Programmierung 1

PD Dr. Kaspar Riesen

Zusammenfassung & Musterlösungen der Serien

HS 2023

Lukas Batschelet

16-499-733

Sämtliches Material ist auch auf GitHub abgelegt: https://github.com/lbatschelet/23HS_P1

Inhaltsverzeichnis

Eigene Zusammenfasung

1.1 Grundlagen

1.1.1 Grundkonzepte der Java-Programmierung

- **Programmieren**: Problemlösung mit Software.
- **Programmiersprache**: Definiert mit Wörtern und Regeln Programmieranweisungen.
- **Java**: Weit verbreitet, vielseitig, plattformunabhängig, objektorientiert.
- **Klassen**: Grundbausteine von Java-Programmen; enthalten Methoden und Variablen.
- **main Methode**: Startpunkt jedes Java-Programms.
- **Kommentare**: Erläutern den Code (`//`, `/* */`, `/** */`).

1.1.2 Notationskonventionen

- **Variablennamen**: Beginnen mit Kleinbuchstaben, CamelCase für zusammengesetzte Namen. Beispiel: `meinAlter`.
- **Konstanten**: Großbuchstaben und Unterstriche. Beispiel: `MAX_WERT`.
- **Methodennamen**: Beginnen mit Kleinbuchstaben, CamelCase für zusammengesetzte Namen, oft Verben. Beispiel: `berechneAlter()`.
- **Klassennamen**: Beginnen mit Großbuchstaben, CamelCase für zusammengesetzte Namen. Beispiel: `Person`.

1.1.3 Variablendeklaration und -zuweisung

- *Variable* := Speicherort für einen Wert oder ein Objekt.
- Variablen müssen mit *Datentyp* und *Bezeichner* deklariert werden.
- Mit dem *Zuweisungsoperator* werden deklarierten Variablen Werte zugewiesen: `int i = 17;`.
- **Deklaration**: Definiert Typ und Namen der Variable, z.B. `int seiten;`.
- **Zuweisung**: Weist der Variablen einen Wert zu, z.B. `seiten = 256;`.
- **Kombinierte Deklaration und Zuweisung**: `int seiten = 256;`.
- **Mehrere Variablen**: Gleichzeitige Deklaration, z.B. `int figures = 46, tables; tables = 17;`.
- Lesen verändert Variablen niemals: `MAX\_POINTS * 5`
- *Zuweisungsoperatoren* und das *Inkrement/Dekrement* machen das Leben einfacher:

```
points = points * 2;
points *= 2;

points = points + 1;
points++;
```

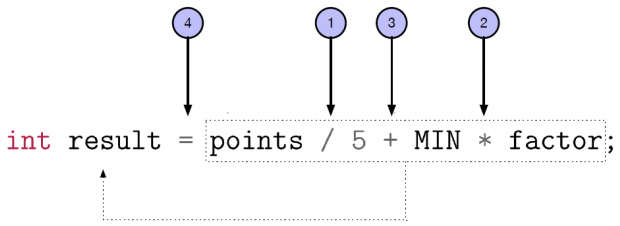


Abbildung 1.1: Reihenfolge der Auswertung: Der Zuweisungsoperator = hat die niedrigste Priorität.

1.1.4 Primitive Datentypen

- `byte`: 8-Bit Ganzzahl, Bereich -128 bis 127.
- `short`: 16-Bit Ganzzahl, Bereich -32,768 bis 32,767.
- `int`: 32-Bit Ganzzahl, Bereich -2^{31} bis $2^{31}-1$.
- `long`: 64-Bit Ganzzahl, Bereich -2^{63} bis $2^{63}-1$.
- `float`: 32-Bit IEEE 754 Fließkommazahl.
- `double`: 64-Bit IEEE 754 Fließkommazahl.
- `boolean`: Wahrheitswert, `true` oder `false`.
- `char`: 16-Bit Unicode-Zeichen.

1.1.5 Casting in Java

- **Implizites Casting**: Automatische Konvertierung von kleineren zu größeren Datentypen, z.B. `int` zu `double`.
- **Explizites Casting**: Man. Konv. von gross zu klein, z.B. `double num = 12.34; int count = (int) num;`.

1.1.6 Aliase und Abhängigkeiten

- **Primitive Datentypen**: Kopien von Variablen sind unabhängig.

```
int num1 = 17;
    int num2 = num1;
    num2 = 99;
    System.out.println(num1); // 17
    System.out.println(num2); // 99
```

- **Objektvariablen**: Kopien von Variablen sind abhängig (Aliase).

```
Integer num1 = new Integer(17);
    Integer num2 = num1;
    num2.setValue(99);
    System.out.println(num1); // 99
    System.out.println(num2); // 99
```

1.1.7 Arithmetische Operatoren und Reihenfolge

1.1.8 Division

- **Ganzzahldivision** (`int`):
  - Das Ergebnis ist eine Ganzzahl, Bruchteile werden abgeschnitten.
  - Beispiel: `int ergebnis = 5 / 2;` ergibt 2.
- **Fließkommadivision** (`double`):

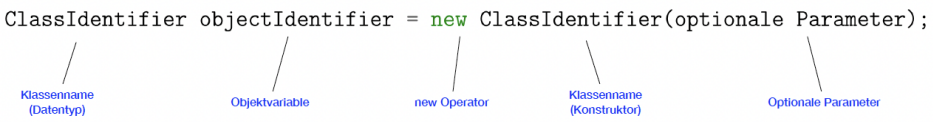


Abbildung 1.2: Deklaration und Instanziierung eines Objektes.

  - Das Ergebnis enthält Nachkommastellen.
  - Beispiel: `double ergebnis = 5.0 / 2.0;` ergibt `2.5`.
- **Casting bei Division**:
  - Bei Casten einer Ganzzahldivision zu `double` bleibt der Bruchteil abgeschnitten.
  - Beispiel: `double ergebnis = (double)(5 / 2);` ergibt `2.0`.

1.2 Java-Klassen

1.2.1 Aufbau einer Java-Klasse

- **Klassendefinition**: Beginnt mit dem Schlüsselwort `class`, gefolgt vom Klassennamen.
- **Attribute**: Variablen innerhalb einer Klasse, repräsentieren den Zustand.
- **Methoden**: Funktionen innerhalb einer Klasse, definieren Verhalten.
- **Konstruktor**: Spezielle Methode zum Erstellen von Objekten.

```
public class Auto {
        // Attribute
        private String marke;
        private int baujahr;

        // Konstruktor
        public Auto(String marke, int baujahr) {
            this.marke = marke;
            this.baujahr = baujahr;
        }

        // Methode
        public void anzeige() {
            System.out.println(marke + ", Baujahr: " + baujahr);
        }
    }
```

1.2.2 Methodenkopf

- Methodenkopf: (1) Sichtbarkeit (2) Datentyp der Rückgabe oder void (3) Bezeichner (4) Formale Parameter in Klammern
- **Sichtbarkeitssmodifizierer**: Bestimmt die Sichtbarkeit (z.B. `public`, `private`).
- **Rückgabetyp**: Datentyp des Rückgabewerts der Methode.
- **Methodenname**: Eindeutiger Bezeichner der Methode.
- **Formale Parameterliste**: Variablen zur übergabe von Werten an die Methode.
- Variablen sollten `private` deklariert werden.
- Methoden können `private` oder `public` deklariert werden (je nach Zweck)

```
public class Integer {

    private int value;

    public Integer(int value) {
    this.value = value;
    }
    public String toString() {
    return this.value + "";
    }
    public void setValue(int value) {
    this.value = value;
    }
}
```

1.2.3 Konstruktoren

- **Definition**: Spezielle Methode zum Erstellen und Initialisieren eines Objekts.
- **Konstruktortypen**: Standardkonstruktor (ohne Parameter) und parametrisierte Konstruktoren.

```
public class Auto {
    private String marke;
    private int baujahr;

    // Standardkonstruktor
    public Auto() {
    }

    // Parametrisierter Konstruktor
    public Auto(String marke, int baujahr) {
        this.marke = marke;
        this.baujahr = baujahr;
    }
}
```

1.2.4 Parameter und variadische Methoden

- **Parameter**: Variablen, die beim Aufruf einer Methode Werte übergeben.
- **Variadische Parameter (Varargs)**: Erlauben eine variable Anzahl von Argumenten.

```
public class Rechner {
    // Variadische Methode
    public int summe(int... zahlen) {
        int summe = 0;
        for (int zahl : zahlen) {
            summe += zahl;
        }
        return summe;
    }
}
```

1.2.5 Generische Klassen

- Wir können Klassen generisch machen:

```
public class Rocket<T> {

    private T cargo;

    public Rocket(T cargo) {
        this.cargo = cargo;
    }
    public void set(T cargo) {
        this.cargo = cargo;
    }
    public T get() {
        return this.cargo;
    }
}
```

- Um eine generische Klasse zu instanziieren, müssen wir sie zusammen mit einem *Typargument* instanziieren:

```
Rocket<Integer> intRocket = new Rocket<Integer>();
    Rocket<String> stringRocket = new Rocket<String>();
```

- Die Typvariable `T` wird nun überall mit dem Typargument ersetzt.

1.2.6 `enum`

- Ein `enum` zählt *alle* zulässigen Werte eines Typs auf

```
public enum Category {
        Mathematik, Geographie
    }
```

```
public enum Category {
        Mathematik(1), Geographie(2);
        private int id;
        private Category(int id) {
            this.id = id;
        }
        public int getId() {
            return this.id;
        }
    }
```

- Jedes `enum` besitzt Methoden (wie z.B. `getId()`)
- Jede `enum`-Klasse besitzt statische Methoden (wie z.B. `values()`)

```
Category[] categories = Category.values();
    for (Category category : categories) {
        System.out.println(category);
    }
```

1.2.7 Statische Variablen und Methoden

- Statische Variablen werden von allen Instanzen geteilt (es existiert also nur eine Kopie der Variablen für alle Objekte)

```java
public class Person {
        public static int globalCount = 0;
        private int id;
        public Person() {
            this.id = Person.globalCount++;
        }
    }
```

- Statische Methoden werden direkt aufgerufen, ohne vorher ein Objekt zu instanziieren

```java
public class Math {
        public static int max(int a, int b) {
            return (a > b) ? a : b;
        }
    }
```

```java
int max = Math.max(3, 7);
```

1.2.8 Klassen des java-API

Wrapper-Klassen und Methoden

Methode	Beschreibung	Bsp. Eingabe	Bsp. Rückgabe
Integer			
`parseInt(String s)`	Konvertiert String zu int	`"123"`	`123`
`valueOf(int i)`	Gibt Integer-Objekt für int-Wert	`123`	`Integer 123`
`compare(int x, int y)`	Vergleicht zwei int-Werte	`compare(3, 7)`	`-1`
`MIN_VALUE`	Gibt den kleinsten int-Wert	-	`-2147483648`
`MAX_VALUE`	Gibt den größten int-Wert	-	`2147483647`
Double			
`parseDouble(String s)`	Konvertiert String zu double	`"123.45"`	`123.45`
`valueOf(double d)`	Gibt Double-Objekt für double-Wert	`123.45`	`Double 123.45`
`compare(double d1, double d2)`	Vergleicht zwei double-Werte	`compare(3.5, 7.5)`	`-1`
`POSITIVE_INFINITY`	Gibt den positiven unendlichen double-Wert	-	`Infinity`
`NEGATIVE_INFINITY`	Gibt den negativen unendlichen double-Wert	-	`-Infinity`
Boolean			
`parseBoolean(String s)`	Konvertiert String zu boolean	`"true"`	`true`
`valueOf(boolean b)`	Gibt Boolean-Objekt für boolean-Wert	`true`	`Boolean true`
Character			
`isLetter(char c)`	Prüft, ob Zeichen ein Buchstabe ist	`'a'`	`true`
`isDigit(char c)`	Prüft, ob Zeichen eine Ziffer ist	`'1'`	`true`
`toUpperCase(char c)`	Wandelt Zeichen in Großbuchstaben	`'a'`	`'A'`
`toLowerCase(char c)`	Wandelt Zeichen in Kleinbuchstaben	`'A'`	`'a'`

weitere Klassen

Klasse & Methode	Beschreibung	Bsp. Eingabe	Bsp. Rückgabe
`String.length()`	Gibt die Länge des Strings zurück	`"Hello"`	`5`
`String.charAt(int index)`	Gibt Zeichen an Index zurück	`"Hello", 1`	`'e'`
`String.substring(int a, int b)`	Gibt Teilstring zurück	`"Hello", 1, 3`	`"el"`
`String.indexOf(String str)`	Gibt Index des Teilstrings oder -1	`"Hello", "ll"`	`2`
`String.toLowerCase()`	Konvertiert String zu Kleinbuchstaben	`"Hello"`	`"hello"`
`String.toUpperCase()`	Konvertiert String zu Großbuchstaben	`"hello"`	`"HELLO"`
`String.contains(CharSequence s)`	Prüft, ob String Teilstring enthält	`"Hello", "ll"`	`true`
`String.equals(Object anObject)`	Vergleicht zwei Strings	`"Hello", "hello"`	`false`

Klasse & Methode	Beschreibung	Bsp. Eingabe	Bsp. Rückgabe
`Math.sqrt(double a)`	Quadratwurzel von a	`4`	`2.0`
`Math.pow(double a, double b)`	a hoch b	`2, 3`	`8.0`
`Math.abs(int a)`	Absolutwert von a	`-5`	`5`
`Math.random()`	Zufällige Zahl zwischen 0.0 und 1.0	-	`0.45`
`Random.nextInt()`	Zufällige Ganzzahl	-	`42`
`Random.nextInt(int bound)`	Zufällige Ganzzahl bis bound (exkl.)	`10`	`5`
`Random.nextBoolean()`	Zufälliger Wahrheitswert	-	`true`
`Random.nextDouble()`	Zufällige Fließkommazahl	-	`0.62`
`System.currentTimeMillis()`	Aktuelle Zeit in Millisekunden seit 1. Januar 1970	-	`1609459200000`
`Scanner(System.in)`	Scanner für Eingaben	-	-
`Scanner.next()`	Liest das nächste Token	-	`"Hello"`
`Scanner.nextLine()`	Liest die nächste Zeile	-	`"Hello World"`
`Scanner.nextInt()`	Liest die nächste Ganzzahl	-	`42`
`DecimalFormat(String pattern)`	Konstruktor mit Muster	`"#0.00"`	-
`format(double number)`	Formatieren einer Zahl	`1234.5678`	`"1234.57"`

- **#** - Stellt eine Ziffer dar; Null wird nicht dargestellt, wenn sie nicht notwendig ist.
- **0** - Stellt eine Ziffer dar; führt zu Nullen, wenn keine Ziffer vorhanden ist.
- **.** - Dezimaltrennzeichen.
- **,** - Gruppierungstrennzeichen.
- **%** - Multipliziert die Zahl mit 100 und zeigt sie als Prozentsatz an.
- **E0** - Trennt die Mantisse und Exponenten in wissenschaftlicher Notation.
- **;** - Trennt Formate; das erste für positive Zahlen und das zweite für negative Zahlen.

1.2.9 `ArrayList<T>`

- Die Klasse `ArrayList<T>` erlaubt es, generische Sammlungen von Objekten des Typs `T` anzulegen.
- Objekte dieser Klasse werden bei der Instanziierung parametrisiert:

```
ArrayList<String> names = new ArrayList<String>();
ArrayList<PlayerCard> cards = new ArrayList<PlayerCard>();
ArrayList<Integer> numbers = new ArrayList<Integer>();
```

- Listen passen Ihre Grösse dynamisch an:

```
names.add("Keanu");
names.add("Kevin");
System.out.println(names); // [Keanu, Kevin]
names.add("Karl");
System.out.println(names); // [Keanu, Kevin, Karl]
names.remove(1);
System.out.println(names); // [Keanu, Karl]
```

1.2.10 `PrintWriter`

- Die Klasse `PrintWriter` erlaubt Ausgaben in Dateien (wirft möglicherweise eine Exception)

```
public static void main(String[] args) throws IOException {
    String fileName = "output.txt";
    PrintWriter outFile = new PrintWriter(fileName);
    outFile.print("Hallo Welt!");
    outFile.close();
}
```

1.2.11 Abhängigkeit von sich selbst

- Eine Klasse kann von sich selbst abhängig sein

```
public class Person {

    private String name;
    private ArrayList<Person> friends;

    public Person(String name) {
        this.name = name;
        this.friends = new ArrayList<Person>();
    }
    public void knows(Person other) {
        this.friends.add(other);
    }
    public String toString() {
        return this.name;
    }
    public ArrayList<Person> getFriends() {
        return this.friends;
    }
}
```

```
Person p1 = new Person("Emilie");
Person p2 = new Person("Ava");
Person p3 = new Person("Maya");
p1.knows(p2);
p1.knows(p3);

System.out.println(p1.getFriends()); // [Ava, Maya]
```

1.2.12 Aggregation

- Aggregation := Ein Objekt besteht z.T. aus anderen Objekten

```
public class Person {

    private String name;
    private Adress address;

    public Person(String name, Adress address) {
        this.name = name;
        this.address = address;
    }
}
```

```java
public class Adress {

    private String street;
    private int zipCode;

    public Adress(String street, int zipCode) {
        this.street = street;
        this.zipCode = zipCode;
    }
}
```

1.2.13 Getter und Setter

- **Getter**: Methode, die den Wert eines Attributs zurückgibt.
- **Setter**: Methode, die den Wert eines Attributs setzt.

```java
public class Auto {
    private String marke;

    // Getter
    public String getMarke() {
        return marke;
    }

    // Setter
    public void setMarke(String marke) {
        this.marke = marke;
    }
}
```

1.3 Schleifen und Bedingungen

1.3.1 `if-else` Anweisung

- **Verwendung**: Zur Kontrolle des Programmflusses basierend auf Bedingungen.
- **Struktur**: Besteht aus einer Bedingung und einem Codeblock, der ausgeführt wird, wenn die Bedingung wahr (‘true‘) ist.

```java
if (bedingung) {
    // Code, der ausgeführt wird, wenn Bedingung wahr ist
} else {
    // Code, der ausgeführt wird, wenn Bedingung falsch ist
}
```

1.3.2 `switch`-Anweisung

- **Verwendung**: Vereinfacht mehrfache ‘if-else‘-Anweisungen, basierend auf dem Wert einer Variablen.
- **Struktur**: Besteht aus einem Ausdruck und mehreren ‘case‘-Labels, die unterschiedliche Fälle repräsentieren.

```
switch (variable) {
    case wert1:
        // Code für wert1
        break;
    case wert2:
        // Code für wert2
        break;
    default:
        // Code, wenn kein anderer Fall zutrifft
}
```

1.3.3 Conditional Operator

- **Verwendung**: Kürzere Form für einfache ‘if-else‘-Anweisungen.
- **Struktur**: Drei Teile - eine Bedingung, ein Ergebnis für ‘true‘ und ein Ergebnis für ‘false‘.

```
int ergebnis = (bedingung) ? wertWennTrue : wertWennFalse;
```

1.3.4 `do-while`-Schleife

- **Verwendung**: Schleife, die den Codeblock mindestens einmal ausführt und danach prüft, ob die Bedingung wahr ist.
- **Struktur**: Die Bedingung wird am Ende jeder Schleifeniteration überprüft.

```
do {
        // Code, der mindestens einmal ausgeführt wird
    } while (bedingung);
```

1.3.5 `for`-Schleife

- **Verwendung**: Schleife mit definierter Anzahl von Iterationen.
- **Struktur**: Besteht aus Initialisierung, Bedingung und Inkrementierung.
- **For-Schleife**: Klassische Schleife mit definierter Anzahl von Iterationen.
- **For-Each-Schleife**: Vereinfachte Form zum Durchlaufen von Arrays oder Sammlungen.
- Der *Schleifenkopf* der `for`-Schleife besteht aus drei Teilen:
  - *Initialisierung*: Wird am Anfang und genau einmal durchgeführt.
  - *Boolesche Bedingung*: Wird immer vor dem nächsten Eintritt in die Schleife überprüft.
  - *Inkrement*: Wird immer am Ende der Schleife durchgeführt.

```
// Klassische For-Schleife
    for (int i = 0; i < 10; i++) {
        // Code, der 10 Mal ausgeführt wird
    }

    // For-Each-Schleife
    int[] zahlen = {1, 2, 3, 4, 5};
    for (int zahl : zahlen) {
        // Code, der für jede Zahl im ArrayList-Element ausgeführt wird
    }
```

1.3.6 Vergleiche

- Vorsich bei == auf Dezimalzahlen

```
final double TOLERANCE = 0.00000001;
    if (Math.abs(num1 - num2) < TOLERANCE) {
        // ...
    }
```

- Vergleich von Zeichen basiert auf Unicode (Ziffern < Grossbuchstaben < Kleinbuchstaben)

```
char c0 = '0', c1 = 'A', c2 = 'a';
    System.out.println(c0 < c1); // true
    System.out.println(c1 < c2); // true
```

- Vorsicht bei == auf Objekten: Testet auf Aliase
- Verwenden/Schreiben der Methode `equals` und der Methode `compareTo`

```
public class Integer {
        private int value;
        public Integer(int value) {
            this.value = value;
        }
        public boolean equals(Integer other) {
            return this.value == other.value;
        }
        public int compareTo(Integer other) {
            return this.value - other.value;
        }
    }
```

```
Integer i1 = new Integer(2);
    Integer i2 = new Integer(17);
    Integer i3 = new Integer(2);
    System.out.println(i1.equals(i2)); // false
    System.out.println(i1.equals(i3)); // true

    System.out.println(i1.compareTo(i2)); // -15
    System.out.println(i1.compareTo(i3)); // 0
    System.out.println(i2.compareTo(i3)); // 15
```

Wächterwerte

- Mit *Wächterwerten* können wir ein Programm kontrollieren:

```
Scanner scan = new Scanner(System.in);
    int input = 1;
    while (input != 0) {
        System.out.print("Mit 0 Beenden Sie den Prozess. ");
        input = scan.nextInt();
    }
    System.out.println("--ENDE--");
```

- `while`-Schleifen können auch zur Kontrolle von Eingaben verwendet werden:

```
Scanner scan = new Scanner(System.in);
    System.out.print("Alter eingeben: ");
    int age = scan.nextInt();
    while (age < 0) {
        System.out.println("Ungültiger Wert.");
        System.out.print("Alter eingeben: ");
        age = scan.nextInt();
    }
```

1.3.7 `do`-Anweisung

- Die `do`-Anweisung ist ähnlich zur `while`-Anweisung, evaluiert aber die Boolesche Bedingung am Ende der Schleife.

```
System.out.print("Erreichte Punkte (0 bis 100): ");
    int points = scan.nextInt();
    while (points < 0 || points > 100) {
        System.out.print("Erreichte Punkte (0 bis 100): ");
        points = scan.nextInt();
    }
```

```
int points;
    do {
        System.out.print("Erreichte Punkte (0 bis 100): ");
        points = scan.nextInt();
    } while (points < 0 || points > 100);
```

1.3.8 Vergleich von Daten

- **Primitive Datentypen**: Verwendung von Vergleichsoperatoren wie ==, !=, <, >.
- **Objekte**: Implementierung der `compareTo()` Methode aus dem `Comparable` Interface.

```
// Vergleich von primitiven Datentypen
    int x = 5;
    int y = 10;
    boolean sindGleich = x == y; // false

    // Implementierung von compareTo
    public class Person implements Comparable<Person> {
        private int alter;

        @Override
        public int compareTo(Person anderePerson) {
            return Integer.compare(this.alter, anderePerson.alter);
        }
    }
```

1.4 Arrays

- Arrays ermöglichen das Deklarieren einer einzigen Variablen eines Typs, die dann mehrere Werte dieses Typs speichern kann
- Arrays haben eine feste, unveränderliche Grösse (Konstante `length`), die bei der Instanziierung angegeben werden muss

```
int num1, num2, num3, num4, num5, num6;

    int[] nums = new int[6];

    int l = nums.length;
```

- Auf einzelne Elemente eines Arrays greift man mit einem Index innerhalb eckiger Klammern zu

```
for (int i = 0; i < nums.length; i++) {
        System.out.println(nums[i]);
```

1.4.1 Instanziierung

- Mit Initialisierungslisten können Arrays instanziiert und mit Werten gefüllt werden

```
int[] nums = {1, 2, 3, 4};
    String[] names = {"Goodbye", "Hello", "Hi", "Howdy"};
```

- Auf Methoden der in einem Array gespeicherten Objekte kann man über Array-Referenzen zugreifen

```
for (int i = 0; i < names.length; i++)
        names[i] = names[i].toUpperCase();
```

- Der Parameter der Methode `main` ist ein `String[]`. Dies sind Programmparameter, die beim Start des Programmes von Äussen"mitgegeben werden können

```
public static void main(String[] args) {
        String language = args[0];
        String version = args[1];
        String author = args[2];
    }
```

1.4.2 Variable Parameterlisten

- Methoden können mit variablen Parameterlisten umgehen

```
public static int min(int first, int ... others) {
        int min = first;
        for (int num : others)
            min = Math.min(min, num);
        return min;
    }
```

```
public class Greetings {
        private String primaryGreeting;
        private String[] greetings;
        public Greetings(String primaryGreeting, String ... otherGreetings) {
            this.primaryGreeting = primaryGreeting;
            this.greetings = otherGreetings;
        }
    }
```

1.4.3 Mehrdimensionale Arrays

- Zweidimensionale Arrays sind Arrays aus Arrays

```
int[][] table = new int[100][5];
    String[][] names = {{"Anne", "Barbara", "Cathrine"}, {"Danny", "Emilie", "Fanny"}};
```

- Für Referenzen auf Elemente in zweidimensionalen Arrays werden zwei Indizes benötigt

```
System.out.println(names[0][2]);
    for (int row = 0; row < names.length; row++)
        for (int col = 0; col < names[row].length; col++)
            names[row][col] = "Hannes";
```

1.5 Schnittstellen und Vererbung

1.5.1 Schnittstellen

- Schnittstellen enthalten abstrakte Methoden und/oder Konstanten

```
public interface Eatable {
        void eat();
    }
```

- Sie verleihen unterschiedlichen Dingen eine gemeinsame Sichtweise, ein gemeinsames Verhalten

```
public class BrusselsSprouts implements Eatable {
```

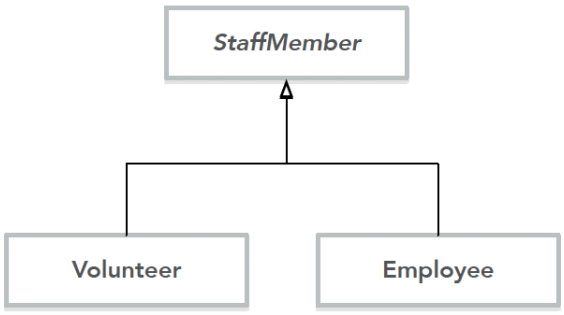
```
public class Potato implements Eatable {
```

```
public class Chocolate implements Eatable {
```

- Polymorphes Verhalten via Schnittstellen

```
Eatable[] storage = new Eatable[3];
    storage[0] = new Chocolate();
    storage[1] = new BrusselsSprouts();
    storage[2] = new Potato();
    for (Eatable eatable : storage)
        eatable.eat();
```

- Schnittstellen erlauben auch das Verbergen von Implementationsdetails und den Austausch von Klassen




```java
public interface Bag {
    void add(Item item);
}
```

```java
public class BackPack implements Bag {
    private ArrayList<Item> items;

    public void add(Item item) {
        this.items.add(item);
    }
}
```

```java
Bag bag = new BackPack();
bag.add(new Item("Socken"));
bag.add(new Item("Hosen"));
bag.add(new Item("Shirt"));
```

```java
public class SuitCase implements Bag {
    private int numberOfItems;
    private Item[] items;

    public SuitCase() {
        this.numberOfItems = 0;
        this.items = new Item[100];
    }
    public void add(Item item) {
        this.items[numberOfItems] = item;
        this.numberOfItems++;
    }
}
```

```java
Bag bag = new SuitCase();
bag.add(new Item("Socken"));
bag.add(new Item("Hosen"));
bag.add(new Item("Shirt"));
```

1.5.2 Vererbung

- Vererbung ist eines der Kernkonzepte der objektorientierten Programmierung
- In Java ist nur Einfachvererbung erlaubt
- Vererbung ist eine Einbahnstrasse
- Geerbte Methoden/Variablen werden weitervererbt
- Abstrakte Klassen dienen als Platzhalter in Hierarchien
- Im Konstruktor der Subklasse wird immer als erstes der Konstruktor der Superklasse aufgerufen
- Geerbte Methoden können überschrieben werden
- Eine konkrete Subklasse einer abstrakten Klasse muss alle abstrakten Methoden implementieren
- Mit der Referenz `super` kann auf die Superklasse zugegriffen werden

```java
public abstract class StaffMember {

    protected String name;

    public StaffMember(String name) {
        this.name = name;
    }
    public abstract double pay();

    public String toString() {
        return this.name;
    }
}
```

```java
public class Volunteer extends StaffMember {
    public Volunteer(String name) {
        super(name);
    }
    public double pay() {
        return 0;
    }
}
```

```java
public class Employee extends StaffMember {

    protected String socialSecurityNumber;
    protected double payRate;

    public Employee(String name,
            String socialSecurityNumber,
            double payRate) {
        super(name);
        this.socialSecurityNumber = socialSecurityNumber;
        this.payRate = payRate;
    }
    public double pay() {
        return this.payRate;
    }
    public String toString() {
        String s = super.toString();
        return s + " " + this.socialSecurityNumber;
    }
}
```

1.5.3 Polymorphismus

- Die Variable `member` ist polymorph

```java
StaffMember member;
member = new Employee("Daniel", "123-456", 8300.00);
member = new Volunteer("Tobias");
```

- Polymorphes Verhalten via Vererbung

```java
ArrayList<StaffMember> staff = new ArrayList<StaffMember>();
staff.add(new Employee("Daniel", "123-456", 8300.00));
staff.add(new Volunteer("Tobias"));
staff.add(new Volunteer("Susanne"));
staff.add(new Employee("Madeleine", "133-456", 15999.00));

for (StaffMember s : staff)
    s.pay(); // Polymorphes Verhalten
```

1.6 Algorithmen und Methoden

1.6.1 Überladen von Methoden

- Methoden können überladen werden
- Methoden mit gleichem Namen, aber unterschiedlichen Parametern
- Signatur (:= Bezeichner + formale Parameter) muss eindeutig sein
- Rückgabetyp kann nicht überladen werden

```java
private void doThis(int val) {
}
private void doThis(int val1, int val2) {
}
private void doThis(double val) {
}
```

1.6.2 Algorithmen

- Sortieren und Suchen sind zwei klassische Problemfelder der Informatik
- Es existieren zahlreiche Lösungen/Algorithmen für beide Problemfelder

for $index = 0, \ldots, (\textit{list.length} - 2)$ **do**
  $min = index$
  for $scan = index + 1, \ldots, (\textit{list.length} - 1)$ **do**
    if $\textit{list}[scan] < \textit{list}[min]$ **then**
      $min = scan$
    end
  end
  Tauche list[index] mit list[min]
end

Algorithm 1: Selection-Sort(*list*)

1.6.3 Generische Typisierung

- Methoden können auch generisch programmiert werden:

```
public static <T> void printArray(T[] array) {
    for (T element : array)
        System.out.println(element);
}
```

- Die Typvariable `T` wird beim Aufruf der Methode definiert:

```
Contact[] friends = new Contact[8];
...
Sorting.insertionSort(friends);
```

1.6.4 Herrsche und Teile

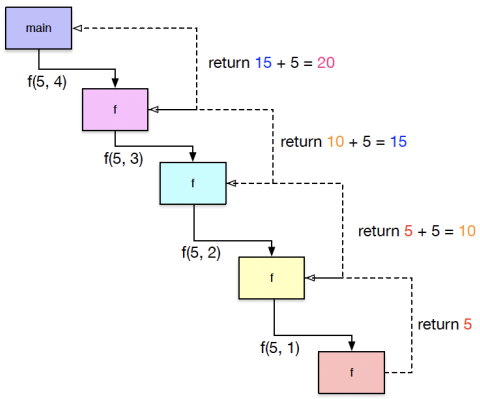
- Jede Funktion, die ein Programm erfüllen soll, muss in einer Methode programmiert sein
- Komplexe Funktionalitäten sollten Sie in mehrere Teile zerlegen
  - Verständlicher
  - Wiederverwendbar
  - leichter zu testen
  - schneller erstellt

```
public static <T> void bubbleSort(Comparable<T>[] list) {
    for (int i = 0; i <= list.length - 2; i++)
        for (int j = list.length - 1; j >= i + 1; j--)
            if (list[j].compareTo((T) list[j - 1]) < 0)
                swap(list, j, j -1);
}

private static <T> void swap(Comparable<T>[] list, int j, int i) {
    Comparable<T> temp = list[j];
    list[j] = list[i];
    list[i] = temp;
}
```

1.6.5 Rekursion

- Rekursion: Etwas ist durch sich selbst definiert
  - `Liste := Zahl`
  - `Liste := Zahl, Liste`
- Jede rekursive Definition benötigt einen nichtrekursiven Teil, den Basisfall, so dass die Rekursion enden kann
- Rekursive Methode: Die Methode ruft sich selber direkt oder indirekt auf
- Jeder rekursive Aufruf einer Methode definiert eigene lokale Variablen und eigene formale Parameter

```java
public class Recursion {
    public static void main(String[] args) {
        System.out.println(f(5, 4));
    }

    public static double f(int n, int m) {
        if (m == 1)
            return n;
        else
            return f(n, m - 1) + n;
    }
}
```



- Mit Rekursion kann man einige Probleme elegant lösen

```java
public static <T> void quickSort(Comparable<T>[] list, int p, int r) {
    if (p < r) {
        int q = partition(list, p, r);
        quickSort(list, p, q - 1);
        quickSort(list, q + 1, r);
    }
}
```

- Rekursive Algorithmen sind aber manchmal nicht so intuitiv wie iterative Algorithmen

```java
public static double f_1(int n, int m) {
    if (m == 1)
        return n;
    else
        return f_1(n, m - 1) + n;
}

public static double f_2(int n, int m) {
    return n * m;
}
```

1.6.6 Testen

- Testen umfasst mindestens das Ausführen eines vollendeten Programms mit verschiedenen Eingaben
- Test Case := Eingaben und Benutzeraktionen, gekoppelt mit den erwarteten Ergebnissen
- Test Suite := Mehrere Test Cases
- Defekttest := Ziel ist es, Fehler zu finden
- Regressionstest: Ziel ist es, nach der Korrektur keine neuen Fehler einzubauen
- Black-Box Test: Beruht nur auf Eingaben und erwarteten Ausgaben

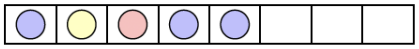
- White-Box Test: Konzentriert sich auf die interne Struktur des Codes

1.7 Collection Framework

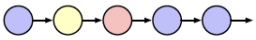
- Sammlung :=Behälter, um (meist gleichartige) Elemente zu organisieren.
- Idee einer Sammlung und die Implementierung sind zwei unterschiedliche Dinge

Idee: Liste

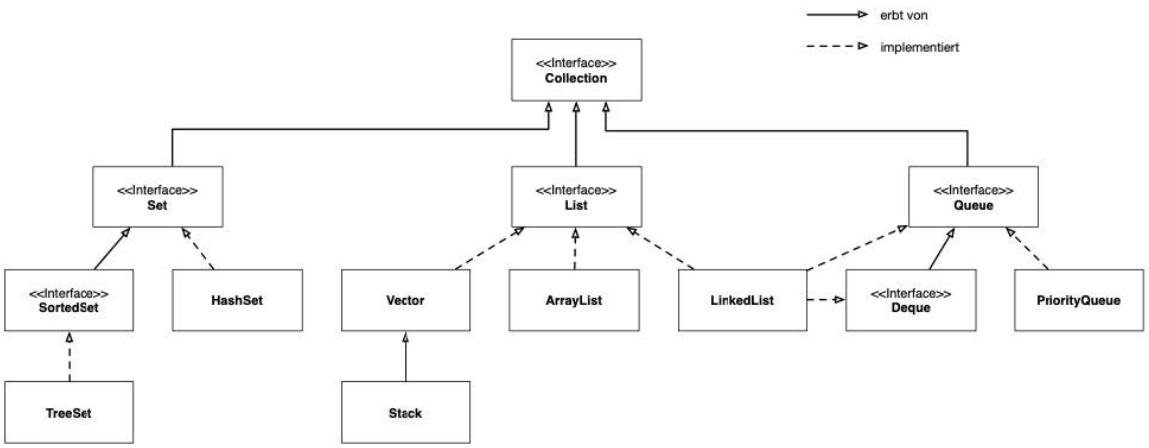
Implementierung mit Array



Implementierung durch Verkettung



- Wichtige abstrakte Datentypen und deren Implementierungen im Java API:
  - Listen (z.B. die Klasse `LinkedList`)
    - Verarbeitung über Index
  - Warteschlangen (z.B. die Schnittstelle `Queue`)
    - FIFO Verarbeitung
  - Stapel (z.B. die Klasse `Stack`)
    - LIFO Verarbeitung
  - Mengen (z.B. die Klasse `TreeSet`)
    - Keine Duplikate, keine Position
  - Assoziativspeicher (z.B. die Klasse `HashMap`)
    - Schlüssel-Wert Paare



- Das Collection Framework des Java APIs erlaubt die Trennung von abstrakten Datentypen (:= Schnittstellen) und Implementierungen

```
List<Door> list = new ArrayList<Door>();

List<Door> list = new LinkedList<Door>();
```

- Einige Methoden sind "weit obenïn der Hierarchie deklariert:
  - `add(T t)` oder `size()`
- Einige Methoden machen nur auf speziellen Datenstrukturen Sinn:
  - `add(int index, T t)` oder `peekFirst()`

1.8 Laufzeitfehler

- Laufzeitfehler sind Objekte der Klasse `Exception`, die eine unübliche Situation repräsentieren
- Fehlermeldungen beinhalten den Namen der `Exception`, möglicherweise eine Nachricht, welche den Fehler umschreibt, sowie eine genaue Angabe, wo im Quellcode die `Exception` geworfen wurde

```
exception in thread "main"
java.lang.ArithmeticException: / by zero at
ExceptionDemo.main(ExceptionDemo.java:11)
```

- Laufzeitfehler können ...
  - ... ignoriert werden (d.h. wir lassen das Programm u.U. absichtlich abstürzen)
  - ... dort aufgefangen werden, wo diese auftreten
  - ... an die aufrufende Methode weitergegeben werden

1.8.1 Laufzeitfehler abfangen

```
public class ExceptionHandling {

    public static void main(String[] args) {

        Scanner scan = new Scanner(System.in);
        boolean ok = false;
        int num = -1;
        while (!ok) {
            try {
                System.out.print("Geben Sie eine ganze Zahl ein:");
                num = Integer.parseInt(scan.nextLine());
                ok = true;
            }
            catch (NumberFormatException exception){
                System.out.println("Das war keine gültige Eingabe!");
            }
        }
        System.out.println("Ihre Eingabe: " + num);
    }
}
```

- Eine `try-catch` Anweisung darf beliebig viele `catch` Klauseln enthalten
- Optional kann ein `finally` Block hinzugefügt werden: Dieser Block wird in jedem Fall ausgeführt (Normalfall, Behandelter Fehlerfall, Unbehandelter Fehlerfall)
- Jede Subklasse der Klasse `Exception` stellt (via Vererbung) zwei Methoden zur Verfügung

```
try {
    // Code, der eine Exception werfen kann
}
catch (Exception exception){
    // gibt die in exception gespeicherte Fehlermeldung aus
    System.out.println(exception.getMessage());
    // gibt die Stapelverfolgung aus
    exception.printStackTrace();
}
```

1.8.2 Laufzeitfehler weitergeben

- Methoden dürfen auftretende `Exception` Objekte an die aufrufende Methode weitergeben

```
public class ExceptionHandling {

    public static void main(String[] args) {
        try {
            int value = readInput();
            System.out.println(value * value);
        } catch (InputMismatchException exc) {
            System.out.println("Ungültige Eingabe!");
        }
    }

    private static int readInput() throws InputMismatchException {
        Scanner scan = new Scanner(System.in);
        System.out.print("Geben Sie eine ganze Zahl ein: ");
        int num = scan.nextInt();
        return num;
    }
}
```

1.8.3 Eigene `Exception` Klassen

- Eigene `Exception` Klassen können erstellt werden, um Fehlermeldungen zu spezifizieren
- Eigene `Exception` Klassen können auch verwendet werden, um Fehler zu signalisieren

```
public class InvalidParameterException extends Exception {
    public InvalidParameterException(int parameter) {
        super(parameter + " ist kein gültiger Parameter. Siehe Handbuch!");
    }
}
```

```
public class ExceptionHandling {

    public static void main(String[] args) {
        try {
            doSomething(17);
            doSomething(-17);
        } catch (InvalidParameterException e) {
            System.out.println(e.getMessage());
        }
    }

    private static void doSomething(int i) throws InvalidParameterException {
        if (i < 0)
            throw new InvalidParameterException(i);
        else
            System.out.println(i);
        }
    }
}
```

tl;dr Kapitel 1 bis 13

2.1 Grundlagen

- *Programmieren* := Lösen von Problemen mit Software
- *Programmiersprache* := Wörter und Regeln um *Programmieranweisungen* zu definieren
- *Java* ist eine *weit verbreitete, vielfältig einsetzbare, plattformunabhängige, objektorientierte* Programmiersprache
- Java Programme werden mit *Klassen* erstellt
- Klassen enthalten *Methoden* (Verhalten) und *Variablen* (Eigenschaften)
- Die Methode `main` ist der Startpunkt eines jeden Java Programmes
- *Kommentare* erläutern, *weshalb* oder *wozu* Sie etwas tun:
  - `//` oder `/* */` oder `/** */`

```
public class Quote {
        /**
        * Gibt ein Zitat von Steve Jobs aus
        */
        public static void main(String[] args) {
            System.out.println("Steve Jobs:");
            System.out.println("Es ist besser, ein Pirat zu sein, "
                        + "als der Marine beizutreten.");
    }
}
```

2.1.1 Datentypen und Konventionen

- *Bezeichner* gehören zu einer der drei Kategorien:
  - Wörter, die für einen *bestimmten Zweck reserviert* sind (`class`, `int`, ...)
  - Wörter, die etwas aus *diesem* Programm bezeichnen (*eigene* Methode oder *eigene* Variable)
  - Wörter, die etwas aus dem *Java API* bezeichnen (`System`, `main`, `println`, ...)
- *Konventionen* für Bezeichner:
  - Klassen: `Student` oder `StudentActivity`
  - Methoden: `start` oder `findMin`
  - Variablen: `grade` oder `nextItem`
  - Konstanten: `MIN` oder `MAX_CAPACITY`
- *Java Quellcode* wird mit `javac` in *Bytecode übersetzt* (kompiliert)
- *Java Bytecode* wird mit `java` ausgeführt (*interpretiert*)
- *Fehler*:
  - Fehler beim Kompilieren (*Kompilier-* oder *Syntaxfehler*)

  - Fehler beim Interpretieren (*Laufzeitfehler*)
  - Fehler in der Semantik (*Logische Fehler*)
- Zeichen innerhalb doppelter Anführungszeichen sind *Zeichenketten*: `"Hallo Java"`
- Zeichenketten sind Objekte der Klasse `String`
- Zeichenketten können mittels *Konkatenation* miteinander «verklebt» werden: `"Hallo" + " " + "Java"`
- Gewisse Sonderzeichen erfordern *Escape-Sequenzen*: `"\n"` oder `"\t"`

```
System.out.println("Es gibt unendlich viele Primzahlen. Ein System, "
        + "welche Zahlen Primzahlen sind, ist nicht bekannt.");

System.out.println("\"Die Anzahl der Dummheiten übersteigt die der "
        + "Primzahlen.\nGibt es nicht unendlich viele "
        + "Primzahlen?\"\n\tGregor Brand");

System.out.println("Daher zählt die " + 1 + " nicht zu den Primzahlen.");
```

2.1.2 Variablen und Datentypen

- *Variable* := Speicherort für einen Wert oder ein Objekt
- Variablen müssen mit *Datentyp* und *Bezeichner* deklariert werden
- Mit dem *Zuweisungsoperator* werden deklarierten Variablen Werte zugewiesen: `int i = 17;`
- Definierte Variablen können gelesen (*referenziert*) werden

```
int pages;
pages = 256;
int figures = 46, tables;
tables = 17;

System.out.println("Anzahl Seiten des Buches: " + pages);
System.out.println("Anzahl Abbildungen: " + figures
+ "; Anzahl Tabellen: " + tables);
```

- Konstanten werden mit `final` modifiziert: `final int MIN = 0;`
- Java kennt acht *primitive* Datentypen: (`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`)
- Wechsel von «kleinen» zu «grossen»:

```
int count = 17;
double num = count; // num = 17.0
```

- Wechsel von «grossen» zu «kleinen» via *Cast*:

```
double num = 12.34;
int count = (int) num; // num = 12.34; count = 12
```

2.1.3 Division

- `int`:

```
int ergebnisInt = 5 / 2; // = 2 (Bruchteil abgeschnitten)
```

- `double`:

```
double ergebnisDouble = 5.0 / 2.0; // = 2.5 (genaues Ergebnis)
```

- **Casting nach Division**:

```
double ergebnisMitCasting = (double)(5 / 2); // = 2.0 (ungenaues Ergebnis)
```

- *Ausdruck* := Kombination von einem oder mehreren *Operanden* und *Operatoren*
- Operanden sind Werte, Variablen oder Konstanten
- *Arithmetische Ausdrücke*:

```
double grade = (double) points / MAX_POINTS * 5 + 1;
```

- Lesen verändert Variablen niemals: `MAX\_POINTS * 5`
- *Zuweisungsoperatoren* und das *Inkrement/Dekrement* machen das Leben einfacher:

```
points = points * 2;
points *= 2;

points = points + 1;
points++;
```

2.1.4 Boolsche Ausdrücke und Verzweigungen

- *Boolesche Ausdrücke* sind entweder `true` oder `false`
- Boolesche Ausdrücke oder Boolesche Variablen können kombiniert und negiert werden

```
boolean smaller = hours < MAX;
boolean decision = (hours < MAX || hours > MIN) && !complete;
```

- Die `if`-*Anweisung* ist eine «Verzweigung», die auf einem Booleschen Ausdruck basiert:

```
if (hours < MAX) {
    hours += 10;
    System.out.println("10 Stunden hinzugefügt.");
    } else
    System.out.println("ACHTUNG: Maximum erreicht!");
```

2.1.5 Java API

- Das *Java API* besteht aus verschiedenen *Packages*, welche Klassen beinhalten, die Lösungen für *häufige Aufgaben* bereitstellen
- Sie kennen verschiedene Klassen: `String`, `Scanner`, `Random`, `DecimalFormat`.
- Der `new`-Operator *instanziiert* mit dem Aufruf des *Konstruktors* ein Objekt aus einer *Klasse* (Datentyp einer Objektvariablen := Klasse)

```
String str = new String("Hallo Welt");
Scanner scan = new Scanner(System.in);
Random rand = new Random();
```

2.1.6 Methoden

- Methoden können mit dem *Punkt-Operator* auf instanziierten Objekten aufgerufen werden

```
int length = str.length();
int number = scan.nextInt();
double randomNumber = rand.nextFloat();
```

2.1.7 Datentypen

- *Primitive Datentypen*: Kopien von Variablen sind *unabhängig*

```
int num1 = 17;
int num2 = num1;
num2 = 99;
System.out.println(num1); // 17
System.out.println(num2); // 99
```

- *Objektvariablen*: Kopien von Variablen sind *abhängig* (*Aliase*)

```
Integer num1 = new Integer(17);
Integer num2 = num1;
num2.setValue(99);
System.out.println(num1); // 99
System.out.println(num2); // 99
```

2.2 Klassen und Methoden

2.2.1 Sichtbarkeitsmodifikatoren

- Klassen enthalten *Variablen* und *Methoden* (Eigenschaften und Verhalten)
- *Sichtbarkeitsmodifikatoren* (`public`/`private`) bestimmen, was extern oder nur intern referenziert werden kann
- Variablen sollten `private` deklariert werden. Methoden können `private` oder `public` deklariert werden (je nach Zweck)

```
public class Integer {

    private int value;

    public Integer(int value) {
    this.value = value;
    }

    public String toString() {
    return this.value + "";
    }

    public void setValue(int value) {
    this.value = value;
    }
}
```

2.2.2 Methoden

- Methoden bestehen aus *Methodenkopf* und *Methodenrumpf*

- Methodenkopf: (1) *Sichtbarkeit* (2) *Datentyp* der Rückgabe oder `void` (3) *Bezeichner* (4) *Formale Parameter* in Klammern
- Konstruktoren besitzen *keinen* Rückgabetyp und heissen immer gleich wie die zugehörige Klasse

```
public Integer(int value) {
    this.value = value;
}

public String toString() {
    return this.value + "";
}

public void setValue(int value) {
    this.value = value;
}
```

- Konstruktoren instanziieren Objekte aus der Klasse und geben eine *Referenz* auf das Objekt zurück

```
Integer num1 = new Integer(17);
```

- *Tatsächlicher Parameter*: Wird beim Aufruf an die Methode mitgegeben

```
num1.setValue(99);
```

- *Formaler Parameter*: Bezeichner, in den der tatsächliche Parameter kopiert wird

```
public void setValue(int value) {
    this.value = value;
}
```

- Methoden können mit einer `return`-Anweisung «etwas» zurückgeben
- Der Rückgabetyp und der Datentyp der Rückgabe müssen übereinstimmen

```
public String toString() {
    return this.value + "";
}
```

```
String value = num1.toString();
```

2.2.3 Wrapper Klassen

- Die Klasse `Math` bietet mathematische Funktionen als *statische Methoden* an
- Statische Methoden können «direkt» ohne instanziiertes Objekt aufgerufen werden: `Math.sqrt(3);`
- Statische Methoden können auch verschachtelt werden:
  - `Math.sqrt(1 - Math.pow(Math.sin(alpha), 2));`
- Für jeden primitiven Datentyp existiert im Java API eine entsprechende *Wrapper Klasse*
- Hauptaufgabe dieser Klassen ist es, einen primitiven Datenwert zu umhüllen
  - `Double d = 4.567;`
- Die Wrapper bieten zudem hilfreiche statische Methoden und Konstanten an:
  - `Integer.MAX_VALUE`

  - `Double.parseDouble("4.567");`
  - `Double.POSITIVE_INFINITY`
  - `Boolean.toString(true)`

- `while`-Anweisungen erlauben es, gewisse Anweisungen mehrfach auszuführen, ohne diese mehrfach programmieren zu müssen

```
Random rand = new Random();
    System.out.println("1 : " + rand.nextInt(100));
    System.out.println("2 : " + rand.nextInt(100));
    System.out.println("3 : " + rand.nextInt(100));
    System.out.println("4 : " + rand.nextInt(100));
    System.out.println("5 : " + rand.nextInt(100));
    System.out.println("6 : " + rand.nextInt(100));
    System.out.println("7 : " + rand.nextInt(100));
    System.out.println("8 : " + rand.nextInt(100));
    System.out.println("9 : " + rand.nextInt(100));
    System.out.println("10 : " + rand.nextInt(100));
```

```
Random rand = new Random();
    int i = 1;
    while (i <= 10) {
        System.out.println(i + " : " + rand.nextInt(100));
        i++;
    }
```

- Mit *Wächterwerten* können wir ein Programm kontrollieren:

```
Scanner scan = new Scanner(System.in);
int input = 1;
while (input != 0) {
    System.out.print("Mit 0 Beenden Sie den Prozess. ");
    input = scan.nextInt();
}
System.out.println("--ENDE--");
```

- `while`-Schleifen können auch zur Kontrolle von Eingaben verwendet werden:

```
Scanner scan = new Scanner(System.in);
System.out.print("Alter eingeben: ");
int age = scan.nextInt();
while (age < 0) {
    System.out.println("Ungültiger Wert.");
    System.out.print("Alter eingeben: ");
    age = scan.nextInt();
}
```

- `while`-Schleifen können auch verschachtelt werden:

```
int counter = 2;
while (counter <= 20) {
    System.out.print("Teiler von " + counter + ":\t");
    int divisor = 1;
    while (divisor <= counter / 2) {
        if (counter % divisor == 0)
            System.out.print(divisor + " ");
        divisor++;
    }
    System.out.println();
    counter++;
}
```

2.2.4 Generische Klassen

- Wir können Klassen *generisch* machen:

```
public class Rocket<T> {

    private T cargo;

    public Rocket(T cargo) {
    this.cargo = cargo;
    }

    public void set(T cargo) {
    this.cargo = cargo;
    }

    public T get() {
    return this.cargo;
    }
}
```

- Um eine generische Klasse zu instanziieren, müssen wir sie zusammen mit einem *Typargument* instanziieren:
  - `Rocket<Integer> intRocket = new Rocket<Integer>();`
- Die *Typvariable* `T` wird nun überall mit dem Typargument `Integer` ersetzt
- Die Klasse `ArrayList` erlaubt es, Sammlungen von Objekten des Typs `T` anzulegen.
- Objekte dieser Klasse werden bei der Instanziierung *parametrisiert*:

```
ArrayList<String> names= new ArrayList<String>();
ArrayList<PlayerCard> cards = new ArrayList<PlayerCard>();
ArrayList<Integer> numbers = new ArrayList<Integer>();
```

2.2.5 Arrays

- Listen passen Ihre Grösse dynamisch an:

```
names.add("Keanu");
names.add("Kevin");
System.out.println(names); // [Keanu, Kevin]
names.add("Karl");
System.out.println(names); // [Keanu, Kevin, Karl]
names.remove(1);
System.out.println(names); // [Keanu, Karl]
```

2.2.6 `switch`-Anweisung

- Die `switch`-Anweisung bietet eine Alternative für (stark) verschachtelte `if`-Anweisungen:

```
if (i == 1)
    System.out.println("Eins");
else
    if (i == 2)
        System.out.println("Zwei");
    else
        if (i == 3)
            System.out.println("Drei");
        else
            if (i == 4)
                System.out.println("Vier");
            else
                if (i == 5)
                    System.out.println("Fünf");
                 else
                         System.out.println("Irgendwas anderes");
```

```
switch (i) {
    case 1: System.out.println("Eins"); break;
    case 2: System.out.println("Zwei"); break;
    case 3: System.out.println("Drei"); break;
    case 4: System.out.println("Vier"); break;
    case 5: System.out.println("Fünf"); break;
    default: System.out.println("Irgendwas anderes");
}
```

2.2.7 Conditionals

- Der *Conditional*bietet eine elegante Möglichkeit bei alternativen Zuweisungen:

```
if (points > MAX)
    points = points + 1;
else
    points = points * 2;
```

```
points = (points > MAX) ? points + 1 : points * 2;
```

2.2.8 `do`-Anweisung

- Die `do`-Anweisung ist ähnlich zur `while`-Anweisung, evaluiert aber die Boolesche Bedingung am Ende der Schleife:

```
System.out.print("Erreichte Punkte (0 bis 100): ");
int points = scan.nextInt();
while (points < 0 || points > 100) {
    System.out.print("Erreichte Punkte (0 bis 100): ");
    points = scan.nextInt();
}
```

```
int points;
do {
    System.out.print("Erreichte Punkte (0 bis 100): ");
    points = scan.nextInt();
} while (points < 0 || points > 100);
```

2.2.9 Schleifen

- Die `for`-Schleife ist gut geeignet, wenn man von Anfang an weiss, wie oft diese durchgeführt werden muss.
- Der Schleifenkopf der `for`-Schleife besteht aus drei Teilen:
  - *Initialisierung*: Wird am *Anfang* und *genau einmal* durchgeführt
  - *Boolesche Bedingung*: Wird immer *vor* dem nächsten Eintritt in die Schleife überprüft
  - *Inkrement*: Wird *immer am Ende* der Schleife durchgeführt

```
for (int i = 0; i < 10; i++)
    System.out.print(Math.pow(i, 2) + " ");
```

- Variante: `for`-`each`-Schleife - in jedem Durchgang zeigt die Variable auf das nächste Element einer `ArrayList`

```
for (String s : list)
    System.out.println(s);
```

- Vorsicht bei == auf Dezimalzahlen

```
final double TOLERANCE = 0.00000001;
if (Math.abs(num1 - num2) < TOLERANCE) {
```

- Vergleich von Zeichen basiert auf Unicode (Ziffern < Grossbuchstaben < Kleinbuchstaben)

```
char c0 = '0', c1 = 'A', c2 = 'a';
System.out.println(c0 < c1); // true
System.out.println(c1 < c2); // true
```

2.2.10 Gleichheit von Objekten

- Vorsicht bei == auf Objekten: Testet auf *Aliase*
- Verwenden/Schreiben der Methode `equals` und der Methode `compareTo`

```
public class Integer {

    private int value;

    public Integer(int value) {
    this.value = value;
    }

    public boolean equals(Integer other) {
    return this.value == other.value;
    }

    public int compareTo(Integer other) {
    return this.value - other.value;
    }
}
```

```
Integer i1 = new Integer(2);
Integer i2 = new Integer(17);
Integer i3 = new Integer(2);

System.out.println(i1.equals(i2)); // false
System.out.println(i1.equals(i3)); // true

System.out.println(i1.compareTo(i2)); // -15
System.out.println(i1.compareTo(i3)); // 0
System.out.println(i2.compareTo(i3)); // 15
```

2.3 Arrays

- *Arrays* ermöglichen das Deklarieren einer *einzigen* Variablen eines Typs, die dann *mehrere Werte* dieses Typs speichern kann.
- Arrays haben eine *feste, unveränderliche Größe* (Konstante `length`), die bei der Instanziierung angegeben werden muss.

```
int num1, num2, num3, num4, num5, num6;
```

```
int[] nums = new int[6];
```

```
int l = nums.length;
```

- Auf einzelne Elemente eines Arrays greift man mit einem *Index* innerhalb eckiger Klammern zu.

```
for (int i = 0; i < nums.length; i++) {
  System.out.println(nums[i]);
}
```

- Mit *Initialisierungslisten* können Arrays instanziiert und mit Werten gefüllt werden.

```
int[] nums = {1, 2, 3, 4};
String[] names = {"Goodbye", "Hello", "Hi", "Howdy"};
```

- Auf Methoden der in einem Array gespeicherten Objekte kann man über *Array-Referenzen* zugreifen.

```
for (int i = 0; i < names.length; i++) {
  names[i] = names[i].toUpperCase();
}
```

2.3.1 `main`-Methode

- Der Parameter der Methode `main` ist ein `String[]`. Dies sind *Programmparameter*, die beim Start des Programmes von „Außen“ mitgegeben werden können.

```
public static void main(String[] args) {
  String language = args[0];
  String version = args[1];
  String author = args[2];
}
```

- Methoden können mit *variablen Parameterlisten* umgehen.

```
public static int min(int first, int ... others) {
  int min = first;
  for (int num : others) {
    min = Math.min(min, num);
  }
  return min;
}
```

```
public class Greetings {
  private String primaryGreeting;
  private String[] greetings;

  public Greetings(String primaryGreeting, String ... otherGreetings) {
    this.primaryGreeting = primaryGreeting;
    this.greetings = otherGreetings;
  }
}
```

- Zweidimensionale Arrays sind Arrays aus Arrays.

```
int[][] table = new int[100][5];
String[][] names = {{"Anne", "Barbara", "Cathrine"}, {"Danny", "Emilie", "Fanny"}};
```

- Für Referenzen auf Elemente in zweidimensionalen Arrays werden zwei Indizes benötigt.

```
System.out.println(names[0][2]);

for (int row = 0; row < names.length; row++) {
  for (int col = 0; col < names[row].length; col++) {
    names[row][col] = "Hannes";
  }
}
```

2.3.2 `enum`

- Ein `enum` zählt *alle* zulässigen Werte eines Typs auf.

```
public enum Category {
  Mathematik, Geographie
}
```

```
public enum Category {
  Mathematik(10), Geographie(3);

  private int points;

  private Category(int points) {
    this.points = points;
  }
}
```

- Jedes `enum` Objekt besitzt Methoden (wie z.B. `name()`).
- Jede `enum` Klasse besitzt statische Methoden (wie z.B. `values()`).

```
Category[] categories = Category.values();
for (Category category : categories) {
  System.out.println(category.name());
}
```

2.3.3 Statische Variablen

- *Statische Variablen* werden von allen Instanzen geteilt (es existiert also nur *eine* Kopie der Variablen für alle Objekte).

```
public class Person {
  public static int globalCount = 0;
  private int id;

  public Person() {
    this.id = Person.globalCount++;
  }
}
```

- Statische Methoden werden direkt aufgerufen, ohne vorher ein Objekt zu instanzieren.

```
Functions.generateRandoms();

public static int[] generateRandoms() {
  // Macht etwas
}
```

2.3.4 Abhängigkeiten

Selbstabhängig

- Eine Klasse kann von sich selbst abhängig sein.

```
Person p1 = new Person("Emilie");
Person p2 = new Person("Ava");
Person p3 = new Person("Maya");

p1.knows(p2);
p1.knows(p3);

System.out.println(p1.getFriends()); // [Ava, Maya]
```

```
public class Person {
  private String name;
  private ArrayList<Person> friends;

  public Person(String name) {
    this.name = name;
    this.friends = new ArrayList<Person>();
  }

  public void knows(Person other) {
    this.friends.add(other);
  }

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

  public ArrayList<Person> getFriends() {
    return this.friends;
  }
}
```

Aggregation

- Aggregation := Ein Objekt besteht z.T. aus anderen Objekten.

```java
public class Person {
    private String name;
    private Address address;

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }
}

public class Address {
    private String street;
    private int zipCode;

    public Address(String street, int zipCode) {
        this.street = street;
        this.zipCode = zipCode;
    }
}
```

- Primitive Datentypen: tatsächliche und formale Parameter sind **unabhängig**

```java
int val = 17;
change(val);
System.out.println(val); // unchanged!
```

- Objektvariablen: tatsächliche und formale Parameter sind **Aliase**

```java
OwnInt val = new OwnInt(17);
change(val);
System.out.println(val); // changed!
```

- Methoden können **überladen** werden
- **Signatur** (:= *Bezeichner + formale Parameter*) muss eindeutig sein

```java
private void doThis(int val) {
}

private void doThis(int val1, int val2) {
}

private void doThis(double val) {
}
```

- *Sortieren* und *Suchen* sind zwei klassische Problemfelder der Informatik
- Es existieren zahlreiche Lösungen/Algorithmen für beide Problemfelder
- Methoden können auch generisch programmiert werden:

```java
public static <T> void insertionSort(Comparable<T>[] list) {
}
```

- Die Typvariable **T** wird beim Aufruf der Methode definiert:

```
Contact[] friends = new Contact[8];
// ...
Sorting.insertionSort(friends);
```

- Jede Funktion, die ein Programm erfüllen soll, muss in einer Methode programmiert sein
- Komplexe Funktionalitäten sollten Sie in mehrere Teile zerlegen
  - Verständlicher
  - Wiederverwendbar
  - leichter zu testen
  - schneller erstellt

```
public static <T> void bubbleSort(Comparable<T>[] list) {
    for (int i = 0; i <= list.length - 2; i++)
        for (int j = list.length - 1; j >= i + 1; j--)
            if (list[j].compareTo((T) list[j - 1]) < 0)
                swap(list, j, j - 1);
}

private static <T> void swap(Comparable<T>[] list, int j, int i) {
    Comparable<T> temp = list[j];
    list[j] = list[i];
    list[i] = temp;
}
```

- Testen umfasst mindestens das Ausführen eines vollendeten Programms mit verschiedenen Eingaben
- **Test Case** := Eingaben und Benutzeraktionen, gekoppelt mit den erwarteten Ergebnissen
- **Test Suite** := Mehrere Test Cases
- **Defekttest** := Ziel ist es, Fehler zu finden
- **Regressionstest** := Ziel ist es, nach der Korrektur keine neuen Fehler einzubauen
- **Black-Box Test**: Beruht nur auf Eingaben und erwarteten Ausgaben
- **White-Box Test**: Konzentriert sich auf die interne Struktur des Codes
- **Sammlung** :=Behälter, um (meist gleichartige) Elemente zu organisieren.
- **Idee** einer Sammlung und die **Implementierung** sind zwei unterschiedliche Dinge
- Wichtige abstrakte Datentypen und deren Implementierungen im Java API:
  - **Listen** (z.B. die Klasse LinkedList)
    - Verarbeitung über Index
  - **Warteschlangen** (z.B. die Schnittstelle Queue)
    - FIFO Verarbeitung
  - **Stapel** (z.B. die Klasse Stack)
    - LIFO Verarbeitung
  - **Mengen** (z.B. die Klasse TreeSet)
    - Keine Duplikate, keine Position
  - **Assoziativspeicher** (z.B. die Klasse HashMap)
    - Schlüssel-Wert Paare
- Das **Collection Framework** des Java APIs erlaubt die Trennung von abstrakten Datentypen (:= Schnittstellen) und Implementierungen:

```
List<Door> list = new ArrayList<Door>();

List<Door> list = new LinkedList<Door>();
```

- Einige Methoden sind „weit oben“ in der Hierarchie deklariert:
  - `add(T t)` oder `size()`
- Einige Methoden machen nur auf speziellen Datenstrukturen Sinn:
  - `add(int index, T t)` oder `peekFirst()`

2.4 Rekursion

- **Rekursion**: Etwas ist durch sich selbst definiert
  - Liste := Zahl
  - Liste := Zahl, Liste
- Jede rekursive Definition benötigt einen *nicht-rekursiven Teil*, den *Basisfall*, so dass die Rekursion enden kann
- **Rekursive Methode**: Die Methode ruft sich selber direkt oder indirekt auf
- **Jeder rekursive Aufruf einer Methode definiert eigene lokale Variablen und eigene formale Parameter**

```java
public class Recursion {
    public static void main(String[] args) {
        System.out.println(f(5, 4));
    }

    public static double f(int n, int m) {
        if (m == 1)
            return n;
        else
            return f(n, m - 1) + n;
    }
}
```

- **Mit Rekursion kann man einige Probleme elegant lösen**

```java
public static <T> void quickSort(Comparable<T>[] list, int p, int r) {
    if (p < r) {
        int q = partition(list, p, r);
        quickSort(list, p, q - 1);
        quickSort(list, q + 1, r);
    }
}
```

- **Rekursive Algorithmen sind aber manchmal nicht so intuitiv wie iterative Algorithmen**

```java
public static double f_1(int n, int m) {
    if (m == 1)
        return n;
    else
        return f_1(n, m - 1) + n;
}

public static double f_2(int n, int m) {
    return n * m;
}
```

2.4.1 Laufzeitfehler

- **Laufzeitfehler** sind Objekte der Klasse `Exception`, die eine unübliche Situation repräsentieren
- Fehlermeldungen beinhalten den **Namen** der `Exception`, möglicherweise eine **Nachricht**, welche den Fehler umschreibt, sowie eine genaue Angabe, **wo im Quellcode** die `Exception` geworfen wurde

```
exception in thread "main"
java.lang.ArithmeticException: / by zero at
ExceptionDemo.main(ExceptionDemo.java:11)
```

- **Laufzeitfehler können . . .**
  - . . . **ignoriert** werden (d.h. wir lassen das Programm u.U. absichtlich abstürzen)
  - . . . **dort aufgefangen** werden, wo diese auftreten
  - . . . an die **aufrufende Methode weitergegeben** werden

```java
public class ExceptionHandling {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        boolean ok = false;
        int num = -1;
        while (!ok) {
            try {
                System.out.print("Geben Sie eine ganze Zahl ein:");
                num = Integer.parseInt(scan.nextLine());
                ok = true;
            } catch (NumberFormatException exception) {
                System.out.println("Das war keine gültige Eingabe!");
            }
        }
        System.out.println("Ihre Eingabe: " + num);
    }
}
```

- Eine `try-catch` Anweisung darf beliebig viele `catch` Klauseln enthalten
- Optional können Sie einen `finally` Block hinzufügen: Dieser Block wird in jedem Fall ausgeführt (Normalfall, Behandelter Fehlerfall, Unbehandelter Fehlerfall)
- Jede Subklasse der Klasse `Exception` stellt (via Vererbung) zwei Methoden zur Verfügung

```java
try {

}
catch (Exception exception){
  // gibt die in exception gespeicherte Fehlermeldung aus
  System.out.println(exception.getMessage());
  // gibt die Stapelverfolgung aus
  exception.printStackTrace();
}
```

- **Methoden dürfen auftretende Exception Objekte an die aufrufende Methode weitergeben**

```java
public class ExceptionHandling {

    public static void main(String[] args) {
        try {
            int value = readInput();
            System.out.println(value * value);
        } catch (InputMismatchException exc) {
            System.out.println("Ungültige Eingabe!");
        }
    }

    private static int readInput() throws InputMismatchException {
        Scanner scan = new Scanner(System.in);
        System.out.print("Geben Sie eine ganze Zahl ein: ");
        int num = scan.nextInt();
        return num;
    }
}
```

- **Wir können eigene Exception Klassen definieren**

```java
public class InvalidParameterException extends Exception {
    public InvalidParameterException(int parameter) {
        super(parameter + " ist kein gültiger Parameter. Siehe Handbuch!");
    }
}
```

```java
public class ExceptionHandling {

    public static void main(String[] args) {
        try {
            doSomething(17);
            doSomething(-17);
        } catch (InvalidParameterException e) {
            System.out.println(e.getMessage());
        }
    }

    private static void doSomething(int i) throws InvalidParameterException {
        if (i < 0)
            throw new InvalidParameterException(i);
        else
            System.out.println(i);
    }
}
```

- Die Klasse `PrintWriter` erlaubt Ausgaben in Dateien

```java
public static void main(String[] args) throws IOException {
  String fileName = "output.txt";
  PrintWriter outFile = new PrintWriter(fileName);
  outFile.print("Hallo Welt!“);
  outFile.close();
}
```

UML Mini Cheat Sheet

Wir fokussieren uns in P1 auf UML *Klassendiagramme*, um die Inhalte von Klassen und deren Beziehungen zu visualisieren.

In einem UML Klassendiagramm wird jede Klasse in einem Rechteck mit drei Bereichen repräsentiert. Der erste Bereich enthält den Klassennamen, der zweite Bereich enthält die Variablen und der dritte Bereich die Methoden der Klasse. In Abb. 1 ist ein Klassendiagramm gezeigt.

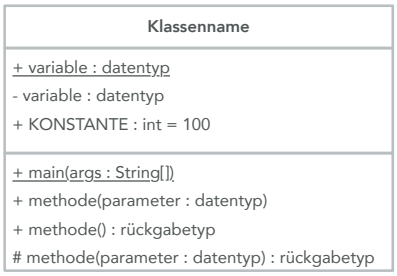


Abbildung 1: Ein UML Klassendiagramm.

UML wurde nicht speziell für Java entwickelt (UML ist eine sprachunabhängige Notationsmöglichkeit). Die Syntax in UML Diagrammen ist deshalb nicht identisch mit der Syntax von Java.

- Die Sichtbarkeiten der Variablen und Methoden werden mit einem Plus (+) für `public`, einem Minus (−) für `private` und mit einem Hash (#) für `protected` visualisiert.
- Nach dem Bezeichner einer Variablen folgt nach einem Doppelpunkt der zugehörige Datentyp.
- Optional kann man nach einem Gleichheitszeichen noch einen initialen Wert der Variablen oder der Konstanten definieren.
- Die Parameter einer Methode werden in Klammern nach dem Bezeichner der Methode ebenfalls nach dem Muster *(Parameter-Bezeichner : Datentyp)* aufgelistet.
- Nach den Parametern in Klammern wird nach einem Doppelpunkt der Rückgabetyp der Methode angegeben.
- Den Typ `void` gibt es in UML nicht. Das heisst, bei Methoden ohne Rückgabe wird in UML der Rückgabetyp einfach weggelassen.
- Bezeichner für statische Variablen und statische Methoden werden unterstrichen dargestellt.

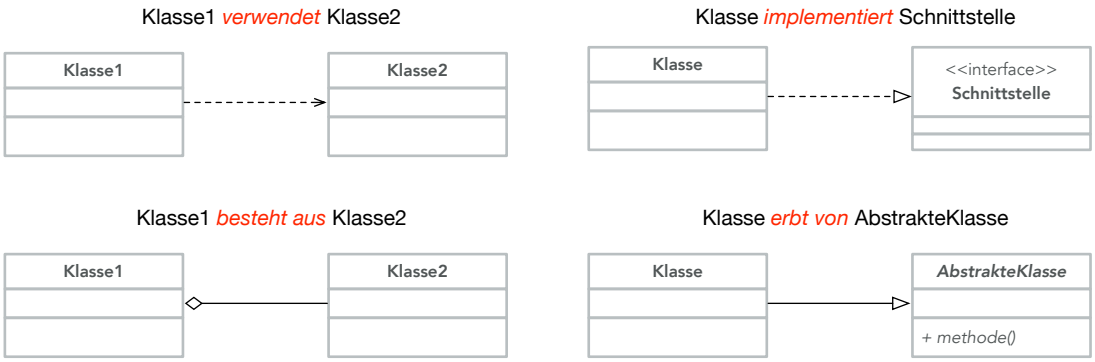


Abbildung 2: Verschiedene Beziehungstypen in einem UML Klassendiagramm.

Der Pfeil, der die zwei Klassen miteinander verbindet, zeigt an, dass eine Beziehung zwischen diesen beiden Klassen existiert. Jeder Beziehungstyp wird in UML mit anderen Typen von Pfeilen visualisiert (Siehe Abb. 2):

- Eine gestrichelte Linie mit offenem Pfeilkopf bedeutet, dass die eine Klasse die andere Klasse *verwendet*.
- Eine Beziehung *besteht aus* nennt man *Aggregation* und wird in UML mit einer ausgezogenene Linie mit leerer Raute dargestellt[1]. Die Raute befindet sich auf der Seite des aggregierten Objektes.
- In UML wird eine Schnittstelle wie eine Klasse dargestellt, mit dem Unterschied, dass `<<Interface>>` oberhalb des Bezeichners der Schnittstelle steht. Eine gestrichelte Linie mit einer geschlossenen Pfeilspitze wird von der Klasse zur Schnittstelle gezeichnet und bedeutet *implementiert*.
- Mit einem ausgezogenen Pfeil mit weissem Kopf zeigt man Vererbung an (eine Subklasse *erbt von* einer Superklasse). Beachten Sie, dass in UML die Bezeichner von abstrakten Klassen und abstrakten Methoden kursiv dargestellt werden.

[1]Die Aggregationen aus `String` Objekten macht man nicht explizit sichtbar. `String` Objekte sind so fundamental, dass diese in UML wie primitive Datentypen behandelt werden.

P1 - Serie 01

Lukas Batschelet (16-499-733)

(01) Bezeichner

Finden Sie passende Bezeichner für...

- eine Java Klasse, die eine Prüfung repräsentieren soll.
- die erreichten Punkte in einer Prüfung.
- eine Methode, welche den Durchschnittswert aller Prüfungen berechnet.
- die maximale Punktzahl, die in einer Prüfung erreicht werden kann.

```java
1. public class Exam {}
2. int points
3. meanPoints()
4. final int MAX_POINTS
```

(02) Variablen und Eigenschaften

Finden Sie für die Klasse `Flight` mindestens drei Variablen (Eigenschaften) und drei Methoden (Verhalten/Funktionen), die in den Klassen modelliert werden könnten.

Variablen

1. `altitude`
2. `cargoWeight`
3. `fuelRemaining`

Methoden

1. `bookSeat()`
2. `cancelFlight()`
3. `readFuelLevel()`

(03) Zitat

Schreiben Sie eine einzige `println` Anweisung, die die folgende Zeichenkette als Ausgabe generiert:

```
"Mein Name ist Winston Wolfe.
Ich löse Probleme!", stellte er sich vor.
```

```java
System.out.println("\"Mein Name ist Winston Wolfe. \n" +
        "Ich löse Probleme!\", stellte er sich vor.");
```

(04) Rechnung

Welchen Wert enthält die Variable `result`, nachdem folgende Anweisungen durchgeführt worden sind?

```java
int result = 25;
result = result + 5;
result = result / 7;
result = result * 3;
```

```java
int result = 25;
result = result + 5; //30
result = result / 7; //4
result = result * 3; //12
```

(05) Rechnung

Welchen Wert enthält die Variable `result`, nachdem folgende Anweisungen durchgeführt worden sind?

```java
int result = 15, total = 100, min = 15, num = 10;
result /= (total - min) % num;
```

```
int result = 15, total = 100, min = 15, num = 10;
result /= (total – min) % num;
result = result / ((total – min) % num;
3 = 15 / (( 100 – 15) % 10;
```

(06) Operationen

Gegeben seien folgende Deklarationen:

```
int result1, num1 = 27, num2 = 5;
double result2, num3 = 12.0;
```

Welches Resultat wird jeweils durch folgende Anweisungen gespeichert?

1. `result1 = num1 / num2;`
2. `result2 = num1 / num2;`
3. `result2 = num3 / num2;`
4. `result1 = (int) num3 / num2;`
5. `result2 = (double) num1 / num2;`

```
int result1, num1 = 27, num2 =5;
double result2, num3 = 12.0;
a. result1 = num1 / num2; //5.0
b. result2 = num1 / num2; //5.4
c. result2 = num3 / num2; //2.4
d. result1 = (int) num3 / num2; //2
e. result2 = (double) num1 / num2; //5.4
```

(07) Boolsche Operationen

Gegeben seien folgende Deklarationen:

```
int val1 = 15, val2 = 20;
boolean ok = false;
```

Was ist der Wert der folgenden Booleschen Ausdrücke?

1. `val1 <= val2`
2. `(val1 + 5) >= val2`
3. `val1 < val2 / 2`
4. `val1 != val2`
5. `!(val1 == val2)`
6. `(val1 < val2) || ok`
7. `(val1 > val2) || ok`
8. `(val1 < val2) && !ok`
9. `ok || !ok`

```
int val1 = 15, val2 = 20;
Boolean ok = false;
a. true
b. true
c. false
d. true
e. true
f. true
g. false
h. true
i. true
```

Implementationsaufgaben

(01) Einfache Ausgabe - `WinterIsComing.java`

Schreiben Sie ein Programm, welches den Satz "Winter is coming" ausgibt (erste Version: auf einer Zeile; zweite Version: jedes Wort auf einer separaten Zeile).

```
public class WinterIsComing {
    public static void main(String[] args) {
                System.out.println("\"Winter is coming\"");
                System.out.println("\"Winter \n" +
                                   "is \n" +
```

```
                                "coming\"");
        }
}
```

(02) Einfache Berechnungen - `Quotient.java`

Schreiben Sie ein Programm, das vom Benutzer die Eingabe von zwei ganzzahligen Werten `a` und `b` fordert. Ihr Programm soll den Quotienten $\frac{a^2}{b}$ sowohl als Gleitkommazahl (d.h. ungerundet) als auch als ganze Zahl mit Rest berechnen und beide Ergebnisse am Bildschirm ausgeben. Testen Sie Ihr Programm mit beliebigen Zahlen.

Beobachten Sie insbesondere das Programmverhalten bei Eingabe der Zahl `0` als Divisor und versuchen Sie diesen Laufzeitfehler abzufangen.

```
import java.util.Scanner;

public class Quotient {
    public static void main(String[] args) {

        System.out.println("Dieses Programm berechnet den Quotienten zweier Zahlen \"a\" und \"b\".");

        Scanner scan = new Scanner(System.in);
        System.out.println("Geben Sie den Teil \"a\" ein:");
        double var1 = scan.nextDouble();

        System.out.println("Geben Sie den Teil \"b\" ein:");
        double var2 = scan.nextDouble();

        if (var2 != 0) { //Wert 0 führt zu divide by zero
            double quotientDouble = (var1 * var1) / var2;
            // int quotientInt = (int) var1 * (int) var1) / (int) var2;
            int quotientInt = (int) quotientDouble;
            System.out.println("_____\n" +
                "Der Quotient Ihrer Zahlen: \t" + quotientDouble +
                "\nUnd als \"int\":\t \t \t" + quotientInt);
        } else {
            System.out.println("Geben Sie nicht 0 ein!");
        }
        scan.close();
    }
}
```

(03) Benutzerinteraktion - `HumanThermometer.java`

Schreiben Sie ein Programm, das vom Benutzer die Eingabe einer Temperatur `t` fordert. Die Ausgabe Ihres Programmes definiert sich danach folgendermassen:

$$Ausgabe = \begin{cases} "Kalt", & \text{wenn } t < 15 \\ "Angenehm", & \text{wenn } 15 \leq t < 24 \\ "Warm", & \text{wenn } t \geq 24 \end{cases}$$

Hinweis: Verwenden Sie Konstanten für beide Temperaturgrenzen.

```
import java.util.Scanner;

public class HumanThermometer {
    public static void main(String[] args) {
                final int LOWER_BOUND = 15;
                final int UPPER_BOUND = 24;

                Scanner scan = new Scanner(System.in);
                System.out.println("Die aktuelle Temperatur: ");
                int temperature = scan.nextInt();

                if (temperature < LOWER_BOUND) {
                        System.out.println("Es ist kalt");
                }
                else if ((LOWER_BOUND <= temperature) && (temperature <= UPPER_BOUND)) {
                        System.out.println("Es ist angenehm");
                }
                else
                        System.out.println("Es ist warm");

                scan.close();
        }
}
```

P1 - Serie 02

Lukas Batschelet - 16-499-733

(01) `new`-Operator

Der `new`-Operator zusammen mit dem Konstruktor einer Klasse erledigt zwei Dinge. Was genau?

- Reservierung von Speicher
- Initialisierung des Objekts zusammen mit dem Konstruktor

(02) `ArrayList` in der Java API

Informieren Sie sich in der Java API Dokumentation über die Klasse `ArrayList` (welche eine Liste für Objekte repräsentiert).

- Wie instanziieren Sie eine solche Liste?
  - `ArrayList list = new ArrayList();`
  - Oder mit einer Zuordnung des Typs (hier mit dem Beispiel `String`)
    - `ArrayList<String> list = new ArrayList<>();`
- Wie fügen Sie ein Objekt zur Liste hinzu?
  - mit der Methode `.add()` wird ein Objekt am Ende der Liste eingefügt
    - Möglicher Parameter ist der Index vom Typ `int`, welcher angibt an welcher Position das Objekt in die Liste eingefügt werden soll
- Wie greifen Sie auf ein Objekt an Position `i` zu?
  - mit der Methode `.get(i)`
- Wie löschen Sie den gesamten Inhalt der Liste?
  - mit der Methode `.clear()`
- Wie können Sie überprüfen, ob ein bestimmtes Objekt in der Liste vorhanden ist?
  - mit der Methode `.contains()` wird ein `boolean` Wert zurückgegeben, ob das Objekt welches als Parameter "mitgegeben wurde" in der Liste ist

(03) Unterschied zwischen Klassen und Objekten

Erläutern Sie anhand der Klasse `String` und des Objektes `"String"` den Unterschied zwischen Klasse und Objekt.

- Die Klasse `String` gibt den "Bauplan" für alle Objekte in ihr vor
  - Sie definiert die Methoden (`length()`, `substring()`, etc.) welche auf einem Objekt der Klasse ausgeführt werden können
- Das Objekt `"String"` wurde mit dem "Bauplan" der Klasse `String` "gebaut"
  - Methoden (in der Klasse definiert) die darauf angewendet werden

(04) Ausgabe

Welche Ausgabe erzeugen folgende Anweisungen?

```java
String testString = "Think different";
System.out.println(testString.length());
System.out.println(testString.substring(0, 4));
System.out.println(testString.toUpperCase());
System.out.println(testString.charAt(7));
System.out.println(testString);
```

```
//Ausgabe
15
Thin
THINK DIFFERENT
i
Think different
```

(05) Random

Gegeben sei eine Objektvariable vom Typ Random mit Bezeichner `rand`. In welchen Intervallen suchen die folgenden Anweisungen eine Zufallszahl?

- `rand.nextInt(100) + 1;`
  - `[1, 100]`
- `rand.nextInt(51) + 100;`
  - `[100, 150]`
- `rand.nextInt(10) - 5;`
  - `[-5, 4]`

- `rand.nextInt(3) - 3;`
  - `[-3, -1]`

(06) Aliase

Was sind Aliase und weshalb können Aliase problematisch sein?

- Aliase sind Variablen, die auf dasselbe Objekt zeigen. Bei primitiven Datentypen passiert das nicht, da der Wert kopiert wird.

```java
int num1 = 17;
int num2 = num1;
num2 = 99; System.out.println(num1); // 17
System.out.println(num2); // 99
```

Bei Objektvariablen sieht es anders aus:

```java
Integer num1 = new Integer(17);
Integer num2 = num1;
num2.setValue(99);
System.out.println(num1); // 99
System.out.println(num2); // 99
```

Hier greifen beide Variablen auf das selbe Objekt zu. Das ist nicht immer wünschenswert, weil:

- Es ist unklar, welche Variable für was zuständig ist.
- Änderungen an einer Variable beeinflussen die andere.
- Der Code wird unübersichtlicher.
- Es kann sogenannter *garbage* und damit einhergehender Datenverlust entstehen.
  - Im obigen Beispiel aus der Vorlesung ist dem Objekt `num1` selber kein Wert mehr zugewiesen, sondern nur noch die Verweisung auf `num2`. Der Wert `17` wird damit zu *garbage*

Implementationsaufgaben

(01) Rechteck

Schreiben Sie ein Programm, das nach der Länge und Breite eines Rechtecks fragt und danach die Fläche und den Umfang des Rechtecks berechnet und ausgibt. Zusätzlich soll Ihr Programm feststellen, ob es sich beim definierten Rechteck um ein Quadrat handelt oder nicht und eine entsprechende Ausgabe erzeugen.

```java
import java.util.Scanner;

public class Rectangle {
    public static void main(String[] args) {

        System.out.println("Geben Sie die Länge des Rechtecks ein:");
        Scanner scan = new Scanner(System.in);
        double length = scan.nextDouble();
        System.out.println("Geben Sie die Breite des Rechtecks ein:");
        double width = scan.nextDouble();
        scan.close();

        double area = length * width;
        double perimeter = 2 * (length + width);

        System.out.println("Die Fläche des Rechtecks beträgt: " + area);
        System.out.println("Der Umfang des Rechtecks beträgt: " + perimeter);
        if (length == width) {
            System.out.println("Das Rechteck ist ein Quadrat.");
        } else {
            System.out.println("Das Rechteck ist kein Quadrat.");
        }
    }
}
```

(02) Zufällige Addition

Schreiben Sie ein Programm, das eine zufällige Additionsaufgabe mit zwei positiven Zahlen anzeigt. Die Summe der beiden Zahlen darf maximal 20 betragen. Der Benutzer soll dann ein Ergebnis eingeben können und das Programm soll überprüfen, ob die Eingabe korrekt war oder nicht und eine entsprechende Rückmeldung ausgeben.

```java
import java.util.Random;
import java.util.Scanner;

public class RandomAddition {
    public static void main(String[] args) {

        Random random = new Random();
        final int MAX = 21;
        int number1 = random.nextInt(MAX);
        // Die Summe der beiden Zahlen darf maximal 20 betragen.
        int number2 = random.nextInt(MAX - number1);
        int sum = number1 + number2;

        Scanner scan = new Scanner(System.in);

        System.out.println("Die Aufgabe lautet: " + number1 + " + " + number2);
        System.out.println("Geben Sie das Ergebnis ein:");
        int guess = scan.nextInt();

        scan.close();

        if (guess == sum) {
            System.out.println("Das Ergebnis ist korrekt!");
        } else {
            System.out.println("Das Ergebnis ist falsch!");
        }
    }
}
```

(03) Username and Password

Schreiben Sie ein Programm, das einen Benutzer separat nach seinem Vorund Nachnamen fragt und diese einliest. Danach soll ein Benutzername nach folgendem Muster erzeugt werden:

$$FL_1L_2L_3L_4L_5D_1D_2D_3$$

wobei

- F dem ersten Buchstaben des Vornamens entspricht.

- L_i dem i-ten Buchstaben des Nachnamens entspricht.
- $D_1D_2D_3$ einer zufälligen Zahl zwischen 000 und 999 entspricht.

Falls der eingegebene Nachname kürzer als 5 Zeichen sein sollte, werden entsprechend weniger Zeichen verwendet.

Zusätzlich erzeugen Sie ein zufälliges Passwort für den Benutzer. Das Passwort soll mit einer 7 oder 8 oder 9 starten, gefolgt von 5 zufälligen ganzen Zahlen von 0 bis 9, gefolgt von einem Bindestrich -, gefolgt von drei zufälligen Grossbuchstaben.

Geben Sie Benutzernamen und Passwort aus.

Tipp: Die Zahlen 65 bis 90 repräsentieren die Grossbuchstaben A bis Z in Unicode und Sie können bspw. die ganze Zahl 77 folgendermassen in einen Buchstaben umwandeln:

```
char d1 = (char) 77;
```

```
import java.util.Scanner;
import java.util.Random;

public class UsernameAndPassword {
    public static void main(String[] args) {

        Scanner scan = new Scanner(System.in);

        System.out.println("Geben Sie Ihren Vornamen ein:");
        String firstName = scan.nextLine();
        System.out.println("Geben Sie Ihren Nachnamen ein:");
        String lastName = scan.nextLine();
        scan.close();

        int lastNameLength = lastName.length();
        final int MAX_LETTERS_OF_LASTNAME = 5;

        String firstLetter = firstName.substring(0, 1);
        // Math.min(a, b) sorgt dafür, dass auch Nachnamen mit weniger
        // als MAX_LETTERS_OF_LASTNAME Buchstaben funktionieren
        String lastNameArray = lastName.substring(0, Math.min(MAX_LETTERS_OF_LASTNAME,
                                                    lastNameLength));

        Random random = new Random();
        int randomNumber = random.nextInt(1000);
        // String.format("%03d", randomNumber) fügt führende Nullen hinzu,
        // falls randomNumber < 100
        String username = firstLetter.toUpperCase() + lastNameArray.toUpperCase() +
                                        String.format("%03d", randomNumber);

        System.out.println("Der Benutzername lautet: " + username);

        final int UNICODE_LOWER_BOUND = 65;
        final int UNICODE_UPPER_BOUND = 91;
        // ermöglicht die zufällige Ausgabe des ganzen Zahlenteils des Passworts
        int randomPasswordNumber = random.nextInt(700000, 1000000);
        char randomPasswordLetter1 = (char) random.nextInt(UNICODE_LOWER_BOUND,

UNICODE_UPPER_BOUND);
        char randomPasswordLetter2 = (char) random.nextInt(UNICODE_LOWER_BOUND,

UNICODE_UPPER_BOUND);
        char randomPasswordLetter3 = (char) random.nextInt(UNICODE_LOWER_BOUND,

UNICODE_UPPER_BOUND);

        String password = randomPasswordNumber + "-" + randomPasswordLetter1 +
                                                randomPasswordLetter2 + randomPasswordLetter3;

        System.out.println("Das Passwort lautet: " + password);
    }
}
```

P1 - Serie 03

(1) Klasse `Thermometer`

Programmieren Sie eine Klasse `Thermometer`, welche einen einfachen Fieberthermometer modelliert. Die Klasse soll eine Temperatur in Celsius als einzige Instanzvariable speichern. Der Konstruktor soll diese Instanzvariable standardmässig auf 37.0 Grad setzen. Schreiben Sie eine Methode `increase`, welche die Temperatur um 0.1 Grad erhöht und einen Getter für die Temperatur. Zudem definieren Sie eine Methode `reset`, welche die Temperatur wieder auf 37.0 zurücksetzt.

Schreiben Sie eine zweite Klasse `ThermometerTest`, in der Sie zwei Objekte vom Typ `Thermometer` instanziieren und deren Methoden ausführlich testen.

Klasse `Thermometer`

```java
public class Thermometer {

    private double temperature;

    public Thermometer() {
        this.temperature = 37.0;
    }

    public void increase() {
        this.temperature += 0.1;
    }

    public double getTemperature() {
        return this.temperature;
    }

    public void reset() {
        this.temperature = 37.0;
    }
}
```

Aufgabe 2: `Car`-Klasse

(2) Klasse `Car`

Programmieren Sie eine Klasse `Car`, welche die Marke, das Modell und den Jahrgang des Fahrzeuges modelliert. Der Konstruktor soll diese drei Instanzvariablen gemäss Parameterübergabe initialisieren – zudem schreiben Sie Getter und Setter für alle Instanzvariablen und eine `toString` Methode für eine einzeilige Repräsentation von `Car` Objekten. Schliesslich definieren Sie eine Methode `isAntique`, welche einen boolean zurückgibt, der anzeigt ob das Auto aktuell älter ist als 45 Jahre.

In einer zweiten Klasse `Garage` instanziieren Sie drei `Car` Objekte und testen alle programmierten Methoden.

```java
import java.time.LocalDate;

public class Car {

    private String brand;
    private String model;
    private int year;
    private boolean isAntique;
    private final int ANTIQUE_AGE = 45;

    public Car(String brand, String model, int year) {
        this.brand = brand;
        this.model = model;
        this.year = year;
        this.isAntique = checkIfAntique();
    }

    public String getBrand() {
        return this.brand;
    }

    public String getModel() {
        return this.model;
    }

    public int getYear() {
        return this.year;
    }

    public boolean getIsAntique() {
```

```
        return this.isAntique;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

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

    public void setYear(int year) {
        this.year = year;
        this.isAntique = checkIfAntique();
    }

    private boolean checkIfAntique() {
        int currentYear = LocalDate.now().getYear();
        if (currentYear - this.year > ANTIQUE_AGE) {
            return true;
        } else {
            return false;
        }
    }

    public boolean isAntique() {
        return this.isAntique;
    }

    public String toString() {
        return this.brand + " " + this.model + " " + this.year + " Is Antique: " + this.isAntique;
    }
}
```

(3) `Cargo` und `Box`-Klassen

Schreiben Sie eine Klasse `Cargo`, welche ein Stückgut mit Länge, Breite, Höhe und einem Namen modelliert (z.B. 30, 44, 65, "Kaffeemaschine"). Schreiben Sie einen Konstruktor, Getter und Setter für alle Instanzvariablen und eine Methode `toString`.

Schreiben Sie eine Klasse `Box`, die Instanzvariablen für die Länge, Breite und Höhe einer Box enthält. Zusätzlich enthält die Klasse `Box` eine Instanzvariable `full` vom Typ boolean, die angibt, ob die Box gefüllt ist oder nicht, sowie eine Instanzvariable `cargo` vom Typ `Cargo`. Der Konstruktor setzt die Länge, Breite und Höhe einer Box gemäss Parametern – neu instanziierte `Box` Objekte sollen standardmässig leer sein. Definieren Sie einen zweiten Konstruktor ohne Parameter, der eine Standard-Box mit Länge, Breite und Höhe 1 generiert. Zusätzlich definieren Sie eine Methode `getCapacity`, die das Volumen der Box berechnet und zurückgibt.

Schliesslich schreiben Sie eine Methode `addCargo`, welche ein Objekt vom Typ `Cargo` als Parameter entgegennimmt. Falls dieses Stückgut gemäss Länge, Breite und Höhe in die Box passt, passen Sie die Variable `full` und die Instanzvariable `cargo` an und geben `true` zurück (andernfalls `false`).

Testen Sie die Klasse `Box`, indem Sie in einer weiteren Klasse `BoxTest` drei `Box` Objekte instanziieren, manipulieren und ausgeben.

Mögliche Lösung

Klasse `Cargo`

```
public class Cargo {

    private String name;
    private int length;
    private int width;
    private int height;

    public Cargo(String name, int length, int width, int height) {
        this.name = name;
        this.length = length;
        this.width = width;
        this.height = height;
    }

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

    public int getLength() {
        return this.length;
    }

    public int getWidth() {
        return this.width;
    }
```

```
    public int getHeight() {
        return this.height;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setLength(int length) {
        if (length > 0) {
            this.length = length;
        }
    }

    public void setWidth(int width) {
        if (width > 0) {
            this.width = width;
        }
    }

    public void setHeight(int height) {
        if (height > 0) {
            this.height = height;
        }
    }

    public String toString() {
        return "Cargo: \tName: " + this.name + " Dimensions: " + this.length + "x" + this.width + "x" + this.height;
    }
}
```

Klasse `Box`

```
import java.util.ArrayList;
import java.util.Collections;


public class Box {

    private Cargo cargo;
    private boolean isFull;
    private int length;
    private int width;
    private int height;

    public Box(int length, int width, int height) {
        this.length = length;
        this.width = width;
        this.height = height;
        this.isFull = false;
    }

    public Box() {
        this(1, 1, 1); // refers to the other constructor
    }

    public int getCapacity() {
        return this.length * this.width * this.height;
    }

                        //... all other getters

    public void setLength(int length) {
        if (length > 0) {
            this.length = length;
        }
    }

                        //... all other setters

    public boolean addCargo(Cargo cargo) {
        if (this.isFull) {
            System.out.println("Box already full");
            return false;

        /*
         * System welches sicherstellt, dass das Cargo schlau eingepackt wird. Box kann ja gedreht werden
         * ArrayList<Integer> erstellt einen Array
         */
        } else {
```

```
            ArrayList<Integer> cargoDimensions = new ArrayList<>();
                cargoDimensions.add(cargo.getLength());
                cargoDimensions.add(cargo.getWidth());
                cargoDimensions.add(cargo.getHeight());
            ArrayList<Integer> boxDimensions = new ArrayList<>();
                boxDimensions.add(this.length);
                boxDimensions.add(this.width);
                boxDimensions.add(this.height);

            Collections.sort(cargoDimensions);
            Collections.sort(boxDimensions);

            boolean fits = true;
            for (int i = 0; i < 3; i++) {
                if (cargoDimensions.get(i) > boxDimensions.get(i)) {
                    fits = false;
                    break;
                }
            }

            if (fits) {
                this.cargo = cargo;
                this.isFull = true;
                System.out.println("Cargo placed in Box!");
                return true;
            } else {
                System.out.println("Box dimensions are too small for that cargo");
                return false;
            }
        }
    }

    public void removeCargo() {
        this.cargo = null;
        this.isFull = false;
    }

    public String toString() {
        return "Box: \tDimensions: " + this.length + "x" + this.width + "x" + this.height + " Capacity: " + this.getCapacity()
+ " " + this.cargo;
    }
}
```

(4) Erweiterung der Klasse `Book`

Auf ILIAS (Übungen → Serie 3) finden Sie eine Datei `Book.java`. Ihre Aufgabe ist es die darin implementierte Klasse `Book` wie folgt zu erweitern:

1. Schreiben Sie einen Konstruktor sowie Getter und Setter für alle Instanzvariablen.
2. Implementieren Sie die Methode `age`, welche das Alter eines Buches (Anzahl Tage seit Erscheinungsdatum) berechnet und zurückgibt.
3. Implementieren Sie die Methode `toString`, die alle Informationen eines `Book`-Objekts als String zurückgibt. Beispiel: 123, Die Blechtrommel, Günter Grass, 1.1.1959
4. Vervollständigen Sie die Methode `input`, welche die Werte für `id`, `title`, `author` und `dateOfPublication` von der Kommandozeile vom Benutzer einliest und im jeweiligen `Book`-Objekt abspeichert. Ungültige Eingaben müssen Sie nicht abfangen.

Testen Sie sämtliche Methoden der Klasse `Book` in einer zusätzlichen Klasse `BookShelf`.

```
import java.util.Date;
import java.util.Scanner;
import java.text.*;

public class Book
{
        private int id;
        private String title;
        private String author;
        private Date dateOfPublication;

        public static final String DATE_FORMAT = "dd.MM.yyyy";

        public Book(int id, String title, String author, String dateOfPublication) {
                this.id = id;
                this.title = title;
                this.author = author;
                this.dateOfPublication = stringToDate(dateOfPublication);
        }

        public Book() {

        }
```

```
        /** Returns the age of the book in days since publication */
        public int age() {
                return (int) ((new Date().getTime() - dateOfPublication.getTime()) / (1000 * 60 * 60 * 24));
        }

        /** Returns a String representation of the book */
        public String toString() {
                return id + ", " + title + ", " + author + ", " + dateToString(dateOfPublication);
        }

        /** Reads all book data from user input */
        public void input() {
                Scanner scanner = new Scanner(System.in);
                System.out.print("ID: ");
                id = scanner.nextInt();
                scanner.nextLine(); // consume newline
                System.out.print("Title: ");
                title = scanner.nextLine();
                System.out.print("Author: ");
                author = scanner.nextLine();
                System.out.print("Date of publication: ");
                dateOfPublication = stringToDate(scanner.nextLine());
                scanner.close();
        }

        //--- Get-/Set-methods ---

        //--- helper methods -- DO NOT CHANGE -----------------------------------
        /** Converts the Date object d into a String object */
        public static String dateToString( Date d )
        {
                SimpleDateFormat fmt = new SimpleDateFormat( DATE_FORMAT );
                return fmt.format( d );
        }

        /** Converts the String object s into a Date object */
        public static Date stringToDate( String s )
        {
                Date r = null;
                try {
                        SimpleDateFormat fmt = new SimpleDateFormat( DATE_FORMAT );
                        r = fmt.parse( s );
                } catch ( ParseException e ){
                        System.err.println( e );
                        System.exit(1);
                }
                return r;
        }
}
```

P1 - Serie 04

Lukas Batschelet (16-499-733)

(01) Schaltjahre

Die gregorianische Schalttagsregelung besteht aus folgenden Regeln:

- Die durch 4 ganzzahlig teilbaren Jahre sind, abgesehen von den folgenden Ausnahmen, Schaltjahre.
- Säkularjahre, also die Jahre, die ein Jahrhundert abschliessen (z.B. 1800 oder 1900), sind *keine* Schaltjahre, es sei denn, dass das Säkularjahr auch durch 400 ganzzahlig teilbar ist (zum Beispiel das Jahr 2000).In diesem Fall ist auch das Säkularjahr ein Schaltjahr.

Schreiben Sie eine Methode, die für ein als Parameter übergebenes Jahr `year` überprüft, ob dieses ein Schaltjahr ist oder nicht. Erzeugen Sie eine entsprechen Ausgabe. Geben Sie eine Fehlermeldung aus, falls das Jahr kleiner ist als 1582 (das Jahr in dem der gregorianische Kalender eingeführt wurde).

Mögliche Lösung

```java
public boolean isLeapYear(int year) {

final int GREGORIAN_START_YEAR = 1582;

        if (year >= GREGORIAN_START_YEAR) {

                if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0) {
                        System.out.println("Das Jahr " + year + " ist ein Schaltjahr!");
                        return true;
                } else {
                        System.out.println("Das Jahr " + year + " ist kein Schaltjahr!");
                        return false;
                }

        } else {
                System.out.println("Fehler: Gregorianischer Kalender noch nicht eingeführt");
                return false;
        }
}
```

(02) Verschachtelte Vergleiche

Es sei

(a) `int num1 = 5, num2 = 4;`
(b) `int num1 = 8, num2 = 4;`
(c) `int num1 = 10, num2 = 11;`
(d) `int num1 = 10, num2 = 12;`
(e) `int num1 = 10, num2 = 13;`

Was ist die Ausgabe des folgenden Code Fragments für diese fünf Fälle?

```
if (num1 >= num2) {
        System.out.print("1 ");
        System.out.print("2 ");
        num1 = num2 - 1;
}
System.out.print("3 ");
if ((num1 + 1) >= num2)
        System.out.print("4 ");
else
        if ((num1 + 2) >= num2) {
                System.out.print("5 ");
                System.out.print("6 ");
        } else
                System.out.print("7 ");
System.out.print("8 ");
```

Lösung

```
(a) 1 2 3 4 8
(b) 1 2 3 4 8
(c) 3 4 8
(d) 3 5 6 8
(e) 3 7 8
```

(03) Methode `isIsoceles(int a, int b, int c)`

Schreiben Sie eine Methode `isIsosceles`, die drei ganze Zahlen als Parameter entgegennimmt (die Längen der drei Seiten eines Dreiecks). Die Methode gibt `true` zurück, falls das Dreieck gleichschenklig aber *nicht* gleichseitig ist (also nur dann, wenn *genau zwei* Seiten gleich lang sind).

Lösung

```
public boolean isIsoceles(int a, int b, int c) {
    return (a == b && a != c) || (a == c && a != b) || (b == c && b != a);
}
```

(04) Methode `countA(String name)`

Schreiben Sie eine Methode `countA`, die in einem als Parameter übergebenem `String` `name` die Anzahl der Zeichen `'a'` und `'A'` zählt und diese Anzahl zurückgibt.

Bemerkung: Lösen mit einer `for`-Schleife

Mögliche Lösung

```
public int countA(String name) {
        int count = 0;
        for (int i = 0; i < name.length(); i++) {
                char c = name.charAt(i);
                if (c == 'a' || c == 'A') {
                        count++;
                }
        }
        return count;
}
```

(05) Ausgabe einer `do`-Schleife

Welche Ausgabe erzeugt folgendes Code-Fragment?

```
int low = 0, high = 5;
do {
        System.out.println(low);
        low++;
} while (low < high);
```

Lösung

```
0
1
2
3
4
```

(06) Ausgabe einer verschachtelten `for`-Schleife

Welche Ausgabe erzeugt folgendes Code-Fragment?

```
for (int i = 1; i <= 3; i++) {
        for (int j = i; j > 0; j--)
                System.out.println(i + " " + j);
}
```

```
1 1
2 2
2 1
3 3
3 2
3 1
```

(07) Umwandeln einer `for`-Schleife in eine `while`-Schleife

Schreiben Sie das folgende Code-Fragment mit Hilfe einer `while`-Schleife um.

```java
int value = 0;
for (int num = 10; num <= 40; num+=10) {
        value += num;
}
System.out.println(value);
```

```java
int value = 0, num = 10;
while (num <= 40) {
        value += num;
        num += 10;
}
System.out.println(value);
```

Implementationsaufgaben

(01) Klasse `Coin`

Programmieren Sie eine Klasse `Coin`, die eine Münze repräsentieren soll. Eine Münze zeigt entweder Kopf oder Zahl an (speichern Sie diese Information als Variable vom Typ `boolean`).

Der Konstruktor der Klasse `Coin` soll eine Münze zufällig instanziieren – also mit 50% Wahrscheinlichkeit soll eine neu instanziierte Münze Kopf zeigen (und sonst Zahl). Programmieren Sie hierzu eine Methode `flip`, die den Münzwurf simuliert. Ergänzen Sie Ihre Klasse mit einer Methode `equals` und einer Methode `toString`. Zwei Münzen sind gleich, wenn beide die gleiche Seite anzeigen. Die Methode `toString` soll Kopf oder Zahl zurückgeben – verwenden Sie hierzu einen *Conditional Operator*. Programmieren Sie ein Programm `CoinRace`: Dieses Programm soll zwei `Coin` Objekte solange werfen, bis einer der beiden Münzen dreimal hintereinander Kopf angezeigt hat (Zählen Sie die Runden mit). Diese Münze gewinnt das Spiel.

Geben Sie das Resultat aller Münzwürfe aus und am Schluss geben Sie an, welche der beiden Münzen gewonnen hat aus. Also bspw:

```
Runde 1: Zahl Zahl
Runde 2: Kopf Zahl
Runde 3: Kopf Kopf
Runde 4: Kopf Kopf
Münze 1 gewinnt
```

Hinweis: Es kann auch Unentschieden geben!

```java
public class Coin {

    public boolean isHeads;

    public Coin() {
        this.isHeads = Math.random() < 0.5;
    }

    public void flip() {
        this.isHeads = Math.random() < 0.5;
    }

    public boolean equals(Coin other) {
        return this.isHeads == other.isHeads;
    }

    public String toString() {
        return this.isHeads ? "Kopf" : "Zahl";
    }
}
```

```java
public class CoinRace {

    public static void main(String[] args) {

                Coin coin1 = new Coin();
                Coin coin2 = new Coin();
                int round = 1;
                int coin1Heads = 0;
                int coin2Heads = 0;

                while (coin1Heads < 3 && coin2Heads < 3) {
                        coin1.flip();
                        coin2.flip();
                        System.out.println("Runde " + round + ": " + coin1 + " " + coin2);
                        if (coin1.isHeads) {
                                coin1Heads++;
                        } else {
```

```
                        coin1Heads = 0;
                    }
                    if (coin2.isHeads) {
                        coin2Heads++;
                    } else {
                        coin2Heads = 0;
                    }
                    round++;
                }
                if (coin1Heads == 3) {
                    System.out.println("\nMünze 1 gewinnt nach " + (round - 1) + " Runden");
                } else {
                    System.out.println("\nMünze 2 gewinnt nach " + (round - 1) + " Runden");
                }
    }
}
```

(02) Würfelsimulation

Übernehmen Sie die Klasse `Dice` aus dem Skript. Schreiben Sie dann eine Klasse `PairOfDice`, welche aus zwei `Dice` Objekten besteht. Definieren Sie Methoden zum Setzen und Auslesen der Punkte der einzelnen Würfel, eine Methode um beide Würfel zu werfen und eine Methode, welche die aktuelle Summe der Würfel zurückgibt.

(03) Spiel *Pig*

Implemetieren Sie das Spiel *Pig*. In diesem Spiel gewinnt derjenige Spieler, der zuerst die Punktesumme 100 gesammelt hat. Innerhalb eines Spielzugs wirft der aktuelle Spieler so oft er möchte je zwei Würfel (verwenden Sie also die Klasse `PairOfDice` aus Aufgabe 2). Dabei werden alle Punkte zusammengezählt, bis entweder eine EINS gewürfelt wird, dann ist der Zug zu Ende und alle Punkte dieses Zugs sind verloren und der andere Spieler ist an der Reihe, oder der Spieler die Würfel weiterreicht und seinen Zugfreiwillig beendet. Nur in diesem Fall werden die gewürfelten Augen aufsummiert und dem Konto des Spielers gutgeschrieben. Sollte der Spieler zwei EINSEN gleichzeitig würfeln, verliert er sämtliche bis zu diesem Zeitpunkt gesammelten Punkte auf seinem Konto und sein Zug ist ebenfalls zu Ende.

Nach jedem Wurf muss ein Spieler also entscheiden, ob er weiter würfeln will (und so riskiert, dass er die Punkte des aktuellen Zuges oder sogar sämtliche Punkte verliert) oder ob er die Würfel an den Gegenspieler abgeben möchte (und so riskiert, dass der Gegenspieler gewinnt).

Implementieren Sie das Spiel in den Modi Ein- und Zweispieler: Im Einspielermodus spielt der Spieler gegen den Rechner. Die Strategie des Rechners ist es, solange die aktuelle Summe kleiner ist als 20, weiterzuwürfeln. Bei einer Summe grösser-gleich 20 gibt er den Zug also immer freiwillig ab.

P1 - Serie 05

Lukas Batschelet (16-499-733)

Implementationsaufgaben

(01) Vier Gewinnt

1. Sie sollen ein “Vier gewinnt“ Spiel programmieren, bei dem man in der Konsole gegen eine zweite Person spielen kann. Laden Sie von ILIAS die Dateien `VierGewinnt.java`, `Player.java` und `Token.java` herunter. Die Klasse `VierGewinnt` enthält bereits Methoden `play()` (definiert den Spielablauf), `main` (startet das Spiel) und `displayField()` (graphische Darstellung des Spielfelds):

```
1
Player X choose a column between 1 and 7: 2
+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
| X |   |   |   |   |   |   |
+---+---+---+---+---+---+---+
| O | X | O |   |   |   |   |
+---+---+---+---+---+---+---+
| X | O | X |   |   |   | X |
+---+---+---+---+---+---+---+
| O | O | O | X | X |   | O |
+---+---+---+---+---+---+---+
  1   2   3   4   5   6   7
Player X wins!
```

2. Um das Spiel zum Laufen zu bekommen, müssen Sie in der Klasse `VierGewinnt` die folgenden Methoden implementieren (die anderen gegebenen Methoden dürfen Sie nicht verändern):
   1. `insertToken`: Der übergebene Stein (Token-Objekt) soll in die gewählte Spalte (column) des Spielfelds (Array[][] board) gefüllt werden. Falls eine nicht existierende oder bereits bis oben gefüllte Spalte gewählt wurde, soll das Programm mit einer Fehlermeldung abbrechen. Verwenden Sie dazu `System.exit(1)`.
   2. `isBoardFull`: gibt genau dann true zurück, wenn alle Felder durch einen Stein besetzt sind.
   3. `checkVierGewinnt`: überprüft – ausgehend vom durch col und row gegebenen Feld – ob es in einer der vier Richtungen (d.h. –,|,/,\) mindestens vier gleiche Steine gibt. In diesem Fall wird true zurückgegeben, andernfalls false. Tipp: Schreiben Sie für jede der vier Richtungen eine Hilfsmethode.

```
public int insertToken(int column, Token tok) {
         // check if the column is valid
         if (column > VierGewinnt.COLS || column < 0) {
                        System.out.println("The entered column number does not exist");
                        System.exit(1);
         }
         // check if the chosen column is full
         if (board[column][VierGewinnt.ROWS - 1] != Token.empty) {
                        System.out.println("This column is already full");
                        System.exit(1);
         }
         // search first free row in column and place tok
         int row = 0;
         while (row < VierGewinnt.ROWS) {
                        if (board[column][row] == Token.empty) {
                                  break; // empty place found -> leave while loop
                        }
                        row++;
         }
         board[column][row] = tok;
         return row;
}
```

```
private boolean isBoardFull() {
        for(int col = 0; col < VierGewinnt.COLS; col++) {
                if(!isColumnFull(col))
                        return false;
        }
        return true;
}

private boolean isColumnFull(int column) {
        return this.board[column][ROWS - 1] != Token.empty;
}
```

```java
public boolean checkVierGewinnt(int col, int row) {
        Token tokToCheck = board[col][row];
        // check straight down
        if (row >= 3) {
                if (checkDown(col, row, tokToCheck)) {
                        return true;
                }
        }
        // check horizontal
        if (checkHorizontal(col, row, tokToCheck)) {
                return true;
        }
        // check diagonal left down to right up
        if (checkDiagonalLeftRight(col, row, tokToCheck)) {
                return true;
        }
        // check diagonal right down to left up
        if (checkDiagonalRightLeft(col, row, tokToCheck)) {
                return true;
        }
        // if none of the checks returned true, return false
        return false;
}
```

Rechnen mit Matrizen

(02) Rechnen mit Matrizen

1. Schreiben Sie in einer Klasse `MatrixOperations` eine statische Methode `readMatrix`, welche die Daten einer Matrix aus einer Datei einliest. Eine Matrix-Datei sei dabei folgendermassen formatiert:

   ```
   1 2 3
   4 5 6
   ```

   Speichern Sie die eingelesenen Daten in einem 2-dimensionalen Array.
2. Schreiben Sie in der Klasse `MatrixOperations` eine statische Methode `transpose` [1], welche eine $n \times n$ Matrix A als Parameter erhält und A^T (die transponierte Matrix A zurückgibt. Falls die übergebene Matrix nicht quadratisch sein sollte, erzeugen Sie eine Fehlermeldung und geben `null` zurück. Beim Transponieren einer Matrix spiegelt man einen Matrixeintrag a_{ij} an der Diagonalen von A. Einfach gesagt, aus a_{ij} wird a_{ji}.

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \text{ und } A^T = \begin{pmatrix} a_{00} & a_{10} & a_{20} \\ a_{01} & a_{11} & a_{21} \\ a_{02} & a_{12} & a_{22} \end{pmatrix}$$

Ihre Methode sollte alle möglichen Werte von n berücksichtigen.

3. Schreiben Sie in der Klasse `MatrixOperations` eine statische Methode `product` [2], welche eine $n \times m$ Matrix A und eine $m \times l$ Matrix B als Parameter entgegennimmt und das Produkt $C = AB$ zurückgibt. Die Dimension von C ist $n \times l$ und die Einträge c_{ij} berechnen sich wie folgt:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

Ihre Methode sollte alle möglichen Werte von m, n und l berücksichtigen. Falls die Anzahl Spalten von A nicht mit der Anzahl Zeilen von B übereinstimmen sollte, erzeugen Sie eine Fehlermeldung und geben Sie `null` zurück.

Beispielechung:

$$Input: \ A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}, B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} Output: C = \begin{pmatrix} 9 & 12 & 15 \\ 19 & 26 & 33 \\ 29 & 40 & 51 \end{pmatrix}$$

4. Erstellen Sie eine Klasse `MatrixTest.java` in der Sie die Methoden `readMatrix`, `transpose` und `product` mit geeigneten Beispielen testen. Achten Sie darauf bei der Methode `transpose` und `product` auch den Fehlerfall zu testen.

```java
public static int[][] readMatrix(String fileName) throws FileNotFoundException {

        ArrayList<String[]> lines = new ArrayList<String[]>();
        Scanner fileReader = new Scanner(new File(fileName));
        while (fileReader.hasNext()){
                String line = fileReader.nextLine();
                lines.add(line.split(" "));
        }

        int[][] matrix = new int[lines.size()][lines.get(0).length];

        for(int i = 0; i < matrix.length; i++) {
                for(int j = 0; j < matrix[0].length; j++) {
                        matrix[i][j] = Integer.parseInt(lines.get(i)[j]);
                }
```

```
        }
        return matrix;
}
```

```
public static int[][] transpose(int[][] matrix){
        if(matrix.length != matrix[0].length) {
                System.out.println("Die gegebene Matrix ist nicht Quadratisch");
                return null;
        }
        int[][] transpose = new int[matrix.length][matrix.length];
        for (int row = 0; row < matrix.length; row++) {
                for (int col = 0; col < matrix[0].length; col++) {
                        transpose[col][row] = matrix[row][col];
                }
        }
        return transpose;
}
```

```
public static int[][] product (int[][] a, int[][] b) {
                if (a[0].length != b.length) {
                        System.out.println("Dimensionen passen nicht!");
                        return null;
                }
                int[][] c = new int [a.length][b[0].length];
                int n = a[0].length;
                for (int i = 0; i < c.length; i++) {
                        for (int k = 0; k < c[0].length; k++) {
                                int sum = 0;
                                for (int j = 0; j < n; j++) {
                                        sum += a[i][j] * b[j][k];
                                }
                                c[i][k] = sum;
                        }
                }
                return c;
        }
}
```

---

1. https://de.wikipedia.org/wiki/Transponierte_Matrix ↩
2. https://de.wikipedia.org/wiki/Matrizenmultiplikation ↩

P1 - Serie 06

Implementationsaufgaben

(01) Buchbestellung Klasse `Order` erstellen:

- Laden Sie die Datei `Book.java` von ILIAS (→ Serie 6 Vorlagen → Aufgabe 1). Verwenden Sie nicht die Datei aus Serie 3.
- Erstellen Sie eine Klasse `Order` für Buchbestellungen, bestehend aus einer id, Kundennamen (`customerName`), Kundenadresse (`customerAddress`) und beliebig vielen `Book` -Objekten.
- Die Klasse soll `toString()` und `addBook(...)` Methoden enthalten.
- Implementieren Sie einen Konstruktor `Order()` , der die `id` automatisch vergibt (1 für das erste Objekt, 2 für das zweite, usw.), mittels einer `static` -Variablen.
- Testen Sie die Klasse `Order` mit der bereitgestellten Klasse `Test` von ILIAS (Übungen → Serie 6 → Serie 6 Vorlagen → Aufgabe 1). Die Ausgabe sollte exakt wie im Beispiel aussehen.

Beispiel Ausgabe:

```
Order id: 1, Customer: Sophie Muster, Mittelstrasse 10, 3011 Bern
1, Homo Faber, Max Frisch, 01.01.1957
2, Harry Potter, J.K. Rowling, 25.07.2000
...
```

Hinweise:

- Implementieren Sie nur die tatsächlich verwendeten Getter und Setter.
- Die Klasse `Test` darf nicht verändert werden.

```java
public class Order {
    private static int idNumber = 1;

    private int id = idNumber;
    private String customerName;
    private String customerAdress;
    private ArrayList<Book> books = new ArrayList<Book>();

    public Order() {
        this.id = idNumber;
        Order.idNumber += 1;
    }

    public void addBook(Book book) {
        this.books.add(book);
    }

    public String toString() {
        String string = "Order id: " + this.getId() + " Customer: " + this.getCustomerName() +
                        ", " + this.getCustomerAdress() + "\n";
        for(Book b : this.books) {
            string += "" + b.toString() + "\n";
        }
        return string;
    }
}
```

```java
public String toString() {
    String string = "Order id: " + this.getId() + " Customer: " + this.getCustomerName() +
                    ", " + this.getCustomerAdress() + "\n";
    for(Book b : books) {
        string += "" + b.toString() + "\n";
    }
    return string;
}
```

(02) Implementierung der Klasse `Store`

- Laden Sie `Store.java` und `Book.java` (nicht dieselbe Datei wie in Teilaufgabe 1) von ILIAS herunter.
- Das Programm `Store` verfügt über ein Menü zur Erfassung neuer Bestellungen (Bücher, DVDs, CDs).
- Ihre Aufgabe ist es, dafür zu sorgen, dass das Programm `Store` einwandfrei funktioniert, ohne `Store` selbst zu verändern.
- Sie müssen folgende Klassen und Schnittstellen programmieren:
  (a) Schnittstelle `IArticle` mit den Methoden `getId()`, `getPrice()` und `getDescription()`. Passen Sie die Klasse `Book` so an, dass sie `IArticle` implementiert.
  (b) Erstellen Sie die Klassen `DVD` und `CD`, die beide `IArticle` implementieren. `CD` soll einen Interpreten statt eines Autors haben, DVD kein Feld `author`.
  (c) Passen Sie die Klasse aus Aufgabe 1 an. Die erforderlichen Methoden für `Order` können Sie aus der Klasse `Store` entnehmen.

Beispiel Menüausgabe `Store`:

```
============================================================
| 1. Create a new order 2. Show all registered articles |
| 3. Show all orders 9. Exit |
============================================================
What do you want to do? 1
1 (Book) Die Blechtrommel, by Guenter Grass, 1959, 29 CHF
...
Enter the customer's name: Susi Meier
Enter the customer's address: Mittelstrasse 10, 3011 Bern
```

Insbesondere muss Order eine Methode `getOrderedArticles()` besitzen. Definieren sie dessen Rückgabetyp als `Iterable<IArticle>`.
(d) Zeichnen Sie ein UML-Klassendiagramm aller involvierten Klassen und Schnittstellen.

Interface

```java
public interface IArticle {
        public int getPrice();
        public int getId();
        public String getDescription();
}
```

Bsp. Book

```java
public class Book implements IArticle {
    private int id;
    private String title;
    private String author;
    private int year;
    private int price; // CHF

    /** constructor */
    public Book(int id, String title, String author, int year, int price) {
        this.id = id;
        this.title = title;
        this.author = author;
        this.year = year;
        this.price = price;
    }

    public String getDescription() {
        return this.id + " (Book) " + this.title + ", by " + this.author +
               ", " + this.year + ", " + this.price + " CHF";
    }

    public int getPrice() {
        return this.price;
    }

    public int getId() {
        return this.id;
    }
}
```

Klasse Order

```java
public class Order {
    private static int idNumber = 1;
    private int id = idNumber;
    private String customerName;
    private String customerAddress;
    private ArrayList<IArticle> articles = new ArrayList<IArticle>();

    public Order() {
        this.id = idNumber;
```

```
        Order.idNumber += 1;
    }

    public void add(IArticle a) {
        this.articles.add(a);
    }

    public String getTotalPrice() {
        int sum = 0;
        for(IArticle a : this.articles) {
            sum += a.getPrice();
        }
        return String.valueOf(sum);
    }

    public Iterable<IArticle> getOrderedArticles() {
        return this.articles;
    }

    public String toString() {
    String string = "Order id: " + this.id + " Customer: " + this.customerName + ", " +
                    this.customerAddress + "\n";
    for (IArticle a : this.articles) {
        string += a.getDescription() + "\n";
    }
    return string;
        }
}
```

(03) Preisberechnungssystem einer Möbelfirma

Sie sollen für eine Firma, welche Möbel herstellt, ein System entwickeln. Die Firma will sehen, ob sich dieses System bewährt, deshalb soll zunächst nur die Preisberechnung für die Tische implementiert werden. Falls die Firmenleitung zufrieden ist, sollen mehr Möbelstücke und mehr Funktionalitäten integriert werden. Implementieren Sie für dieses System `Furniture.java`, `Material.java` und `Table.java` nach dem folgenden UML-Diagramm (FurnitureTest.java ist auf Ilias verfügbar):

Anweisungen:
(a) Die Klassen sollen nach diesem UML-Diagramm implementiert werden ohne zusätzliche Variablen oder Methoden zu verwenden.
(b) Vergeben Sie jedem Objekt in der Aufzählung einen anderen Preis pro Quadratmeter (speichern Sie diese Information in einer Instanzvariablen)
(c) Erstellen Sie einen passenden Konstruktor für `Furniture` und `Table`. Der Konstruktor von `Table` soll dabei den Konstruktor von `Furniture` verwenden.
(d) Die Variable `pricePerHour` gibt die Kosten pro Stunde an für die Anfertigung des Möbelstückes. `workedHours` gibt die Anzahl Stunden an, welche nötig waren, um das Möbelstück fertigzustellen. Die Methode `calculateEffort()` soll nun den Aufwand für die Herstellung eines Möbelstückes berechnen.
(e) Da in der Aufwandsberechnung noch nicht der Materialpreis inbegriffen ist, soll in der Methode `totalPrice()` zunächst mithilfe der Methode aus `Furniture` den Preis für den Aufwand berechnet werden. Danach wird der Materialpreis berechnet (gegeben durch multipliziert mit dem Quadratmeterpreis des Materials) und zum Aufwand dazu addiert.

Klasse `Furniture`

```
import java.text.DecimalFormat;

public class Furniture {
    public Material material;
    protected double pricePerHour;
    protected double workedHours;

    DecimalFormat form = new DecimalFormat("0.## CHF");

    public Furniture(Material material, double pricePerHour, double workedHours) {
        this.material = material;
        this.pricePerHour = pricePerHour;
        this.workedHours = workedHours;
    }

    public double calculateEffort() {
        return this.pricePerHour * this.workedHours;
    }
}
```

Enum `Material`

```
public enum Material {
    Esche(2.5),
    Buche(1.8),
    Eiche(5.5);

    private double materialCost;

    private Material(double materialCost) {
        this.materialCost = materialCost;
    }

    public double materialCost() {
        return this.materialCost;
    }
}
```

Klasse `Table`

```
public class Table extends Furniture {
    private double area;

    public Table(Material mat, double pricePerHour, double workedHours, double area) {
        super(mat, pricePerHour, workedHours);
        this.area = area;
    }

    public double totalPrice() {
        return this.calculateEffort() + this.material.materialCost() * this.area;
    }
}
```

P1 - Serie 07

Lukas Batschelet (16-499-733)

(1) Methodenköpfe

Welche der folgenden Methodenköpfe repräsentieren tatsächlich unterschiedliche Signaturen:

1. `public String describe(String name, int count)`
   `public String describe(int count, String name)`
2. `public int count()`
   `public void count()`
3. `public int howMany(int compareValue)`
   `public int howMany(int upper)`
4. `public boolean greater(int val1)`
   `public boolean greater(double val1)`
5. `public boolean greater(int val1)`
   `public boolean greater(int val1, int val2)`
6. `public void say()`
   `private void say()`

1. unterschiedliche Signatur
2. nicht unterschiedlich da nur der Rückgabetyp unterschiedlich ist
3. nicht unterschiedlich da nur der Bezeichner des Parameters anders ist.
4. unterschiedliche Signatur
5. unterschiedliche Signatur
6. nicht unterschiedliche Signatur, da nur die Sichtbarkeit unterschiedlich ist.

(2) BizzBuzz

Geben Sie einen Algorithmus in Pseudo-Code an, der für die ganzen Zahlen von 1 bis 100 überprüft, ob diese durch 3 und/oder durch 5 teilbar sind. Falls die Zahl weder durch 3 noch durch 5 teilbar ist, soll die Zahl ausgegeben werden. Ist die aktuelle Zahl durch 3 teilbar, soll statt der Zahl “Bizz” ausgegeben werden. Ist die aktuelle Zahl durch 5 teilbar, soll statt der Zahl “Buzz” ausgegeben werden. Ist die Zahl sowohl durch 3 als auch durch 5 teilbar, soll “BizzBuzz” ausgegeben werden.

pseudo

```
FÜR jede Zahl i von 1 bis 100:
    WENN i durch 3 teilbar ist UND i durch 5 teilbar ist:
        Gib "BizzBuzz" aus
    SONST, WENN i nur durch 3 teilbar ist:
        Gib "Bizz" aus
    SONST, WENN i nur durch 5 teilbar ist:
        Gib "Buzz" aus
    SONST:
        Gib die Zahl i aus
```

(3) Addition zweier Binärzahlen

Wir betrachten das Problem der Addition von zwei n-Bit Binärzahlen, die in zwei Arrays A und B der Länge n gespeichert sind. Die Summe der beiden Zahlen soll in einem Array C der Länge n + 1 gespeichert werden. Geben Sie zur Lösung dieses Problems einen Algorithmus in Pseudo-Code an.
Beispiel: Mit A = 101011 und B = 111110 soll C = 1101001 sein.

```
DEFINIERE Algorithmus AddiereBinär(A, B)
    SEI n die Länge von A
    INITIALISIERE Array C mit Länge n + 1 auf Null

    SEI Übertrag = 0
    FÜR i = n – 1 BIS 0 (RÜCKWÄRTS):
        SEI Summe = A[i] + B[i] + Übertrag
        WENN Summe >= 2 DANN
            C[i + 1] = Summe – 2
            Übertrag = 1
        SONST
            C[i + 1] = Summe
            Übertrag = 0

    C[0] = Übertrag
    GIB C ZURÜCK
```

(4) Pseudocode übersetzten

Übersetzen Sie folgenden Pseudocode in eine statische Methode. Hinweis: Schreiben Sie eine Hilfsmethode für das Tauschen zweier Elemente in einer Liste.

Algorithm 1 shuffle(Liste list mit ganzen Zahlen)

```
n = Größe von list – 1
for i = n, n–1, ..., 2, 1 do
    r = Zufallszahl zwischen 0 und i (i inklusive)
    tausche die Elemente in der Liste an Position i und r
end for
gib die gemischte Liste list zurück
```

```java
public static int[] shuffle(int[] arr) {
        Random random = new Random();

        for (int i = arr.length-1; i > 0; i--) {
                swap(list, i, random.nextInt(i + 1);
        }
        return arr;
}

// Hilfsmethode
private static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
}
```

(5) Welche Ausgabe erzeugt folgendes Programm:

```java
public class Parameter {
     public static void main(String[] args) {
        Language s1 = new Language("Java");
        Language s2 = new Language("Python");
        int i = 12345;
        pass(s1, s2, i);
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(i);
    }
    private static void pass(Language s1, Language s2, int i) {
        s1 = new Language("Ruby");
        s1.increaseVersion();
        s2.increaseVersion();
        i = 54321;
    }
}
//-------------------------------------------------------
public class Language {
    private String name;
    private double version;
    public Language(String name) {
        this.name = name;
        this.version = 1.0;
    }
    public String toString() {
        return this.name + " " + this.version;
    }
    public void increaseVersion() {
        this.version += 0.1;
    }
}
```

```
Java 1.0
Python 1.1
12345
```

(6) Stack nach Operationen

Wie sieht der Stack s aus, nachdem folgende Operationen durchgeführt worden sind:

```
s.push(5);
s.push(21);
s.pop();
s.push(72);
s.push(37);
```

```
s.push(15);
s.pop();
```

```
37 <-- Top
72
5  <-- Bottom
```

(7) Queue nach Operationen

Wie sieht eine Queue q aus, nachdem folgende Operationen durchgeführt worden sind:

```
q.offer(5);
q.offer(21);
q.poll();
q.offer(72);
q.offer(37);
q.offer(15);
q.poll();
```

```
15 <-- last in
37
72 <-- first out
```

(8) Methode `set` in `OwnArrayList` schreiben

Betrachten Sie die Klasse OwnArrayList aus dem Skript. Schreiben Sie eine Methode `set(int index, Object object)`, welche das Element an Position index in der aktuellen Liste mit dem Element object überschreibt. Im Erfolgsfall geben Sie true zurück – falls der Parameter index zu gross ist (grösser als die aktuelle Liste), geben Sie false zurück.

```
public boolean set(int index, Object object){
        if (index >= this.size || index < 0)
                return false;
        this.listElements[index] = object;
        return true;
        }
}
```

(9) Methode `size` in `OwnLinkedList` schreiben

Betrachten Sie die Klassen Node und OwnLinkedList aus dem Skript. Schreiben Sie eine Methode `size()`, welche die Größe der Liste zurückgibt.

```
public int size() {
        Node<E> current = this.startNode;
        if (current == null) { return 0; }
        int count = 1;
        while (current.getNext() != null) {
                count++;
                current = current.getNext();
        }
        return count;
}
```

(10) Methoden `get` und `set` in der Schnittstelle Collection

Weshalb werden die Methoden `get` und `set` nicht in der Schnittstelle `Collection` vorgegeben?

- Die Methoden `get` und `set` arbeiten mit Indizes.
  - Nicht alle Klassen des `Collection` Frameworks stellen Mengen dar, welche durch Indizes geordnet sind (z.B. Klassen, die das Interface `Set` implementieren, sind ungeordnete Mengen, in denen die Objekte keinen Index besitzen).
  - Bei anderen Klassen sollte es gar nicht möglich sein, beliebige Elemente der `Collection` auszulesen resp. zu überschreiben (wie bspw. bei den Klassen `Stack` und `Queue`).

P1 - Serie 08

Lukas Batschelet (16-499-733)

Theorieaufgaben

(01) Rekursive Definition

Schreiben Sie eine rekursive Definition von x^y, wobei x und y ganze Zahlen sind und $y > 0$ gilt.

Lösung

$$\begin{cases} x & \text{wenn } y = 1 \\ x \cdot x^{y-1} & \text{wenn } y > 1 \end{cases}$$

(02) Rekursive Definitionen in Java

Gegeben seien folgende Definitionen:

- `Java_Letter = {A, B, C, ..., Z, a, b, c, ..., z, _, $}`
- `Java_Digit = {0, 1, 2, ..., 9}`

Schreiben Sie eine rekursive Definition für `Java_Identifier` für gültige Java Bezeichner. Sie müssen nur eine Definition angeben und keinen (Pseudo-)Code schreiben. (vgl Skript Kapitel 12.1 s 252)

Mögliche Lösung

1. `Java_Identifier` = `Java_Letter` (Basisfall)
2. `Java_Identifier` = `Java_Identifier + (Java_Letter || Java_Digit)` (Rekursion)

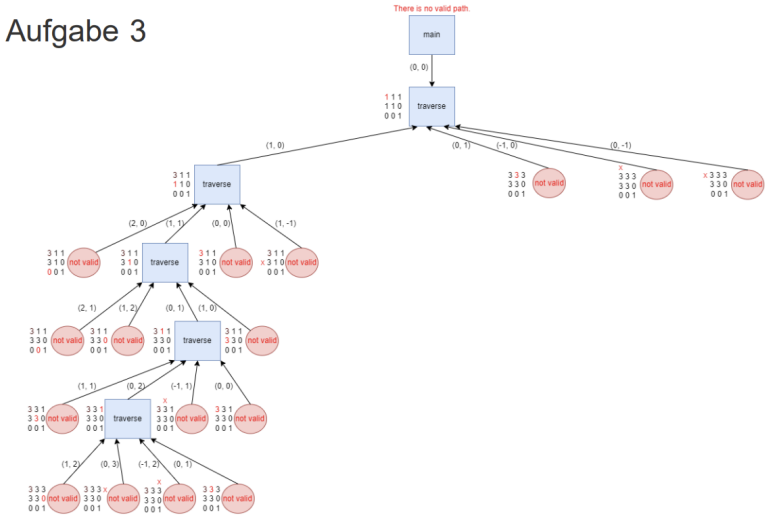
(03) Labirinth rekursiv durchqueren

3. Illustrieren Sie die rekursiven Aufrufe der Methode `traverse` (ähnlich zu Abb. 12.7 im Skript) für das folgende Labyrinth der Größe 3 × 3:

```
1   1, 1, 1
2   1, 1, 0
3   0, 0, 1
```

Es müssen alle Aufrufe beachtet werden, also auch die, die außerhalb der Matrix oder auf einem bereits besuchten Feld sind.

Lösung



Implementationsaufgaben

(01) Datenstruktur `Bag`

1. Setzen Sie für folgende rekursive Definition der Datenstruktur `Bag` in Java um:
   Ein `Bag` ist entweder leer oder enthält ein Objekt vom Typ `Integer` und einen `Bag`.

   - Schreiben Sie zwei Konstruktoren `public Bag()` (für einen leeren `Bag`) und `public Bag(int value)` (für einen `Bag` mit einem initialen Wert).
   - Schreiben Sie eine Methode `public void addValue(int value)`, welche einen neuen Wert zum `Bag` hinzufügt.
   - Schreiben Sie eine Methode `public String toString()` zur Ausgabe des Inhaltes von `Bag`.

   Die Ausgabe von folgendem Testprogramm:

```
Bag b1 = new Bag();
System.out.println(b1);
Bag b2 = new Bag(1);
System.out.println(b2);
Bag b3 = new Bag(1);
b3.addValue(2);
b3.addValue(3);
System.out.println(b3);
```

   soll sein:

```
EMPTY
(1, EMPTY)
(1, (2, (3, EMPTY)))
```

Mögliche Lösung

```
public class Bag {

    private int value;
    private Bag innerBag;

    public Bag() {
        this.innerBag = null;
    }

    public Bag(int value) {
        this.value = value;
        this.innerBag = new Bag();
    }

    public void addValue(int value) {
        if (this.innerBag == null) {
            this.value = value;
            this.innerBag = new Bag();
        } else {
            this.innerBag.addValue(value);
        }
    }

    public String toString() {
        if (this.innerBag == null) {
            return "EMPTY";
        } else {
            return "(" + this.value + ", " + this.innerBag + ")";
        }
    }
}
```

(02) Implementierung der Fibonacci-Funktion

Schreiben Sie eine rekursive Methode `static long fib(int i)`, die die i-te Zahl der Folge
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...
berechnet. Z.B. soll der Aufruf `fib(7)` den Wert 13 liefern. Die erste Zahl der Folge ist 0 (entsprechend `fib(0)`), die zweite 1 (`fib(1)`), danach ist jede Zahl die Summe ihrer beiden Vorgänger, z.B. `fib(8) = fib(7) + fib(6)`.
Allgemein: `fib(n) = fib(n-1) + fib(n-2)` (für `n >= 2`).
Schreiben Sie dazu eine passende `main`-Methode, die die ersten 50 Zahlen der Folge am Bildschirm ausgibt. Was stellen Sie beim Ausführen des Programms fest?

Mögliche Lösungen

```java
public static long fib(int i) {
        switch (i) {
            case 0:
                return 0;
            case 1:
                return 1;
            default:
                return fib(i - 1) + fib(i - 2);
        }
    }

public static long fib(int n) {
        return (n < 2 ? n : fib(n - 1) + fib(n - 2));
}
```

(03) Koordination konkurrierender Aktivitäten

Wir betrachten das Problem der Koordination verschiedener konkurrierender Aktivitäten, die eine gemeinsame Ressource exklusiv nutzen. Angenommen, Sie haben ein Array `A` von `n` Aktivitäten, die zu jedem Zeitpunkt jeweils nur von einer Aktivität genutzte Ressource beanspruchen möchten (z.B. einen Hörsaal an einer Uni, eine Maschine in einem produzierenden Unternehmen, etc.). Jede Aktivität hat zwei Attribute `s` und `f`, welche die Start- und Endzeit der Aktivität definieren (`A[i].s` ist zum Beispiel die Startzeit der i-ten Aktivität). Es gilt: $0 \leq A[i]s < A[i]f < \infty$.

Zwei Aktivitäten `A[i]` und `A[j]` sind kompatibel, wenn sie sich nicht überlappen, d.h., es gilt entweder $A[i].s \geq A[j].f$ oder $A[j].s \geq A[i].f$. Im Aktivitäten-Auswahl Problem wollen wir eine maximale Teilmenge paarweise kompatibler Aktivitäten finden.

Wir setzen voraus, dass die Aktivitäten nach ihren Endzeiten aufsteigend sortiert sind: $A[0].f \leq A[1].f \leq \ldots \leq A[n].f$. Zudem ist `A[0]` eine Dummy-Aktivität mit Start- und Endzeit Null ($A[0].s = A[0].f = 0$).

Wir wollen einen rekursiven Algorithmus für dieses Problem entwickeln. Intuitiv sollten wir zu jedem Zeitpunkt die Aktivität wählen, die die Ressource für möglichst viele andere Aktivitäten freilässt, also eine Aktivität, die möglichst früh endet. Wenn diese Wahl getroffen wurde, haben wir ein verbleibendes Teilproblem zu lösen: eine maximale Menge an Aktivitäten zu bestimmen, die starten, nachdem `A[i]` fertig ist.

Der Algorithmus `Activity-Selector` erhält als Eingabe ein Array `A` mit Aktivitäten, den Index `k`, der das zu lösende Teilproblem definiert, und die Größe `n` des ursprünglichen Problems. Der erste Aufruf von `Activity-Selector` erfolgt mit den Parametern `Activity-Selector(A, 0, A.length)`.

Algorithm 1 Activity-Selector(A, k, n)

```
m = k + 1
while m < n AND A[m].s < A[k].f do
            m = m + 1
end while
if m < n then
    return {A[m]} ∪ Activity-Selector(A, m, n)
else
    return ∅
end if
```

Laden Sie von ILIAS die Dateien `Activity.java` und `ActivitySelector.java` herunter. Die Klasse `Activity` definiert eine Aktivität durch einen Namen, eine Start- und eine Endzeit. In der Klasse `ActivitySelector` wird in der Methode `main` ein Array mit 12 Aktivitäten erzeugt und an die rekursive Methode `activitySelection` übergeben. Ihre Aufgabe ist es, diese Methode zu implementieren und zu testen.

Mögliche Lösung

```java
private static ArrayList<Activity> activitySelection(Activity[] activities, int k, int n) {
        ArrayList<Activity> selection = new ArrayList<Activity>();
        int index = k + 1;

        // Finds the next compatible Activity
        while (index < n && activities[index].getStartTime() < activities[k].getEndTime()){
                index++; }

        if (index < n) {
                selection.add(activities[index]);
                selection.addAll(activitySelection(activities, index, n)); // recursive call
        }
        return selection;
}
```