# 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++;
```

### 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`):
  - 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`.


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

## 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.

```java
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.

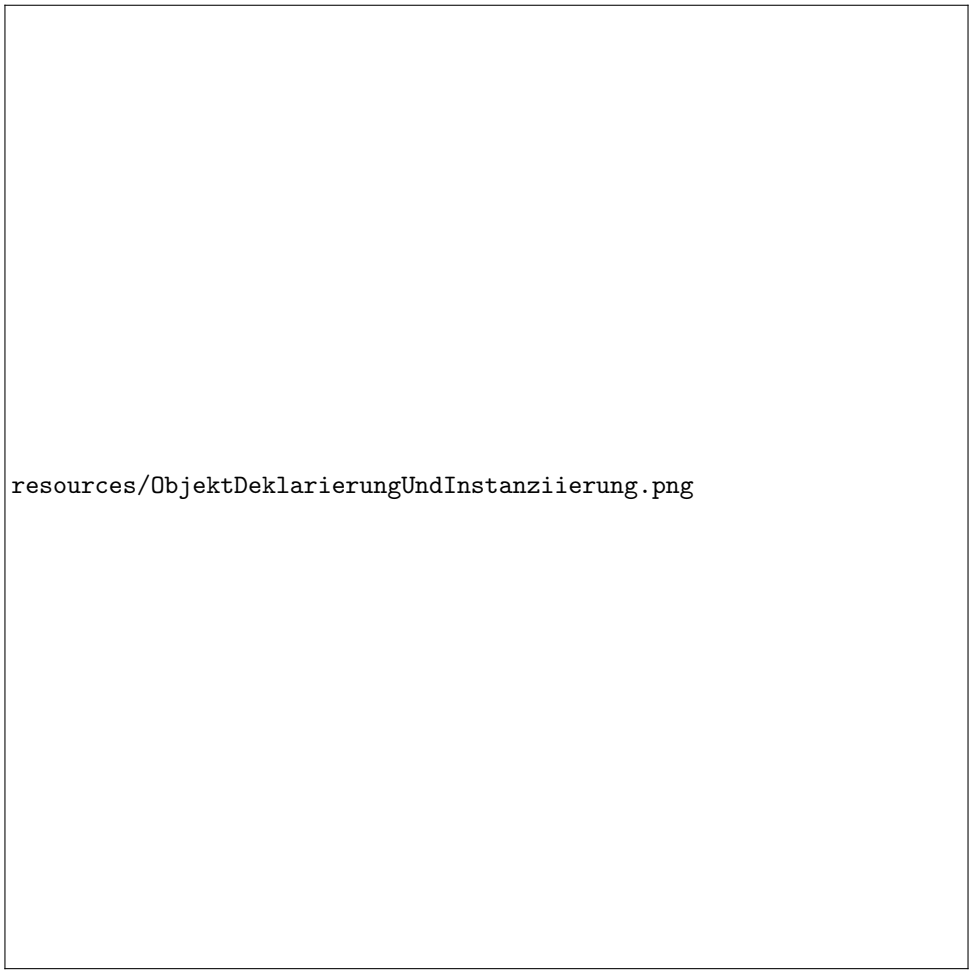


Abbildung 1.2: Deklaration und Instanziierung eines Objektes.

- **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)

```
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

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

```
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"` |

| Klasse & Methode | Beschreibung | Bsp. Eingabe | Bsp. Rückgabe |
|---|---|---|---|
| `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` |
| `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