

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

Dieses Werk ist lizenziert unter einer [Creative Commons](#) “Namensnennung – Nicht-kommerziell – Weitergabe unter gleichen Bedingungen 4.0 International” Lizenz.



Inhaltsverzeichnis

1	Eigene Zusammenfassung	3
1.1	Grundlagen	3
1.1.1	Grundkonzepte der Java-Programmierung	3
1.1.2	Notationskonventionen	3
1.1.3	Variablendeklaration und -zuweisung	3
1.1.4	Primitive Datentypen	4
1.1.5	Casting in Java	4
1.1.6	Aliase und Abhängigkeiten	4
1.1.7	Arithmetische Operatoren und Reihenfolge	4
1.1.8	Division	4
1.2	Java-Klassen	5
1.2.1	Aufbau einer Java-Klasse	5
1.2.2	Methodenkopf	5
1.2.3	Konstruktoren	6
1.2.4	Parameter und variadische Methoden	6
1.2.5	Generische Klassen	6
1.2.6	enum	7
1.2.7	Statische Variablen und Methoden	7
1.2.8	Klassen des java-API	8
1.2.9	ArrayList<T>	9
1.2.10	PrintWriter	9
1.2.11	Abhängigkeit von sich selbst	10
1.2.12	Aggregation	10
1.2.13	Getter und Setter	11
1.3	Schleifen und Bedingungen	11
1.3.1	if-else Anweisung	11
1.3.2	switch-Anweisung	11
1.3.3	Conditional Operator	12
1.3.4	do-while-Schleife	12
1.3.5	for-Schleife	12
1.3.6	Vergleiche	13
1.3.7	do-Anweisung	14
1.3.8	Vergleich von Daten	14
1.4	Arrays	14
1.4.1	Instanziierung	15
1.4.2	Variable Parameterlisten	15
1.4.3	Mehrdimensionale Arrays	16
1.5	Schnittstellen und Vererbung	16
1.5.1	Schnittstellen	16
1.5.2	Vererbung	17
1.5.3	Polymorphismus	18
1.6	Algorithmen und Methoden	18
1.6.1	Überladen von Methoden	18
1.6.2	Algorithmen	19
1.6.3	Generische Typisierung	19
1.6.4	Herrsche und Teile	19
1.6.5	Rekursion	20
1.6.6	Testen	20

1.7	Collection Framework	21
1.8	Laufzeitfehler	22
1.8.1	Laufzeitfehler abfangen	22
1.8.2	Laufzeitfehler weitergeben	23
1.8.3	Eigene Exception Klassen	23
2	tl;dr Kapitel 1 bis 13	24
2.1	Grundlagen	24
2.1.1	Datentypen und Konventionen	24
2.1.2	Variablen und Datentypen	25
2.1.3	Division	25
2.1.4	Boolsche Ausdrücke und Verzweigungen	26
2.1.5	Java API	26
2.1.6	Methoden	27
2.1.7	Datentypen	27
2.2	Klassen und Methoden	27
2.2.1	Sichtbarkeitsmodifikatoren	27
2.2.2	Methoden	27
2.2.3	Wrapper Klassen	28
2.2.4	Generische Klassen	30
2.2.5	Arrays	30
2.2.6	switch-Anweisung	31
2.2.7	Conditionals	31
2.2.8	do-Anweisung	31
2.2.9	Schleifen	32
2.2.10	Gleichheit von Objekten	32
2.3	Arrays	33
2.3.1	main-Methode	34
2.3.2	enum	35
2.3.3	Statische Variablen	35
2.3.4	Abhängigkeiten	36
2.4	Rekursion	39
2.4.1	Laufzeitfehler	40

Eigene Zusammenfassung

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

- **Variablenamen:** 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++;
```

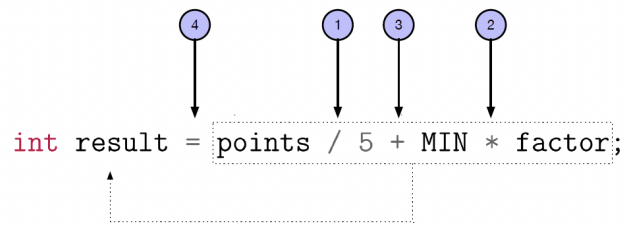


Abbildung 1.1: Reihenfolge der Auswertung: Der Zuweisungsoperator = hat die niedrigste Priorität.

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`).
- **Rückgabentyp:** 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 instanziiieren, müssen wir sie zusammen mit einem *Typargument* instanziiieren:

```
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 instanziiieren

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

```
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.

```
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.

```
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

- 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
```

- 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 {
```

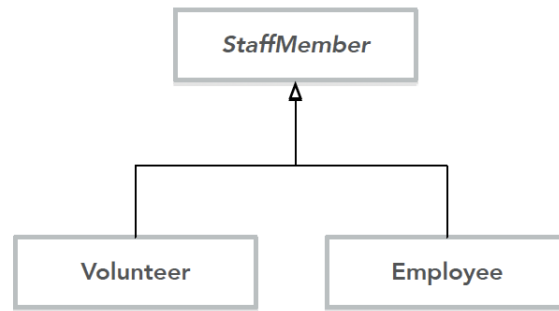
```
public class Potato implements Eatable {
```

```
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



```

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

```

```

public class Backpack implements Bag {
    private ArrayList<Item> items;

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

```

```

Bag bag = new Backpack();
bag.add(new Item("Socken"));
bag.add(new Item("Hosen"));
bag.add(new Item("Shirt"));

```

```

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++;
    }
}

```

```

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

```
public abstract class StaffMember {

    protected String name;

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

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

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

```
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

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

- Polymorphes Verhalten via Vererbung

```
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
- Rückgabebetyp kann nicht überladen werden

```
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, ..., (list.length - 2) do
    min = index
    for scan = index + 1, ..., (list.length - 1) do
        if list[scan] < 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

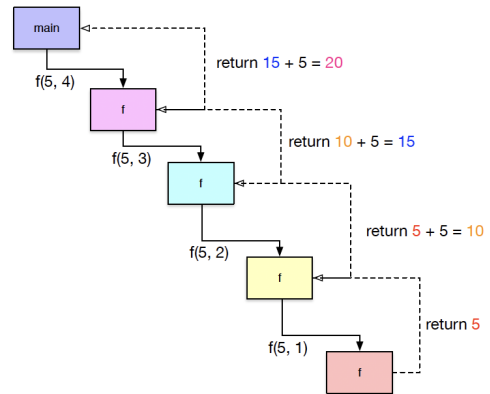
```
public static <T> void bubbleSort(Comparable<T>[] list) {
    for (int i = 0; i <= list.length - 2; i++)
        for (int j = list.length - 1; j >= i + 1; j--)
            if (list[j].compareTo((T) list[j - 1]) < 0)
                swap(list, j, j - 1);
}

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

```
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

```
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

```
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

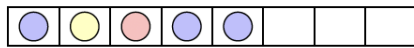
- White-Box Test: Konzentriert sich auf die interne Struktur des Codes

1.7 Collection Framework

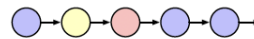
- Sammlung := Behälter, um (meist gleichartige) Elemente zu organisieren.
- Idee einer Sammlung und die Implementierung sind zwei unterschiedliche Dinge

Idee: Liste

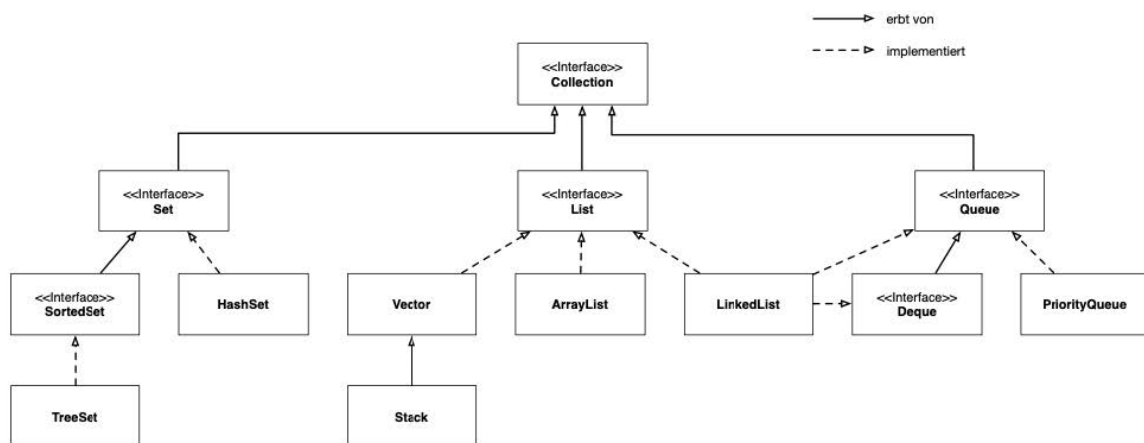
Implementierung mit Array



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 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()`

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("\nDie Anzahl der Dummheiten übersteigt die der "
    + "Primzahlen.\nGibt es nicht unendlich viele "
    + "Primzahlen?\n\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 (ungenaueres 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 Boolesche 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* Rückgabety 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 instanziiieren 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 Rückgabety 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 instanziiieren, müssen wir sie zusammen mit einem *Typargument* instanziiieren:
 - `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 *einzig* 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.

```

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**

```

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

```

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

```

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

```

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

```

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:

```

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
- **Defektttest** := 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**

```
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**

```
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**


```

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

```

1 exception in thread "main"
2 java.lang.ArithmeticException: / by zero at
3 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

```

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, Behandelte Fehlerfall, Unbehandelter Fehlerfall)
- Jede Subklasse der Klasse `Exception` stellt (via Vererbung) zwei Methoden zur Verfügung

```
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

```
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

```
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);
    }
}

```

- Die Klasse `PrintWriter` erlaubt Ausgaben in Dateien

```

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

```