 verknüpft! Diese Verknüpfung ist unidirektional, da nur der Student den Computer kennt, der Computer den Studenten aber nicht. Wenn beide Objekte sich hingegen gegenseitig kennen, spricht man von einer bidirektionalen Assoziation.

Aufgaben

1. Erweitern Sie die im obigen Klassendiagramm dargestellte Assoziation, sodass auch der Computer den Studenten kennt. Gehen Sie dabei zunächst davon aus, dass an jedem Computer gleichzeitig maximal ein Student arbeiten kann.

2. Setzen Sie die Assoziation in Java um. Fügen Sie zur Computer-Klasse ein Student-Attribut namens benutzer ein. Weisen Sie dem Computer in der main-Methode den Studenten als Benutzer zu und testen Sie die Assoziation mit einer Textausgabe.

3. Schreiben Sie eine neue Klasse namens CIPInsel, die über einen Namen verfügt (z.B. IC 04/630). Erweitern Sie die Klasse Computer um eine Assoziation zur zugehörigen CIP-Insel. Testen Sie die Assoziation in der Testklasse und lassen Sie sich den Namen des Raums des Rechners des Studenten s1 ausgeben.

Lösung:

Student.java

```
public class Student {
    public String name;
    public long matrikel;

    public Computer computer;

    public Student(String n, long m) {
        name = n;
        matrikel = m;
    }
}
```

Computer.java

```
public class Computer {
    public String typ;

    // Aufgabe 2
    public Student student;

    // Aufgabe 3
    public CIPInsel cipInsel;

    public Computer(String t) {
        typ = t;
    }
}
```

CIPInsel.java

```
public class CIPInsel {
    public String name;
    public Computer[] computer = new Computer[10];
    public CIPInsel(String n) {
        name = n;
    }
}
```

AssoziationsTest.java

```java
public class AssoziationsTest {
    public static void main(String[] args) {
        Student s1 = new Student("Peter Petersen", 108089288888L);
        Computer c1 = new Computer("Dell OptiPlex 755 MT");

        s1.computer = c1;
        System.out.println(s1.name + " benutzt einen " +
        ↪  s1.computer.typ);

        // Aufgabe 2
        c1.student = s1;
        System.out.println(c1.typ + " wird benutzt von " +
        ↪  c1.student.name);

        // Aufgabe 3
        CIPInsel insel1 = new CIPInsel("IC 04/630");
        c1.cipInsel = insel1;


        insel1.computer[0] = c1;

        for(int i = 1; i < 10; i++) {
            insel1.computer[i] = new Computer("Dell OptiPlex 755 MT");
            insel1.computer[i].cipInsel = insel1;
        }
    }
}
```

Die bisher betrachteten Assoziationen hatten die Eigenschaft, dass jeweils immer ein Objekt mit genau einem anderen Objekt verbunden ist, d.h. im UML-Klassendiagramm steht am jeweiligen Ende der Assoziationslinie jeweils eine 1. Diese Zahl bezeichnet man als Multiplizität. Eine Multiplizität von 1 ist jedoch für viele Anwendungen nicht ausreichend. Betrachten wir hierzu erneut die oben begonnene Assoziation zwischen Computer und CIPInsel:

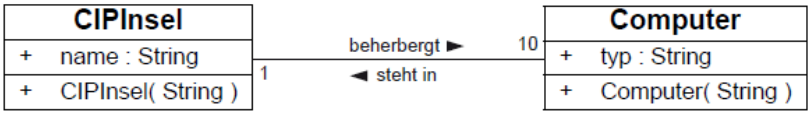

Abbildung 2:

In jeder CIP-Insel stehen nun genau 10 Computer, d.h. jedes CIPInsel-Objekt ist mit 10 Objekten der Klasse Computer assoziiert. Wie man in einem Attribut mehrere Objekte ablegen kann, haben wir bereits gelernt: Durch Arrays!

```
public class CIPInsel {
    public String name;
    public Computer[] computer = new Computer[10];
    public CIPInsel(String n) {
        name = n;
    }
}
```

Zum Testen ändern wir die main-Methode so ab, dass die CIPInsel direkt mit 10 Rechnern gefüllt wird und der Student den ersten Computer benutzt:

```
public class AssoziationsTest {
    public static void main(String[] args) {
        Student s1 = new Student("Peter Petersen", 108089288888L);
        CIPInsel c1 = new CIPInsel("IC 04/630");
        for(int i = 0; i < 10; i++) {
            c1.computer[i] = new Computer("Dell OptiPlex 755 MT");
            c1.computer[i].cip = c1;
        }
        s1.computer = c1.computer[0];
        c1.computer[0].benutzer = s1;
    }
}
```

Nun können wir Assoziationen mit einer festen Multiplizität umsetzen (z.B. in der CIPInsel stehen 10 Rechner). Wenn wir einen 11. Rechner in den Raum stellen wollten, müssten wir das Klassendiagramm und den Quelltext entsprechend anpassen. Was aber ist, wenn wir die Anzahl der Objekte gar nicht kennen oder wir sie nicht begrenzen wollen? Nehmen wir an, wir wollen eine weitere Klasse namens Vorlesung erstellen, und wollen alle Studenten, die an dieser Vorlesung teilnehmen, erfassen. Da wir die Zahl der Studenten vor Semesterbeginn nicht genau kennen, benötigen wir eine Assoziation mit "beliebig großer" Multiplizität.

Unsere Klasse Vorlesung soll Methoden zum Hinzufügen, Entfernen und Suchen von Studenten besitzen. Die maximale Anzahl an Studenten ist nicht begrenzt, was durch das *-Symbol angegeben wird. Für die Umsetzung solcher Assoziation benötigen wir wieder die in der letzten Übung vorgestellte ArrayList, da diese ihre Größe an die Erfordernisse anpassen kann (anders als das Array, dessen Größe nicht mehr geändert werden kann).

Abbildung 3: Codevervollständigung in IntelliJ

```
import java.util.*;
public class Vorlesung {
    public String bezeichnung;
    public String semester;
    public ArrayList<Student> studenten =
        new ArrayList<Student>();
    public Vorlesung(String b, String s) {
        bezeichnung = b;
        semester = s;
    }
    public void studentHinzufuegen(Student s) {
        studenten.add(s);
    }
    public void studentEntfernen(Student s) {
        studenten.remove(s);
    }
    public Student sucheStudent(long matrikel) {
        for(int i = 0; i < studenten.size(); i++) {
            Student s = studenten.get(i);
            if(s.matrikel == matrikel) {
                return s;
            }
        }
        return null;
    }
}
```

Aufgaben

1. Erweitern Sie die Klasse Student so, dass jeder Student mit beliebig vielen Vorlesungen assoziiert ist. Sehen Sie Methoden zum Hinzufügen und Entfernen von Vorlesungen vor.

Lösung:

Student.java

```
package vorlesungen;

import java.util.ArrayList;

public class Student {

    String name;
    long matrikel;
    ArrayList<Vorlesung> vorlesungen;

    public Student(String name, long matrikel) {
        this.name = name;
        this.matrikel = matrikel;
        vorlesungen = new ArrayList<Vorlesung>();
    }

    public void vorlesungHinzufuegen(Vorlesung v) {
        vorlesungen.add(v);

        // Der Student muss auch in die Vorlesung eingetragen werden,
        ↪  damit die gegenseitige Beziehung hält
        v.studentHinzufuegen(this);
    }

    public void vorlesungEntfernen(Vorlesung v) {
        vorlesungen.remove(v);
        v.studentEntfernen(this);
    }

}
```