




Java Lernzettel

Dies ist der Lernzettel für die IT Klausur - Basierend auf dem herausgegebenen Lernzettel von Herrn Lam und mit vielen Ergänzungen, Übungsaufgaben und Empfehlungen von Dennis.

Inhalt

1.  [Objektorientierte Programmierung](#)
2.  [Konzepte von Greenfoot](#)
3.  [Java Grundlagen](#)
4.  [Datentypen](#)
5.  [Rechnen mit Variablen](#)
6.  [Klassen und Objekte](#)
7.  [Bedingungen](#)
8.  [Schleifen](#)
9.  [Ausgaben](#)
10.  [Sauberen Code schreiben!](#)

Legende

Markierung	Bedeutung
	Besonders <i>wichtiger</i> Abschnitt.
	Kurze <i>Zusammenfassung</i> .
	<i>Zusatzinfos</i> , die beim Verständnis helfen können, aber nicht notwendig sind.

Objektorientierte Programmierung

Java ist eine der größten sogenannten *Objektorientierten Programmiersprachen* und Greenfoot verwendet Java als Programmiersprache. Deshalb ist es wichtig, überhaupt zu verstehen, womit wir arbeiten.

Was ist das überhaupt?

In der klassischen Programmierung geht es darum, einen *Ablaufplan* zu erstellen, dessen Befehle einfach chronologisch vom jeweiligen Gerät abgearbeitet werden.

Im Gegensatz dazu versucht die *Objektorientierte Programmierung* so viele Programmteile wie möglich zu **Modularisieren**, dadurch kann der Code sehr viel kürzer werden, da Wiederverwendungen vermieden werden. Dieser kürzere Code ist im Zweifel auch leichter zu verstehen, zu testen und zu warten.

Konzepte der Modularisierung

Um einen modularen Code zu erreichen, werden Codeabschnitte in **Objekte** unterteilt. Diese Objekte können mehrfach erzeugt werden und bleiben gespeichert, solange das Programm läuft (Oder bis sie im Programm gelöscht werden). Objekte können festgelegte Eigenschaften haben (**Attribute** genannt) und eigene Funktionen besitzen (**Methoden** genannt). Um ein Objekt zu erzeugen, muss zuvor eine Art Bauplan erstellt werden, der alle verfügbaren Attribute und Methoden eines Objektes festlegen kann. Ein solcher Bauplan heißt **Klasse**.

Um dieses Konzept besser zu verstehen, hier ein **Beispiel**:

Es gibt eine Klasse namens *Auto*. In der Klasse ist festgelegt, dass alle Autos eine *Marke* haben. Außerdem kann jedes Auto *fahren*, Autos haben also eine Funktion namens *fahren()*.

Dann wird das Programm ausgeführt und ein *Auto* wird erzeugt. Dieses spezielle *Auto* ist jetzt also gebaut worden und existiert jetzt als *Objekt*. Dieses Auto bekommt die *Marke* "Lada" zugewiesen und es soll drei Mal *fahren*, diese Funktion wird also 3 Mal ausgeführt.

✳ Advanced Stuff

Viele Programmiersprachen unterstützen außerdem das Konzept der *Vererbung*. So kann man in Java beispielsweise eine Klasse "Auto" erstellen und die Klasse "VW" erbt dann alle **Eigenschaften und Funktionen** der "Auto"-Klasse, wird aber z.B. um die Eigenschaften "Modell" und "PS" erweitert.

Objekte sind grundsätzlich von der Außenwelt *abgekapselt* und können nicht ohne weiteres von außen beeinflusst werden (es sei denn, man will das). Objekte entscheiden grundsätzlich selbst über ihr Verhalten. Mehr dazu im Kapitel [Klassen und Objekte](#).

Konzepte von Greenfoot

Greenfoot ist eine sogenannte **IDE** (Integrated Development Environment). Zu Deutsch, es handelt sich um eine Entwicklungsumgebung, also ein Programm, in dem man Programmiercode schreiben kann. Deshalb nimmt Greenfoot dem Entwickler etwas Arbeit ab, indem dieser den Code nicht selbst zu Maschinencode umwandeln muss. Was Greenfoot außerdem tut, liest Du hier:

Actor und World

In Greenfoot gibt es sogenannte **Actors**. Im Prinzip ist das nur eine festgelegte Oberklasse, aus der all deine selbst-erstellten Actors erben. Alle Actors können *act()* ausführen, das passiert in Greenfoot bei jedem Tick / Frame. Außerdem haben sie ein Anzeigebild, das man in Greenfoot vereinfacht durch Rechtsklick auf den Actor verändern kann.

Außerdem gibt es **Worlds**, dies sind auch nur festgelegte Klassen, in denen Objekte generiert und platziert werden können. Jede World wird also beim Ausführen auch nur als ein Objekt erstellt, das dann die anderen (Actor-)Objekte erstellt und dadurch weiterhin auf diese zugreifen kann.

Oberfläche

Die **Oberfläche** von Greenfoot enthält ein großes Fenster, in dem live (als Vorschau) eine Welt geladen werden kann, außerdem können *act()* Zyklen simuliert werden. Ein Klick auf "Run" simuliert diese Zyklen im Loop.

An der rechten Leiste sind alle Klassen des Spiels aufgelistet. Dort können neue Actors und Worlds erstellt werden.

Java Grundlagen

Java ist eine *Objektorientierte Programmiersprache*, die auf vielen Geräten und Systemen ausführbar ist. Java-Code wird als Programm in einen **Zwischencode** umgewandelt ("compiled"), der dann von je nach System / Gerät in den jeweiligen Maschinencode **interpretiert** wird. Dies übernimmt der *Java Interpreter*. Der große Vorteil von Java ist, dass es auf so vielen Systemen funktioniert. Der Nachteil ist, dass solcher *interpretierter Code* etwas langsamer ist, Spiele laufen in Java generell langsamer als z.B. in C++.

Die main()-Methode

Da Java zu 100% Objektorientiert ist, ist selbst das Programm nur ein Objekt. Jedes Programm hat eine *main()*-Funktion, die als Einstiegspunkt dient, damit der *Interpreter* weiß, was zu tun ist.

Um das zu veranschaulichen, hier das bekannte "Hello-World!" Beispiel:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello world!");
    }
}
```

- Die Hauptklasse des Programms heißt *HelloWorld*.
- Diese Klasse besitzt die Funktion *main()* mit folgenden Eigenschaften:
 - Sie ist *public*, kann also von einem anderen Objekt ausgeführt werden.
 - Sie ist *static*, d.h. existiert pro Klasse nur einmal.
 - Sie ist *void*, d.h. sie gibt keinen Wert zurück, bzw. hat kein festes Ergebnis.
 - Sie heißt "*main*"
 - Der "(*String[] args*)"-Parameter kann weitere Argumente beim Ausführen an die Funktion weitergeben, d.h. wenn das Programm in der Konsole mit "helloworld.exe -f -j" gestartet wird, ist die Variable "args" gleich ['-f','j'], also ein Array aus mehreren Strings.
 - In der Funktion wird der Befehl *System.out.println("Hello World!");* ausgeführt, der einfach *Hello World* in der Konsole ausgibt.

Syntax

Der **Syntax**, also wie Code in Java strukturiert und artikuliert werden muss, ist recht eindeutig:

- Befehle werden immer als *befehl();* geschrieben, also mit Klammern und einem Semikolon.
- Funktionsblöcke werden immer in *{ }* gefasst.
- Arrays stehen in *[]* Klammern.
- Es kann eingerückt und verschoben werden, wie nur möglich, aber die *{ }* und *;* Zeichen müssen gesetzt werden.
- Einzeilige Kommentare werden mit *//* und mehrzeilige mit */** und **/* geschrieben.

Alles weitere

...wird in den folgenden Abschnitten detailliert erklärt.



Datentypen

Wann immer etwas neues im **Speicher** abgelegt werden soll, müssen dessen Zugriffsrechte und der Datentyp festgelegt werden.

Zugriffsrechte

Bei der Erstellung muss mithilfe eines **Zugriffsmodifikators** festgelegt werden, von wo auf die Daten zugegriffen werden kann. Sie können festgelegt werden für *Klassen, Variablen & Methoden* - Folgendes ist Möglich:

Code	Beschreibung
<i>Standardwert</i>	Sichtbar für das Programmpaket.
<code>public</code>	Sichtbar für alle.
<code>private</code>	Sichtbar innerhalb der eigenen Klasse.
<code>protected</code>	Sichtbar für das Programmpaket und alle Subklassen.

✳ Modifikatoren

Abseits der Zugriffsmodifikatoren können sogenannte **Nicht-Zugriffs Modifikatoren** festgelegt werden.

Code	Gilt für	Beschreibung
<code>static</code>	<i>Klassen, Methoden</i>	Begrenzt die Erstellung auf ein Exemplar pro Klasse
<code>final</code>	<i>Klassen, Methoden & Variablen</i>	Verhindert eine Änderung nach der Erstellung.
<code>abstract</code>	<i>Klassen, Methoden</i>	Verhindert Initialisierung (Erstellung) als Objekt, Subklassen können aber noch initialisiert werden.

Datentypen

Bei der Erstellung von **Variablen** muss der Typ festgelegt werden, also was gespeichert werden soll.

Code	Wertebereich	Beschreibung
<code>boolean</code>	<code>true</code> oder <code>false</code>	Ist wie ein Schalter - kann nur aktiviert oder deaktiviert werden.
<code>byte</code>	-128 bis 127	Speichern von kleineren ganzen Zahlen.
<code>short</code>	-32768 bis 32767	Speichern von etwas größeren Ganzzahlen.
<code>int</code>	-2147483648 bis 2147483647	Speichern von großen Ganzzahlen, wird am häufigsten verwendet.
<code>long</code>	-9223372036854775808 bis 9223372036854775807	Speichern von sehr großen Ganzzahlen.
<code>float</code>	3.4e-38 bis 3.4e+38	Speichern von Kommazahlen.
<code>double</code>	1.7e-308 bis 1.7e+308	Speichern von sehr kleinen Kommazahlen. (Genauer / Besser als <code>float</code> , verbraucht aber 64 statt 32 Bits im RAM)
<code>BigDecimal</code>	$\pm 2240 \cdot 10^{232}$	Speichern von großen Zahlen mit Kommaanteil (z.B. Währungen)
<code>char</code>	Buchstabe / Zeichen	Speichern einzelner Buchstaben / Zeichen.
<code>String</code>	Sequenz mehrerer <code>char</code> 's	Speichern von längeren Sätzen oder Wörtern.
<code>[]</code>	Array enthält Variablen, Objekte, Arrays, etc.	Speichern beliebiger Daten an einem Ort. Kann gut durch Loops erstellt und abgerufen werden.

Arrays

Ein Array ist eine **Liste von Objekten** (Also auch Variablen). Man nutzt dabei die `[]` wie folgt:

```
// Erstelle eine Liste von Integers
int zahlen[] = {128,1,29,-69};

// Zugriff auf die 2. Zahl im Array:
// WICHTIG: Der Array-Index startet bei 0. Also ist das 0te Item das erste, das 1te ist das
// zweite etc.
System.out.println(zahlen[1]); // Gibt "1" aus.
```

Funktionen / Methoden

Eine **Funktion** ist ein Codeabschnitt, der separat abgespeichert wird, damit er einfacher mehrmals ausgeführt werden kann. Eine Funktion erfüllt normalerweise immer einen bestimmten Zweck.

Beispiel: Überprüfe etwas. Oder setze mehrere Variablen zurück.

```
// Eine Variable
int eineZahl = 16;

// Funktion definieren
public void neueZahl() {
    eineZahl = 10;
}

// Funktion ausführen
neueZahl();
```

Funktionen werden in Java **Methoden** genannt und gehören immer zu *der Klasse / dem Objekt* in dem sie definiert wurden.

Rückgabetypen

Am einfachsten lassen sich **Rückgabetypen** anhand einer Funktion erklären:

```
boolean isValid = checkValid(3);
```

Hier soll überprüft werden, ob die Zahl 3 "valide" ist (was immer das hier heißen mag). Dazu wird die Funktion `checkValid()` mit dem Parameter 3 ausgeführt, diese **gibt dann etwas zurück**.

Die Funktion `checkValid()` könnte so aussehen:

```
public boolean checkValid(int input) {
    if (input >= 2) {
        return true;
    } else {
        return false;
    }
}
```

Lässt man sich das Ergebnis von `checkValid(3)` ausgeben, so gibt die Konsole `true` aus.

Statt `boolean` lassen sich auch beliebige andere Datentypen verwenden. Gibt es keinen Rückgabety, muss `void` verwendet werden. Gibt es einen, muss `return` dann eine Antwort zurückgeben. Jeder Parameter muss in der Klammer mit Datentyp angegeben werden. Argumente werden per Komma getrennt.

✳ Scope

Der **Scope** legt fest, zu welchem Objekt bzw. welcher Klasse die *Variablen und Methoden* gehören. Das wird festgelegt, je nach dem wo der Code geschrieben steht. Beispiel:

```


public world() {
    int globalCounter = 0;

    public void starteLevel() {
        // Erstelle die Variable "gestartet" und setze sie auf "true"
        boolean gestartet = true;
        globalCounter++;
    }

    public void beendeLevel() {
        globalCounter++;
    }
}

```

- Die Variable `globalCounter` gehört zu `world()` und ist außerhalb nicht verfügbar.
- Sowohl `starteLevel()` als auch `beendeLevel()` können auf `globalCounter` zugreifen und diesen erhöhen.
- `gestartet` gehört zu `starteLevel()` und ist außerhalb nicht verfügbar. `beendeLevel()` kennt die Variable nicht.

 **Zusammenfassend** gibt es eine klare Hierarchie und erstellte *Objekte, Methoden und Variablen* können immer nur in derselben oder einer tieferen Ebene genutzt werden.

Beispiel

```

// Erstelle zwei Integers, die gleich einem wert zugewiesen werden.
int posX = 89;
int posY = -1200;

// Erstelle einen Boolean und setze ihn danach auf true.
boolean schalter1;
schalter1 = true;

// Erstelle eine Kommazahl, die noch keinen wert hat.
double preis = 0.93127;

// Erstelle einen String, der festgelegt wird.
String name = "Abigail";

// Gebe alle Daten in der Konsole aus, zum Testen:
System.out.println("posX: " + posX);
System.out.println("posY: " + posY);
System.out.println("schalter1: " + schalter1);
System.out.println("preis: " + preis);
System.out.println("name: " + name);

```

Kopiere diesen Code ruhig in Greenfoot und führe ihn aus! z.B. könntest Du ihn in den Konstruktor (nach dem `super();`) der aktuellen Welt einfügen und dann per `Rechtsklick > new Myworld()` auf die Welt den Code ausführen.



Rechnen mit Variablen

Mithilfe des `=` Operators lässt sich eine **Variable zuweisen**. Das lässt sich nutzen. Auf der rechten Seite kann deshalb auch eine Rechnung stehen, die wird vom Programm dann zuerst ausgerechnet und dann ganz normal zugewiesen. Normale Rechenoperatoren wie `+` `-` `*` `/` `()` werden ganz normal verstanden.

Außerdem lassen sich spezielle Mathe-Funktionen zur Berechnung nutzen.

Beispiel

```
// Erstelle eine Variable
int ergebnis;

// Berechnung irgendeiner Formel
// Das Ergebnis wird der Variable zugewiesen
ergebnis = (6+3)*2-1;

// Gebe das Ergebnis aus:
// Erwartungswert: 6+3 = 9 und 9*2 = 18 und 18-1 = 17.
System.out.println("Ergebnis: " + ergebnis);

// Eine weitere Rechnung:
double einViertel = 1.00/4; // Die Zahl der Nachkommestellen muss hier schon festgelegt werden.
double zweiViertel = 1*0.5;

// Ausgabe:
System.out.println("einViertel: " + einViertel + " zweiViertel: " + zweiViertel);

// Wiederverwendung einer alten Variable:
ergebnis = ergebnis + 18;
System.out.println("Neues Ergebnis: " + ergebnis);

// Direktes Rechnen in der Ausgabe:
System.out.print("Drittes Ergebnis: ");
System.out.println(ergebnis * 123);
```

Output

In der **Konsole**, die dann aufgeht, steht dafür folgendes:

```
Ergebnis: 17
einViertel: 0.25 zweiViertel: 0.5
Neues Ergebnis: 35
Drittes Ergebnis: 4305
```



Klassen und Objekte

Wie im Kapitel [Objektorientierte Programmierung](#) erklärt, gibt es in Java sogenannte **Klassen und Objekte**, wie diese Konzepte in Java umzusetzen sind, lässt sich wieder anhand eines Beispiels anschaulich erklären:

Beispiel

```
import greenfoot.*; // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

public class ExampleWorld extends world
{
    // Variable:
    int zufall = 0;

    // Konstruktor für ExampleWorld:
    public ExampleWorld()
    {
        // Setze die weltgröße für Greenfoot>world
        super(600, 400, 1);

        // Setze "zufall" auf irgendwas
        zufall++;

        // Die Methode wird ausgeführt:
        outputLo1();
    }

    // Zusätzliche Methode:
    public void outputLo1() {
        System.out.println("Lo1.");
    }
}
```

- *Importiere* alle verfügbaren Klassen von Greenfoot, in diesem Fall wird vor allem die *Welt*-Klasse gebraucht
- `public class Exampleworld extends world` erstellt eine neue Welt, die aus der Oberklasse `world` erbt. Die Oberklasse stammt aus dem importierten Greenfoot-Paket.
- Die Variable `zufall` wird in der Klasse erstellt, aber sobald aus der Klasse ein Objekt erstellt wird (Sobald der Bauplan gebaut wird), wird die Variable erhöht:
- Der **Konstruktor** `public Exampleworld()` wird ausgeführt, sobald ein `Exampleworld()`-Objekt erstellt wird.
- Die Methode `outputLo1()` wird ohne [Rückgabety](#) erstellt und im Konstruktor ausgeführt.

Bedingungen

In der Programmierung ist es wichtig, **entscheiden zu können**, was passieren soll. Zum Beispiel innerhalb von *If-Blöcken* kann einfach die Ausführung eines Blocks an eine **Bedingung** geknüpft werden.

Bedingungen

Das Ergebnis einer Bedingung muss immer `wahr` oder `false` sein. Eine Funktion kann nicht zu 50% ausgeführt werden - Entweder sie wird ausgeführt oder nicht.

Die einfachste Bedingung lautet daher: `(true)` oder `(false)`. Beispiel:

```
while (true) {  
    // Code  
}
```

Dies ist ein einfacher, unendlicher Loop. Jedes etwas ältere Spiel hat einen solchen Loop.

Um solche Entscheidungen auch Dynamisch zu treffen, kann eine Reihe von Überprüfungen angewandt werden. Das Ergebnis ist immer `true` oder `false`:

Operator	Funktionsweise
<code><</code>	A Kleiner als B
<code>></code>	A Größer als B
<code><=</code>	A Kleiner oder Gleich als B
<code>>=</code>	A Größer oder Gleich als B
<code>==</code>	A Genau gleich wie B
<code>!=</code>	A Nicht gleich wie B

Das kann beispielsweise so aussehen:

```
if (a < 15) {  
    // Mache etwas  
}
```

Man kann auch mehrere Bedingungen miteinander verknüpfen:


- `||` ist gleich *oder*, es wird aktiv wenn eine der Bedingungen wahr ist.
- `&&` ist gleich *und*, es wird aktiv wenn beide Bedingungen wahr sind.
- `!` kann jede Bedingung invertieren.

Das kann dann so aussehen:

```
if (3 <= 18 && true) {  
    // Bedingung ergibt "wahr" -> Code wird ausgeführt  
}  
  
if(((7 > 5) && (8 > 5)) || (5 < 3)) {  
    // Bedingung ergibt auch "wahr" -> Code wird ausgeführt  
    // Bedingungen können mit () eingegrenzt werden,  
    // sonst wird von vorne nach hinten geprüft.  
}
```

If, Else und Switches

Verzweigungen erlauben nach der Prüfung einer Bedingung, zu entscheiden, ob Code ausgeführt werden soll. Es gibt dazu folgende Befehle:

Code	Beschreibung
<code>if (Bedingung)</code>	Wenn die Bedingung zutrifft wird der nachfolgende Code ausgeführt. Der Code kann entweder danach in derselben Zeile stehen, oder zwischen <code>{ }</code>
<code>else</code>	Kann nach den <code>{ }</code> eines <code>if</code> -Blocks stehen und wird ausgeführt, wenn dieser nicht ausgeführt wurde.
<code>else if</code>	Wird ausgeführt, wenn der vorherige <code>if</code> -Block nicht ausgeführt wurde. Besonders hilfreich in einer Verkettung aus mehreren Bedingungen.
 <code>switch (var)</code>	Kann viele Möglichkeiten für die Variable in Klammern durchgehen. Format: <code>switch (var) { case 1: code1(); case 2: code2(); }</code> Wenn <code>var == 1</code> dann wird <code>code1()</code> ausgeführt.

Das kann so aussehen:

```
// Einzeilige If-Bedingung
if (schalter == true) schalteEsAb();

// Mehrzeilige If-Bedingung
if (a > 1 || a < 0) {
    b++;
}

// Mehrere Möglichkeiten durch If / Else
if (unterkurs.vorhanden >= 1) {
    sammleUnterkurse();
} else if (unterkurs.vorhanden == 0) {
    glueckwunsch();
} else {
    System.out.println("Error!");
}

// viele Möglichkeiten einer variable
switch (monat) {
    case 1: monatName = "Januar";
    case 2: monatName = "Februar";
    case 3: monatName = "März";
    // Und so weiter...
}
```

Ein weiterer wichtiger Bestandteil der Programmierung ist es, Codeabfolgen wiederholen zu können. Das spart Code und ermöglicht u.a. erst Spiele.

Arten von Schleifen

While-Schleife

```
while ( Bedingung ) {  
    // Ausführbarer Code  
}
```

Die *While-Schleife* ist im Grunde nur ein If-Block, der am Ende wieder zum Anfang springt und die Bedingung prüft etc...

For-Schleife

```
for(a = 0; a<= 60; a++) {  
    // Ausführbarer Code  
}
```

Die *For-Schleife* ist wie eine While-Schleife, die automatisch zwei Code-Blöcke ausführt. Das sieht so aus:

(Start-Code; Bedingung; End-Code)

- Der **Start-Code** wird einmal als erstes beim Erreichen der Schleife ausgeführt.
- Die **Bedingung** funktioniert wie bei der *If-Schleife*.
- Der **End-Code** wird nach jedem Durchlauf des *Ausführbaren Codes* ausgeführt.

In der Praxis wird das meistens verwendet, um einen **Counter** zu haben, der den Code eine bestimmte Häufigkeit lang ausführt.

Ausgaben

Konsolenausgabe

- `System.out.println()` gibt den Text in Klammern in die Konsole aus und erstellt dann eine neue Zeile.
- `System.out.print()` gibt den Text in Klammern in die Konsole aus, ohne neue Zeile.

✳ Systempakete

In Java werden Programme in sogenannte **Pakete** unterteilt. Einige Pakete gibt es immer wie `System` und `Java`. In Greenfoot wird auch z.B. das Paket `Greenfoot` importiert.

Deshalb heißt auch der Befehl zur Konsolenausgabe `System.out.print()`, er gehört zum `System`-Paket, in diesem Paket gibt es ein `out`-Paket, das wiederum für Ausgaben zuständig ist.

Pakete sind übrigens auch nichts anderes als **Objekte**. Man kann auf Objekt-Eigenschaften genauso zugreifen: `Objekt.Eigenschaft = 0;`

Sauberen Code schreiben!

✳️ Dieses Kapitel ist zu 100% optional!

Sauberen Code zu schreiben ist wichtig!

Sauber ist der Code dann, wenn er einfach zu lesen ist. Außerdem sollte er keine Logikfehler und Bugs haben.

Man sollte Code möglichst **sauber** halten, damit er einfacher zu verstehen, dadurch weniger Fehleranfällig und im Zweifel auch performanter ist. Mit gutem Code lässt sich auch einfacher arbeiten. Deswegen gibt es in diesem Kapitel ein paar Beispiele und Ideen, wie guter Code zu schreiben ist.

Namen sinnvoll wählen!

Ein Beispiel: Eine Variable, die speichern soll, **ob** das Programm **aktiv** ist, könnte `akt` heißen. Aber jemand, der das Programm nicht geschrieben hat, wüsste nicht auf Anhieb, was diese Variable enthält.

Stattdessen könnte man diese Variable `isActive` nennen. Die `is`-Vorsilbe zeigt sofort, dass es sich um ein `boolean` handelt und `Active` ist eindeutig zu verstehen.

Zweites Beispiel: Eine **Klasse**, die allen *dämonischen* Welten übergeordnet ist, heißt `DämonenKlasse`.

Aber aus diesem Namen geht nicht hervor, dass es sich um Welten handelt. Außerdem sollte *Klasse*, also der Datentyp, nicht direkt in den Namen. Zuletzt programmiert man immer auf *Englisch*, Java lässt also einen Klassennamen mit einem `ä` gar nicht zu, das sollte generell vermieden werden.

Besser wäre hier: `UnterWelten` oder `DaemonWorlds`

Gute Formattierung!

Folgende Regeln könnten dir helfen:

- Rücke Inhalte in geschweiften Klammern immer ein!
- Halte auch mal einen Absatz Freiraum zwischen den Zeilen für bessere Struktur.
- Halte dich an deine eigenen Regeln! Wenn Du z.B. gerne `(a > b)` schreibst statt `(a>b)`, dann zieh das durch! Bleibe konsequent.
- Fasse dich kurz. Der eigentliche Code sollte immer kurz und übersichtlich sein.

Beispiel für schlechten Code

```
if ( a==32)
{
  lol(); raaaad(25)}
      System.out.println("Fertig!");
```

Beispiel für guten Code

```
if ( a >= 3 ) {
    resetValues();
} else {
    System.out.println("A ist kleiner als 3!");
}
```

Kommentare

```
// Einzeiliger Kommentar

/*
    Mehrzeiliger Kommentar
*/
```

Nutze sie! Nicht jede Zeile muss kommentiert werden, aber zu beschreiben, wie Du ein bestimmtes Problem gelöst hast, ist wichtig! Du selbst wirst diesen Code dadurch sehr viel einfacher lesen können, genauso wie alle anderen Leute, die deinen Code lesen.





Netzwerktechnik Lernzettel

Dies ist der Lernzettel für die IT Klausur, basierend auf dem letzten Lernzettel für die Netzwerktechnik-Klausur!

⚠ Zum **Filius-Teil** geht es [hier](#)!

⚠ Zum **Zahlensysteme-Teil** geht es [hier](#)!

Inhalt

1.  [Das OSI-Modell](#)
2.  [IP-Adressen und Subnetze](#)
3.  [Protokolle](#)
4.  [Geräte](#)

Das OSI-Modell

Das „*Open Systems Interconnection Model*“ ist ein **Referenzmodell**, das seit 1984 durch die ISO (International Organization for Standardization) anerkannt wurde. Es stellt die **Netzwerkprotokolle als unabhängige Ebenen** dar, die ineinander verkapselt dann genau so in der Praxis verwendet werden. So sieht es aus:

Ebene	Orientierung	DoD-Schicht	Einordnung	Protokolle	Geräte
7 Anwendungen	Anwendung	Anwendung	Ende zu Ende	HTTP, DNS, DHCP, XMPP ...	Gateway, Proxy ...
6 Darstellung	Anwendung	Anwendung	Ende zu Ende	HTTP, DNS, DHCP, XMPP ...	Gateway, Proxy ...
5 Sitzung	Anwendung	Anwendung	Ende zu Ende	HTTP, DNS, DHCP, XMPP ...	Gateway, Proxy ...
4 Transport	Transport	Transport	Ende zu Ende	TCP, UDP ...	Gateway, Proxy ...
3 Vermittlung	Transport	Internet	Ende zu Ende	ICMP, IP ...	Router
2 Sicherung	Transport	Netzzugriff	Direkte Verbindung (Punkt zu Punkt)	Ethernet, MAC	Bridge, (Normaler) Switch
1 Physisch	Transport	Netzzugriff	Direkte Verbindung (Punkt zu Punkt)	-	Kabel, Repeater, Hub

Weitere Erläuterungen dazu:

Begriff	Erklärung
DoD	Kleineres Schichtenmodell
Ende zu Ende	Paket kann über viele Rechner springen, hat aber einen Start und ein Ziel.
Direkte Verbindung	Beide Geräte müssen direkt miteinander verbunden sein.

Was machen die einzelnen Ebenen?

Schicht	Erklärung
7 Anwendungen	Was-auch-immer-der-Nutzer-machen-möchte. z.B. Spiele oder Email-Clients
6 Darstellungen	Übersetzt und verschlüsselt die Daten auf Anwendungsebene, damit der Nutzer sie lesen kann.
5 Sitzung	Erstellt und verwaltet Verbindungen auf <i>Anwendungsebene</i> . Es wird sichergestellt, dass eine Verbindung zwischen Programmen dauerhaft aufrecht erhalten wird.
4 Transport	Je nach Paketprotokoll sollen hier die Pakete erstellt und etikettiert werden. Dazu müssen Netzwerkdaten in Blöcke unterteilt werden und es wird ein Port zugeordnet.
3 Vermittlung	Diese Ebene sorgt dafür, dass "Pakete" also <i>etikettierte Datenblöcke</i> über viele Rechner hinweg reisen können. Dazu werden unter anderem IP-Adressen verwendet. Man nennt sie auch die "Netzwerk-Schicht". Die Pakete aus Schicht 4 werden mit einer IP-Adresse versehen und so versandfähig gemacht.
2 Sicherung	Soll die Verbindung absichern, indem 3.-Schicht Daten in "Frames" also kleinere Blöcke geteilt und durch " Prüfsummen " mathematisch abgesichert werden.
1 Physisch	Kabelverbindungen oder "dumme" Geräte, denen die gesendeten Daten egal sind.

1 IP-Adressen und Subnetze

Aufbau

Jede IP-Adresse besteht aus 4 Zahlen, die jeweils im Bereich von 0-255 liegen. Beispielsweise **192.0.2.42**. Dies ist eine IPv4 Adresse, das heutzutage geläufigste Format, da die neuere Version 6 (also IPv6) des IP-Protokolls sich noch nicht durchsetzen konnte. IPv6 Adressen bestehen aus 8 Blöcken, die jeweils aus 4 Hexadezimalen Zahlen bestehen. Beispiel: **2001:0db8:85a3:0000:0000:8a2e:0370:7344**. So können statt 4.2 Milliarden Adressen in IPv4 665 Billionen Adressen dargestellt werden.

Letztlich dient jede IP Adresse der eindeutigen Identifikation eines Gerätes über mehrere Netzwerke hinweg. Folgende Adressen sind reserviert:

 1550519544492

Wichtig ist im Grunde nur, dass Adressen, die mit 192.168 anfangen, innerhalb eines privaten Netzes verwendet werden.

Standardvergabe

In einem Netzwerk steht die erste Adresse immer für das Netzwerk selbst und die letzte Adresse für den Broadcast. Wenn ein Gerät an diese IP sendet, wird es an alle Geräte gesendet. Beispiel:

Im Netzwerk 192.168.178.60 ist 192.168.178.0 das Netzwerk und 192.168.178.255 der Broadcast

Alle anderen IPs dazwischen können Geräten zugewiesen werden.

Subnetze

Um Netzwerke in kleinere Netze zu unterteilen, gibt es sogenannte Subnetze. Um das zu verstehen, sollte man sich die *IP-Adressen auf Bitebene* ansehen:

	Dezimal	Binär
IP-Adresse	24.98.0.233	00011000.01100010.00000000.11101001
Subnetzmaske	255.255.255.192	11111111.11111111.11111111.11000000

Die Subnetzmaske definiert von links nach rechts, welche Bits zum Netz gehören und welche zum Host. Der Netzanteil besteht aus den 1en, dieser ist innerhalb eines Netzwerkes festgelegt. Der Hostanteil variiert von Gerät zu Gerät, angezeigt durch die 0en.

In einem Netzwerk mit einer Subnetzmaske, die nur aus 0en und 255en besteht, ist das dann sehr einfach. Beispielsweise ein Netzwerk mit der IP 192.168.178.0 und der Subnetzmaske 255.255.255.0 könnte dann alle IPs von 192.168.178.0 - 192.168.178.255 enthalten.

Jedoch lassen sich die IP-Blöcke wie im Beispiel oben auch noch weiter unterteilen. Dort sind die ersten zwei Bits der letzten Zahl „markiert“ und gehören zum Netz. Das unterteilt die 256 möglichen Adressen in 4 weitere Subnetze. Wieso? Man zählt die 1en im Block von links und rechnet den Wert genauso aus, wie man es sonst tun würde. Nur von links. Das heißt:

$192 \Rightarrow 1100\ 0000 \Rightarrow 2^0 + 2^1 = 3$. Bedenke, dass 0 auch noch ein darstellbarer Wert ist, das heißt es sind 4 Werte darstellbar. Deshalb gibt es in diesem Netz 4 Subnetze. Jedes Subnetz erhält daher einen Bereich von $256/4 = 64$ Adressen. Also sehen die Bereiche so aus:

Nr	Adressbereich
0	24.98.0. 0 bis 24.98.0. 63
1	24.98.0. 64 bis 24.98.0. 127
2	24.98.0. 128 bis 24.98.0. 191
3	24.98.0. 192 bis 24.98.0. 255

Das geht auch über mehrere Blöcke hinweg:

	Dezimal	Binär
IP-Adresse	172.57.66.200	10101100.00111001.01000010.11001000
Subnetzmaske	255.255.240.0	11111111.11111111.11110000.00000000

Dieses Mal sind 16 Subnetze eingeteilt, jeweils mit 16×256 also 4096 Adressen. Beispielsweise das erste Netz verläuft dann von 172.57.**0**.0 bis 172.57.**15**.255

Netzklassen

Diese Subnetzmasken werden in grobe Klassen unterteilt: Klasse A, B und C.





Klasse A	Klasse B	Klasse C
Maske 255.0.0.0	Maske 255.255.0.0	Maske 255.255.255.0
16.777.216 Adressen	65.536 Adressen	256 Adressen

Bedenke, dass zum Beispiel 255.192.0.0 z.B. noch als Klasse A gelten würde, das dann nur weiter eingeteilt wurde. 255.255.128.0 würde zu Klasse B gehören etc.

Protokolle

Um in einem Netzwerk kommunizieren zu können, gibt es mehrere **Netzwerkprotokolle**, die das Format bestimmen, also *wie* die Daten übermittelt werden.

Eine Übersicht aller wichtigen Protokolle des OSI-Modells gibt es hier:

Name	OSI-Schicht	Zweck	Schaubild / Erklärung
IP	3 - Vermittlung	Etikettierung von Paketen	 ip_protocol
ICMP	3 - Vermittlung	Fehler- und Protokollfunktionen (Grundfunktionen)	Ermöglicht u.a. Befehle wie <i>Ping und Traceroute</i>
TCP	4 - Transport	Zuverlässiger Datenaustausch	Datenpakete werden vom Empfänger bestätigt, bevor die nächsten Daten gesendet werden. (Wird z.B. für Web genutzt)
UDP	4 - Transport	Schneller Datenaustausch	Datenpakete werden "drauf los" zum Empfänger gesendet, ob die Verbindung noch besteht muss zusätzlich geprüft werden (z.B. in Spielen) - Nicht nur Ende zu Ende
DHCP	5-7	Vergabe von IPs	 ip_protocol
DNS	5-7	Auflösen von Domains, z.B. von "google.de"	 ip_protocol
HTTP	5-7	Übertragen von Websites	 ip_protocol
HTTPS	5-7	Sicheres und Verschlüsseltes übertragen von Websites	 ip_protocol
FTP	5-7	Übertragen von Dateien	Wird u.a. verwendet um auf das Dateisystem eines Servers zuzugreifen. z.B. wenn man bei Nitrado einen Gameserver mietet.
ARP	2 - Sicherung	Auflösen von MAC-Adressen (Physische Adressen)	IP-Adressen können sich ändern, um also innerhalb eines lokalen Netzes zuverlässig die Pakete zuordnen zu können, benötigt man die MAC-Adresse. ARP sucht und findet diese.
QoS	3 - Router	Aufteilen der Bandbreite	Moderne Router (Aber auch z.B. Software oder andere Geräte) können dynamisch die Bandbreite zwischen Geräten oder Programmen aufteilen - Damit das Laden von Wikipedia nicht Netflix unterbricht!

Geräte

In diesem Kapitel sind einige verschiedene Geräte aufgetaucht, insbesondere im OSI-Modell, die aber selten wirklich erklärt wurden. Deshalb hier eine kurze Liste der Netzwerkgeräte und was sie tun:

- **Kabel** übertragen Daten.
- **Repeater** verbinden sich meist über Wifi mit dem Router und fungieren selbst als Access-Point. Sie leiten die Anfragen dann einfach an den Router weiter.
- **Hubs** nutzt man heute nicht mehr. Sie waren eine Art Verteiler für Daten, wo man mehrere Kabel hineinstecken konnte um mehr als nur 2 PCs zu verbinden. Allerdings werden bei einem Hub alle Daten an alle Teilnehmer verschickt - Anstatt wie bei einem Switch genau zuzuordnen. Das sorgt für hohe Netzauslastung und unnötigen Verkehr.
- **Layer-2 Switches** sind "einfache" Switches, d.h. wie ein intelligenter Hub, der aber die MAC-Adressen der Teilnehmer kennt und dementsprechend weiterleitet.
- **Layer-3 Switches** sind "riesige / komplexere" Switches, oft verwendet in Unternehmen oder z.B. unserer Schule. In diesen Hochleistungs-Switches ist meistens auch ein Router integriert, sodass die angeschlossenen Geräte voneinander getrennt werden können. So ein Switch hat teilweise mehr Anschlüsse als ein Netz IPs hätte - Daher die Router-Funktionalität.
- **Router** stellen eine Verbindung zwischen mehreren Netzen her. Dadurch sind sie eine Art "Tor zur Außenwelt" - Sie leiten die Anfragen dann weiter.
- **Bridges** sind auch alt und werden nicht mehr verwendet. Sie funktionieren ähnlich wie ein Switch, nur mit weniger Anschlüssen - Konnten aber damals auch Architekturen wie Ethernet oder Token Ring verbinden. Heute nimmt man lieber Switches oder Router, weil meist sowieso das MAC-Verfahren verwendet wird.
- **Modem** ist ein altes Gerät, das Internet-Signale in (hörbare) Töne für die Übertragung durch die Telefon-Leitung umwandelt. Vor dem DSL-Ausbau sehr verbreitet. In Filius wird damit die Brücke zwischen mehreren Rechnern simuliert.



Filius Lernzettel

Die Lernsoftware **Filius** dient der Visualisierung von Netzwerken und dessen Konfiguration.

⚠ Zum **Netzwerk-Teil** geht es [hier](#)!

Inhalt

1. 🧠 [Interface](#)
2. 💻 [Rechner](#)
3. 🔌 [Switch](#)
4. 📶 [Router](#)
5. 📡 [Modem](#)



Interface

Simulation

- Es ist möglich, zwischen dem "🔌 Baumodus" und dem "▶ Simulationsmodus" zu wechseln.
 - Der "🔌 Baumodus" dient dem Platzieren von Geräten, wie auch dem konfigurieren der Netzwerkoptionen dieser Geräte.
 - Der "▶ Simulationsmodus" ermöglicht den Zugriff auf die Software der Geräte und startet die Geräte. Man zieht außerdem, wann wo Daten fließen.

- Die Prozentanzeige ermöglicht die Simulationsgeschwindigkeit zu regeln.

Aufbau

- Im Hauptfenster können die Geräte platziert werden, ein Doppelklick auf ein Gerät öffnet dessen Konfiguration.
- Der "Bleistift" ermöglicht das Kommentieren und Strukturieren des Netzwerkes.

Konfiguration

- Für jedes Gerät öffnet sich am unteren Rand ein Fenster mit allen Einstellungsmöglichkeiten.



Rechner

Rechner, bzw. auch **Notebooks** (*Die Funktionalität ist dieselbe.*) können folgende Eigenschaften haben:



1550586572144

Option	Beschreibung
Name	Die Namen der Geräte werden im Hauptfenster angezeigt. Sinnvoll, um klar den Zweck eines Gerätes zuordnen zu können.
MAC-Adresse	Kann nicht geändert werden! - Hardwaregebundene Adressen, um IP-Adressen (z.B. durch DHCP) zuordnen zu können.
IP-Adresse	<i>Kann entweder manuell eingetragen werden oder per DHCP zugeordnet werden.</i> ⚠ Wichtig: Um ohne Gateway kommunizieren zu können, müssen die IP-Adressen im selben Subnetz liegen! (Siehe Netzwerktechnik)
Netzmaske	<i>Wird im Falle eines DHCPs automatisch übermittelt.</i> Gibt an, mit welchem Adressbereich das Gerät kommunizieren kann. (Siehe Netzwerktechnik)
Gateway	<i>Wird im Falle eines DHCPs automatisch übermittelt.</i> Gibt an, wohin Daten gesendet werden sollen, wenn der Empfänger nicht im selben (Sub-)Netz liegt.
DNS	<i>Wird im Falle eines DHCPs automatisch übermittelt.</i> Gibt an, wo die IP-Adressen zu Domains (z.B. "google.de") nachgeschaut werden sollen.

Software

Auf jedem PC / Notebook können folgende Programme installiert werden: (*Ein paar sind ausgelassen, die nicht verwendet werden.*)

Name	Beschreibung
Befehlszeile	Per Befehl können vor allem (z.B. durch <i>ping</i> oder <i>tracert</i>) Verbindungen getestet werden.
Bildbetrachter	Im Dateisystem hinterlegte Bilder können hiermit angezeigt werden.
Datei-Explorer	Ermöglicht den <i>Zugriff auf das Dateisystem</i> . Es ist auch möglich, Dateien in Filius zu importieren.
DNS-Server	Ergänzt die <i>DNS-Funktionalität</i> in einem Rechner. Die IP des Rechners muss dann als DNS-Server anderorts eingetragen werden.
Firewall	Ermöglicht das Blockieren von Ports auf dem System.
Text-Editor	Ermöglicht das Editieren von Dateien im Dateisystem.
Webserver	Ergänzt <i>HTTP-Server-Funktionalität</i> im Rechner. Die im Ordner "webserver" hinterlegten Dateien werden über die IP-Adresse fürs Netzwerk zugänglich gemacht.
Webbrowser	Aufrufen von <i>Html-Seiten</i> über das Netzwerk.

Webserver

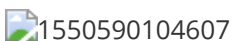
Jeder Webserver muss über das Fenster gestartet werden. Dort können auch "Virtuelle Hosts" aktiviert werden.

Im **Dateisystem** sehen die Server so aus:

```
> webserver
> index.html
> splashscreen-mini.png
```

- Die *Index*-Datei wird grundsätzlich immer aufgerufen, wenn in der Adresszeile keine spezielle Datei angegeben ist. Ein Aufruf von `http://192.168.0.10/datei.html` öffnet `datei.html`. Wenn in der URL keine Datei angegeben ist (`http://192.168.0.10/`), wird immer automatisch die `index.html` Datei aufgerufen.
- Die `splashscreen-mini.png` Datei wird in der HTML Datei verwendet, auch das ist möglich.

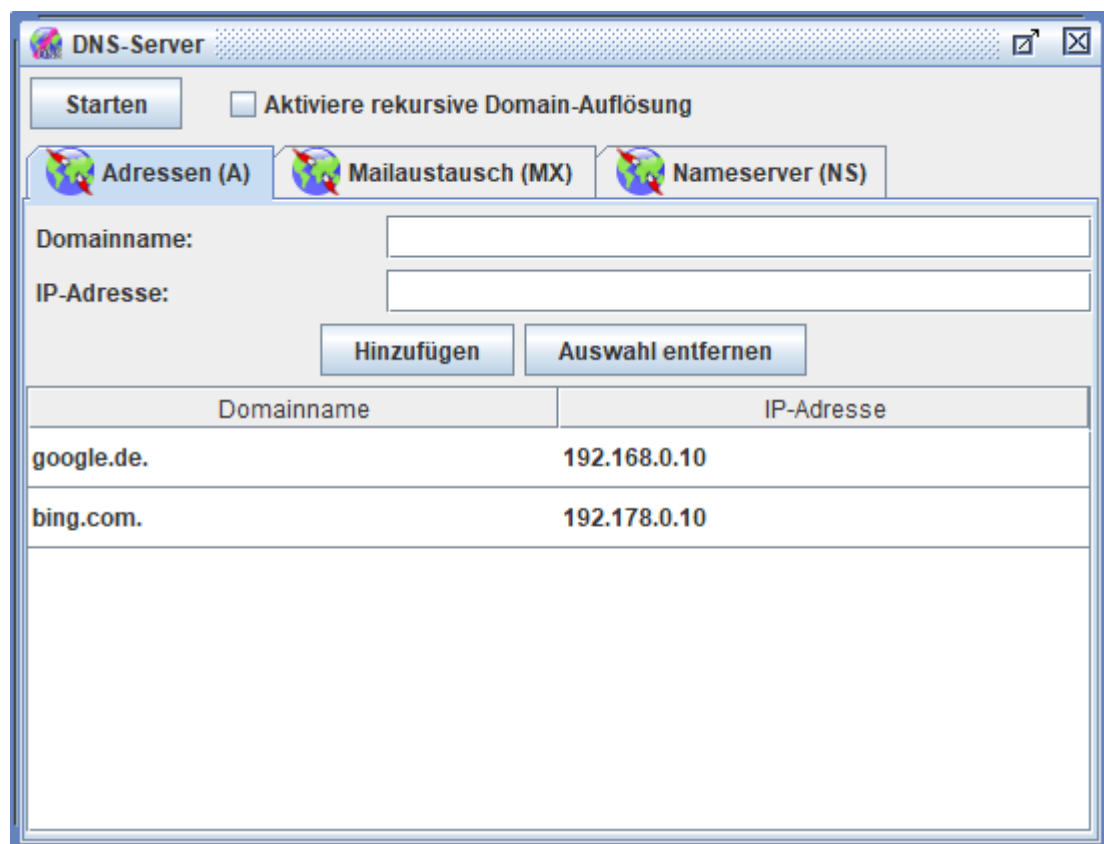
Es können außerdem sogenannte **virtuelle Server** erstellt werden. Dazu muss in der Software einfach der Haken gesetzt werden. Dann können zusätzliche Websites konfiguriert werden:



In diesem Beispiel wird über den Link `http://192.168.0.10/yey` die Website `webserver/101/index.html` aufgerufen. Oder wahlweise über `http://192.168.0.10/yey/web.html` die Seite `webserver/101/web.html`. Das Dateisystem könnte mit virtuellen Server z.B. so aussehen:

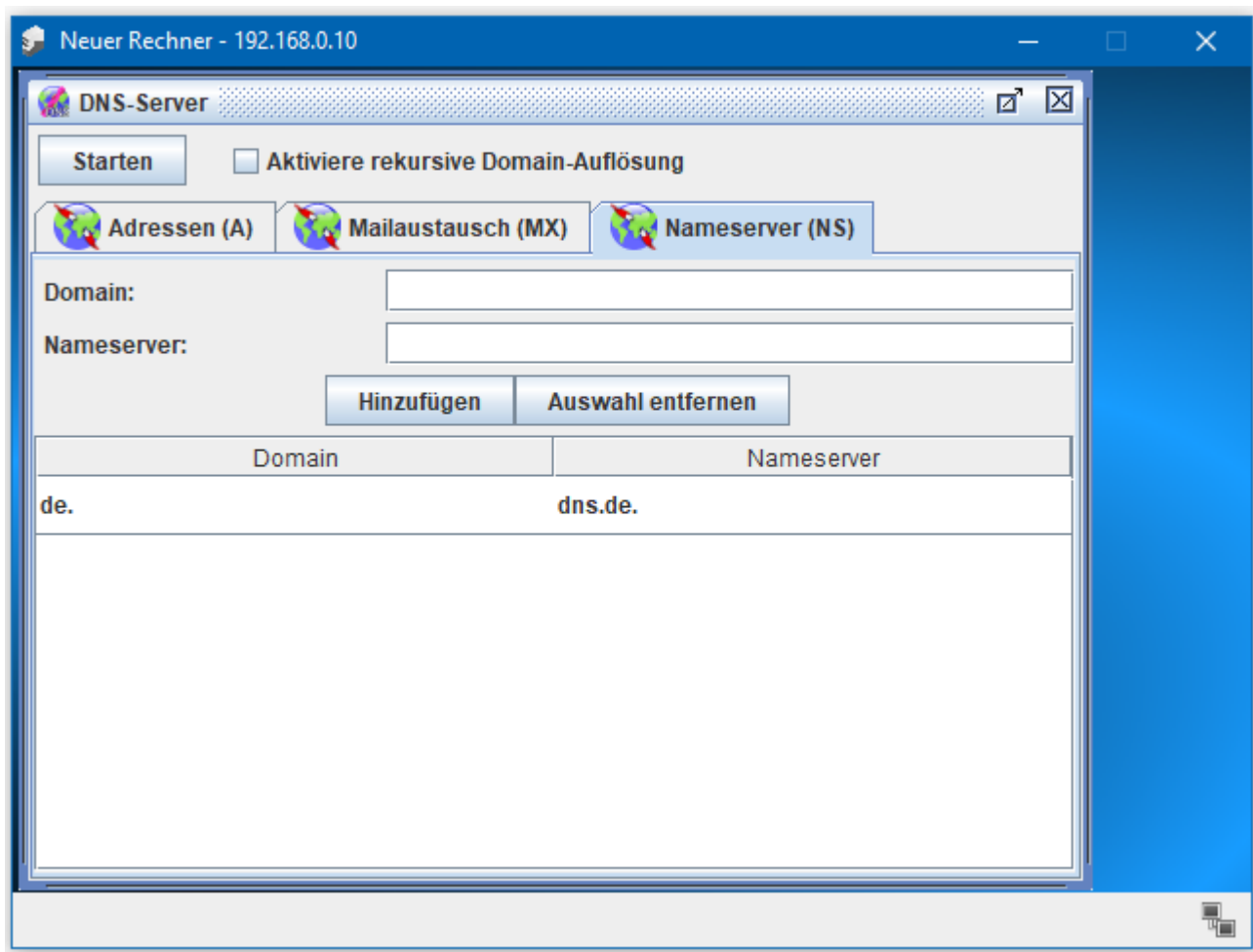
```
> webserver
> index.html
> splashscreen-mini.png
> doogle
> index.html
> suche.html
> logo.png
> github
> index.html
> awesome.html
```

DNS-Server



Auch DNS-Server müssen zunächst gestartet werden.

Jeder DNS-Server enthält eine Liste von Zuweisungen. Andere PCs können, wenn dort dieser DNS-Server als IP eingetragen wurde, die Domains zu IPs auflösen lassen.



Dies ist der Nameserver-Tab, denn es wäre unpraktisch alle Domains wieder und wieder in jeden DNS-Server eintragen zu müssen. Daher kann man hier auf andere DNS-Server verweisen, an die unbekannte Domainanfragen weitergeleitet werden sollen. Die Domain `.` steht für "Alle Anfragen", hier werden alle `.de`-Domains weitergeleitet. Der Nameserver `dns.de` muss bei den Adressen auch noch angegeben werden.

Switch

Der sogenannte **Switch** in Filius ermöglicht das Verbinden von vielen Geräten an dieselbe Leitung. An einen Switch kann eine nicht begrenzte Anzahl an Geräten angeschlossen werden, jede Verbindung selbst (per Kabel) ist aber trotzdem Ende-zu-Ende. Der Switch verteilt die Daten dann jeweils an die richtigen MAC-Adressen weiter.

Router

Ein **Router** hat eine feste Anzahl an Anschlüssen, zwischen denen er dann vermitteln kann. In Filius heißen Router "*Vermittlungsrechner*". Ein Router vermittelt zwischen mehreren Netzwerken, jeder der vorher konfigurierten Anschlüsse muss daher in einem anderen (Sub-)Netz liegen.

Konfiguration

- Es gibt einen Haupttab, in dem *Name*, *Gateway*, *Firewall* und *Anschlüsse* konfiguriert werden können.
 - Außerdem kann *Automatisches Routing* aktiviert werden, wenn es deaktiviert ist, tritt die *Weiterleitungstabelle* in Kraft. (Dazu unten mehr)

- Für jeden Anschluss können folgende Einstellungen gemacht werden:

Name	Wirkung
IP-Adresse	Der Router ist in jedem Netz als Gerät verfügbar und kann von den dortigen Geräten als <i>Gateway</i> angegeben werden. Weil der Router als Gerät im Netz ist, braucht er für dieses Netz auch eine IP.
Netzmaske	Die Netzmaske des Netzwerks, damit feststeht, welche Geräte im Netz erreichbar sind.
MAC-Adresse	<i>Kann nicht verändert werden.</i> - Ist nur wichtig, um z.B. Anfragen in Switches nachzuvollziehen.

Weiterleitungstabelle


In dieser Tabelle können, wenn die Option *Automatisches Routing* deaktiviert ist, Weiterleitung über mehrere Netze hinweg eingestellt werden. Dies ist nützlich, wenn z.B. entferntere Netze sonst nicht angesteuert werden können, weil sie hinter mindestens einem weiteren Router liegen.

Es kann folgendes eingestellt werden:

Ziel	Netzmaske	Nächstes Gateway	Über Schnittstelle
IP-Adresse des Netzes (oder Gerätes), an das gesendet werden soll.	Subnetzmaske des Ziels.	Ein Router, an den die Daten weiter gesendet werden sollen, wenn sie zu dem "Ziel" sollen.	Die IP-Adresse des gewünschten Ausgangs am eigenen Router.

Tip: Nimm den Haken bei "Alle Einträge anzeigen" heraus, um nur die wichtigen Einträge in der Tabelle zu sehen.

Firewall

 Konfiguration der Firewall

Netzwerkschnittstellen
 Firewall-Regeln

☒ Firewall aktivieren
 Erst durch Aktivieren der Firewall werden die Firewall-Regeln angewendet. Bei deaktivierter Firewall findet keine Filterung statt.

☒ ICMP-Pakete filtern
 Alle ICMP Pakete, z.B. Ping-Anfragen, werden von der Firewall verworfen.

☒ nur SYN-Pakete verwerfen
 Neue Verbindungsanfragen zu gesperrten Ports werden verworfen.
 Der Rückkanal für erlaubte Verbindungsanfragen wird akzeptiert.

In der Übersicht kann die Firewall aktiviert werden und es gibt folgende Einstellungen:

- "ICMP-Pakete filtern" => Ob z.B. Ping anfragen in der Firewall hängen bleiben sollen
- "nur SYN-Pakete verwerfen" => Öffnet automatisch den Rückkanal von erlaubten Anfragen.

ID	Quell-IP	Netzmaske	Ziel-IP	Netzmaske	Protokoll	Port	Aktion?
1					TCP		verwerfen

Hier können IP und Subnetzmaske jeweils aus dem Quellnetz und Zielnetz eingetragen werden - Frei lassen zum Aktivieren von allen. Die Spalten sollten sich selbst erklären.

Oben kann die Standardaktion eingetragen werden, also was ohne Regel passieren soll.

Wichtig ist noch, dass Regeln höher in der Liste auch höhere Priorität haben.

Modem

Ein **Modem** ist immer eine Brücke über das *reale Netzwerk* (Außerhalb von Filius) zu einer anderen Instanz zu Filius. Dazu muss im Modem folgendes angegeben werden:

Option	Beschreibung
Name	Dient der Übersichtlichkeit.
Auf Verbindung warten	Eins der beiden Modems muss diese Option aktivieren. Dieses Modem wird zum "Host" und wartet auf die Verbindung.
IP-Adresse	Die reale IP-Adresse des PCs, auf dem Filius mit dem anderen Modem läuft.
Port	Ein Port, auf den sich zwischen beiden Modems geeinigt wurde. Dadurch können mehrere Modem-Verbindungen gleichzeitig bestehen.

Grundbegriffe der IT

Zeichen und Daten

- **Informationen** sind Kenntnisse und Wissen, das in der Informatik durch **Zeichen** dargestellt wird.
- Ein **Zeichen** ist ein Element aus einem Zeichenvorrat z.B. *Buchstaben, Ziffern, Steuerzeichen...*
- Eine einzelne **Zeichenfolge** wird als **Nachricht** bezeichnet
- Wenn die Zeichenfolgen eine zu verarbeitende Information enthalten, dann nennt man sie **Daten** - z.B. *aufgezeichnete Luftfeuchtwerte über 24h hinweg*

Signalarten

Signale (siehe [STEUERN und REGELN](#)) übertragen Informationen. Diese Signale können wahlweise einer **analogen** oder **digitalen** Natur sein.

- **Analoge** Signale sind **stufenlos**, daher viel genauer. z.B. *Mechanische Übertragung, Mikrophon-Sprechwechselspannung, Wasserpegel*
- **Digitale** Signale sind **abgestuft**, sie können innerhalb eines *Wertebereichs* nur bestimmte (*diskrete*) Signalwerte annehmen. z.B. *Temperaturstufen eines Backofens, Lautstärkestufen am Handy, gerundete Messwerte*

- **Binäre Signale** sind eine Unterart der Digitalsignale. Sie können nur 0 oder 1 annehmen bzw. Low/High oder Aus/An

Sowohl der Wertebereich, wie auch der Zeitbereich (X- / Y-Achse) können entweder **kontinuierlich** (stetig, smoooooth) oder **diskret** (durch Abstände getrennt) sein.

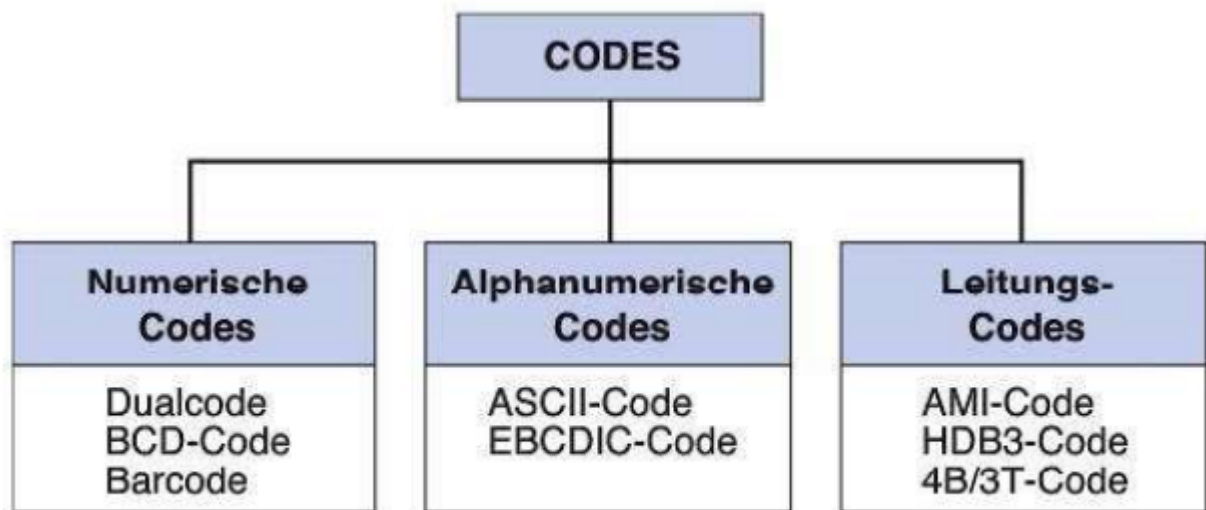
- Ein **wert- und zeitkontinuierliches Signal** kann zu jedem Zeitpunkt jeden beliebigen Signalwert annehmen.
- Ein **wertdiskretes zeitkontinuierliches Signal** kann zu jedem Zeitpunkt nur bestimmte Werte (in einem Bereich mit festen Abständen) annehmen.
- Ein **wertkontinuierliches zeitdiskretes Signal** kann zu bestimmten Zeitpunkten (in einem Bereich mit festen Abständen) jeden beliebigen Wert annehmen.
- Ein **wert- und zeitdiskretes Signal** kann nur bestimmte Werte zu bestimmten Zeiten annehmen.

□ CODES

In der Informationstechnik werden, wie auch in der zwischenmenschlichen Kommunikation, gewisse Konventionen bzw. Systeme bestehend aus Zeichensätzen verwendet.

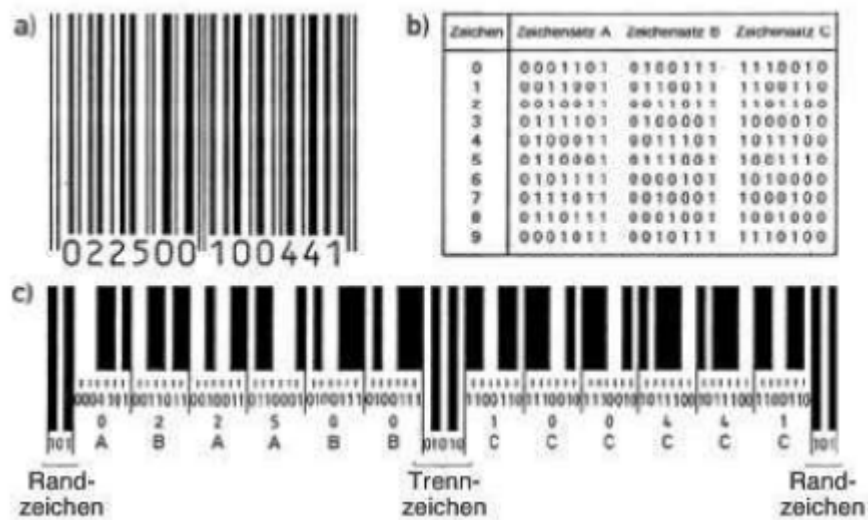
Ein **Code** ist eine Vorschrift für die eindeutige Zuordnung der Zeichen eines Zeichensatzes zu den Zeichen eines anderen Zeichensatzes.

Code-Arten



- **Numerische Codes** stellen Zahlen dar. *Verwendung z.B. bei Postleitzahlen, Artikelnummern o.ä.*
- **Alphanumerische Codes** stellen Zahlen, Buchstaben und Steuerzeichen dar. *Verwendung z.B. innerhalb von Websites, Programmen, Betriebssystemen, etc.*
- **Leitungscodes** stellen binäre Signale als Digitalsignale dar. *Verwendung z.B. in Kupferleitung, Glasfaser etc.*

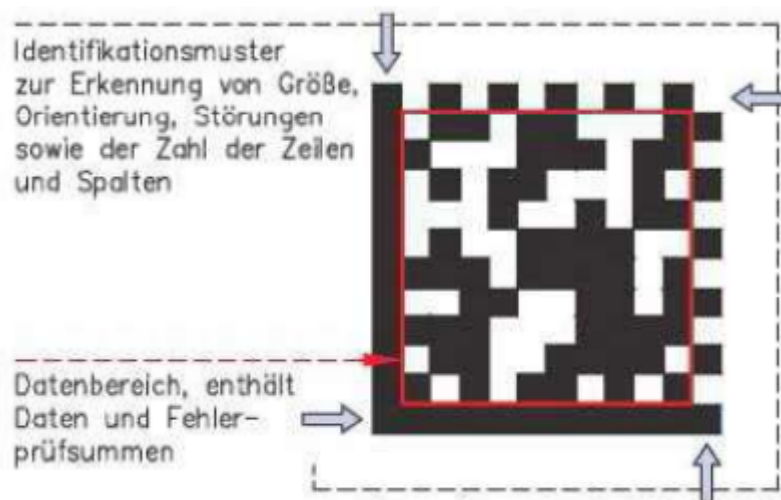
Barcodes



Barcodes sind **binäre Zeichencodes**, die vor allem der *Kennzeichnung von Waren* im Handel oder Lager dienen. In diesem Dokument wird der **EAN-Code** verwendet, (kurz European-Article-Numbering,) der aus 2 Hälften mit je 6 Dezimalziffern besteht.

- Eingegrenzt wird der Code durch jeweils ein **Randzeichen** (101), die vor allem der Kalibrierung des Scanners dienen
- In der Mitte gibt es ein **Trennzeichen** (01010) mit demselben Zweck, außerdem unterteilt es die beiden unterschiedlichen Hälften des Codes
- Die Dezimalzahlen sind jedoch nicht wie normale Binärzahlen kodiert, sondern werden durch drei Zeichensätze dargestellt. (siehe Tabelle b) *Die linke Hälfte besteht aus den Codes ABAABB und die rechte aus C*

2D-Codes

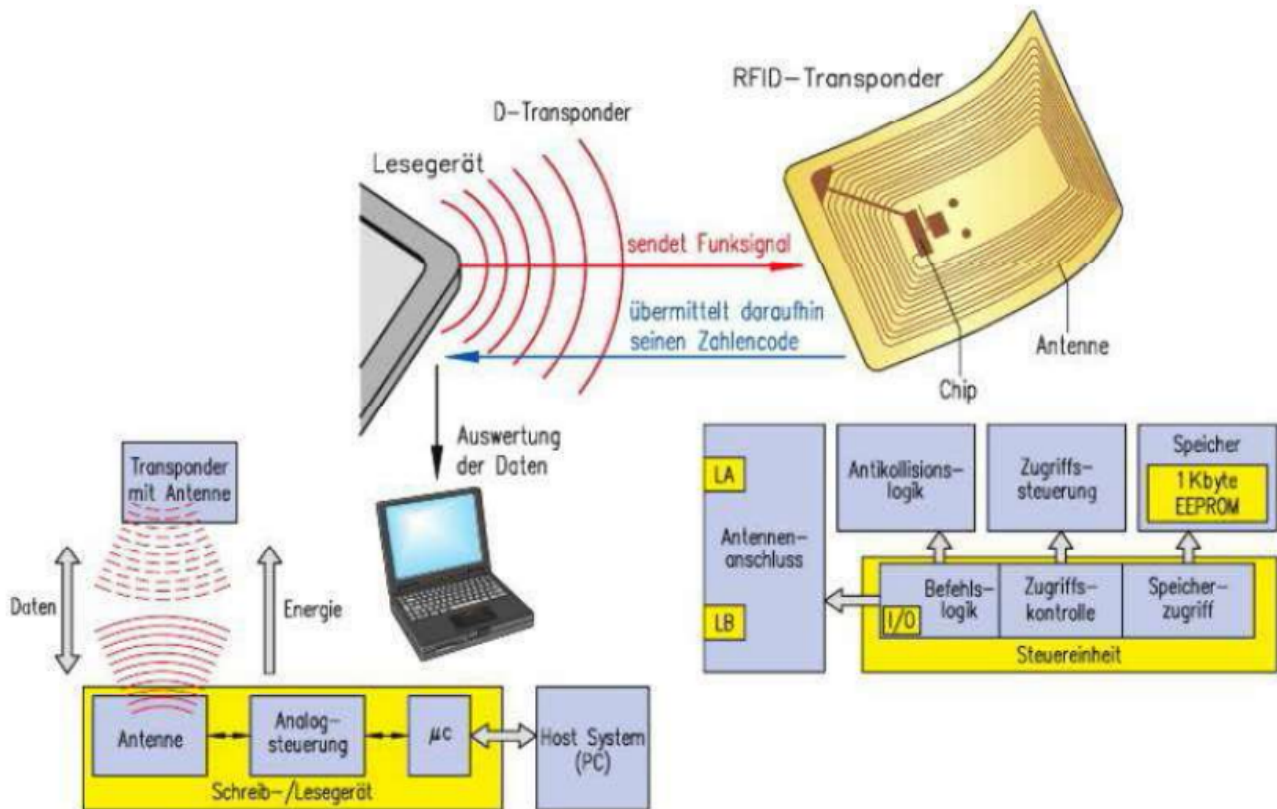


Ein Barcode ist ja im Prinzip ein Eindimensionaler Code. Passend dazu gibt es auch **zweidimensionale Codes**. In diesem Beispiel wird der **Data-Matrix-Code** (DMC) verwendet.

In der aktuellsten Version gibt es eine eingebaute Fehlerkorrektur (*ECC - Error Checking Correction*), dazu wird wie im Bild zu sehen die unterste Reihe für **Prüfsummen** verwendet. Mit den Prüfsummen lässt sich mathematisch die Korrektheit der oberen Ziffern bestimmen und ggf. lassen sich sogar Teile Rekonstruieren (bis zu 25%).

- Es gibt ein **Identifikationsmuster**, sprich einen Rahmen, anhand dessen sich die Ausrichtung und Größe des Codes erkennen lässt.
- Ein DMC besteht aus **mehreren Datenregionen**, welche je nach Symbolauswahl z.B. 88 numerische oder 64 alphanumerische Zeichen speichern können. Im Bild zu sehen ist eine dieser Datenregionen, z.B. auf Briefumschlägen sind 4 davon, die zusammen ein größeres DMC bilden.

RFID



RFID (Radio Frequency Identification) ist eine Technik zur **berührlosen Identifikation**.

Ein RFID-System funktioniert, indem ein **Funksignal** vom Lesegerät aus gesendet wird, durch das Prinzip der Induktion wird in der Antenne des Transponders eine sehr geringe Spannung erzeugt, die aber ausreicht, um den Zahlencode im Chip abzufragen und zurück zu senden.

Es gibt allerdings auch Transponder, wie z.B. in manchen Modekaufhäusern, die eine kleine Batterie besitzen, da die Sendeleistung sonst nicht für größere Distanzen ausreichen würde.

Steuern und Regeln

Maschinen und Anlagen werden mithilfe der **Steuerungs-** und **Anlagentechnik** automatisiert. Auf dieser Grundlage gibt es grundsätzlich zwei Möglichkeiten:

- Beim **Steuern** wird eine *feste Eingabe* getätigt, die dann von der Maschine umgesetzt werden soll.
- Beim **Regeln** wird ein *Soll-Wert* festgelegt und ein *Ist-Wert* gemessen, der dann automatisch korrigiert wird.

Der *große Unterschied* ist, dass beim Steuern *keine Fehlerkorrektur* stattfindet. Man dreht den Strom auf, aber ob er ankommt ist unklar. Ein *Regler* ist in der Lage die Funktionalität zu gewährleisten und kann teilweise sogar Fehler ausgeben.

⚠ **Wichtig:** Die folgenden Abschnitte thematisieren alle die **Steuerungstechnik** und nicht die Regelungstechnik!

Das EVA-Prinzip

- **Eingabe** von Signalen z.B. *durch Taster, Druckschalter und Sensoren*
- **Verarbeitung** der Signale z.B. *durch Verknüpfung in einem Relais*
- **Ausgabe** der Signale z.B. *an einem Antriebsmotor*

Dieses Prinzip lässt sich in der Steuerungstechnik fast immer anwenden, es findet sich aber auch in der Informatik wieder. Nützlich ist die Unterteilung vor allem für Planung und Analyse der Maschinen.

Grundbegriffe

- Ein **Signal** kann *mechanisch* oder *elektr(on)isch* sein und ist der Kern einer Steuerung.
- Ein **Signalgeber** erzeugt ein Signal z.B. *ein Knopf*
- Ein **Stellglied** entscheidet über die Veränderung z.B. *ein Steuergerät*
- Eine **Stellgröße** ist ein Wert, den das Stellglied zum steuern/regeln erzeugt z.B. *Spannung zwischen Steuergerät und Motor*
- Eine **Steuergröße** ist die gesteuerte Größe also z.B. *Wegstrecke eines Bohrers*
- Eine **Steuerstrecke** ist die Strecke zwischen Stellglied und dem zu steuernden Objekt - z.B. *Strecke zwischen Steuergerät und Bohrkopf*
- Eine **Steuerkette** bzw. ein Wirkungsablauf ist eine Gesamtübersicht, oft mit Pfeilen, über die Abläufe der Anlage

Steuerungsarten

Die Einteilung kann entweder nach der Signalverarbeitung oder nach der Programmierung unterschieden werden.

Einteilung nach Art der Signalverarbeitung

- **Verknüpfungssteuerungen** schalten nur, wenn *logische Bedingungen* erfüllt sind
- **Ablaufsteuerungen** schalten in einer bestimmten *Reihenfolge*
 - **Zeitabhängige** Ablaufsteuerungen werden z.B. *durch Nockenschaltwerke, Zeitrelais oder Taktgeber* bestimmt
 - **Prozessabhängige** Ablaufsteuerungen durchlaufen einen Abschnitt nach dem anderen. Ein Abschnitt beginnt erst, wenn der vorherige Abgeschlossen wurde.
Wenn Arbeitsschritte wenn Abschnitte den Wegen der Maschinen entsprechen, z.B. *bei einem Maschinentisch*, dann nennt man das **Wegplansteuerung**

Einteilung nach Art der Programmierung

- **Verbindungsprogrammierte Steuerungen** geben den Programmablauf durch Bauteile und deren Verbindungen fest vor.
- **Speicherprogrammierte Steuerungen (SPS)** legen den Programmablauf durch ein Programm fest.

Zahlensysteme bilden die Grundlage der IT, aber auch außerhalb dieses Themas sind Zahlensysteme sehr wichtig.

Ein Zahlensystem ist ein festgelegtes System, das numerische Werte mithilfe von uns bekannten Symbolen darzustellen versucht.

Ein **kleiner Überblick** über die Zahlensysteme:

Name	Symbole	Basis (Werte pro Ziffer)
Dual / Binär	0-1	2
Dezimal	0-9	10
Hexadezimal	0-9, A-F	16

Das **Dezimalsystem** ist das für uns Menschen normale System. Die 10 lässt sich an beiden Händen abzählen, deshalb ist es für uns Menschen so praktisch.

Das **Binärsystem** ist für Maschinen normal und praktisch. Ganz einfach weil der Strom dann entweder AN oder AUS sein kann.

Das **Hexadezimalsystem** wird beispielsweise gerne auf einer höheren Protokollebene verwendet, um z.B. Bandbreite zu sparen.

Begriffe

Es ist wirklich wichtig, sich im Rahmen dieser Systematiken **klar auszudrücken**.

- Eine **Ziffer** ist eine Stelle einer evtl. größeren Zahl. z.B. bei 312, sind sowohl 3, 2, 1 Ziffern.
- Eine **Zahl** besteht aus mehreren (oder einer) Ziffer(n).
- Ein **Wert** lässt sich in verschiedenen Zahlensystemen unterschiedlich ausdrücken. Im Zweifel wird der Wert im Dezimalsystem angegeben.

☪☪☪ könnte im Dezimalsystem 3 oder im Binärsystem 11 bedeuten.

Systematik

Jedes (hier behandelte) Zahlensystem hat ein paar gleiche grundlegende Regeln:

- Wenn sich ein Wert *nicht mehr durch die aktuelle Ziffernmenge ausdrücken lässt*, **fügt man** links **eine Stelle hinzu**.
- Die **Wertigkeit** aller hinzugefügten Ziffern lässt sich wie folgt berechnen:

$$\text{Basis}^s$$

s = Stelle [z.B. bei 4028 wäre die 2 auf Stelle 1 und die 4 auf Stelle 3]

Die Basis hängt vom System ab.

Zur einfachen Demonstration, hier ein Beispiel:

Dezimal	Binär	Hexadezimal
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Berechnung

Um Zahlenwerte **ins Dezimalsystem umzurechnen**, geht man so vor:

$$11011011 \Rightarrow 2^7 + 2^6 + 2^4 + 2^3 + 2^1 + 1 = 219$$

Rückwärts subtrahiert man nach und nach die größtmöglichen Ziffern von der Dezimalzahl und notiert sich das entsprechend:

$$219 - 128 = 91 - 64 = 27 - 16 = 11 - 8 = 3 - 2 = 1 - 1 = 0 \Rightarrow 11011011$$

128, 64, 16, 8, 2 sind jeweils die 2er Potenzen. Dementsprechend wird hier in das Binärsystem umgerechnet.

TDLR;

("Too long, didn't read")

Nutze einfach den **Windows-Taschenrechner** (Kategorie "Programmierer") in der Klausur!

Ergänzung:

Watermann gab uns eine leicht andere (meiner Meinung nach kompliziertere) Variante, die Zahlensysteme umzurechnen. Nur der Vollständigkeit halber, falls die Rechnung tabellarisch gefordert wird:

- Die zur Darstellung einer Zahl erforderlichen Ziffern werden von einer Markierung – dem **Komma** – ausgehend nebeneinander geschrieben und nummeriert. Links vom Komma stehen Zahlen ≥ 1 , rechts vom Komma stehen Zahlen < 1 .
- Jede Stelle hat einen eigenen **Stellenwert W**; er berechnet sich aus der **Basis B** des **Zahlensystems** und der Stellennummer n: Stellenwert vor dem Komma : $W = B^{n-1}$
- Stellenwert nach dem Komma: $W = B^{-n} = \frac{1}{B^n}$
- Die Basis des Zahlensystems ist gleich der Anzahl der verfügbaren Ziffern.
- Der Potenzwert einer Stelle ergibt sich durch Multiplikation der Ziffer mit dem Stellenwert.
- Der Zahlenwert ist die Summe aller Potenzwerte.
- Wird beim Hochzählen in einer Stelle die höchste Ziffer (im Dezimalsystem also die 9) erreicht, so wird im folgenden Schritt ein **Übertrag** von 1 in die nächsthöhere Stelle geschrieben und die hochgezählte Stelle beginnt wieder mit 0 (Bild 4.10).

Hexadezimalsystem				Dezimalsystem				Dualsystem				
16^3	16^2	16^1	16^0	10^3	10^2	10^1	10^0	2^4	2^3	2^2	2^1	2^0
4096	256	16	1	1000	100	10	1	16	8	4	2	1
			0				0					0
			1				1					1
			2				2				1	0
			3				3				1	1
			4				4			1	0	0
			5				5			1	0	1
			6				6			1	1	0
			7				7			1	1	1
			8				8		1	0	0	0
			9				9		1	0	0	1
			A			1	0		1	0	1	0
			B			1	1		1	0	1	1
			C			1	2		1	1	0	0
			D			1	3		1	1	0	1
			E			1	4		1	1	1	0
			F			1	5		1	1	1	1
		1	0			1	6	1	0	0	0	0
		1	1			1	7	1	0	0	0	1