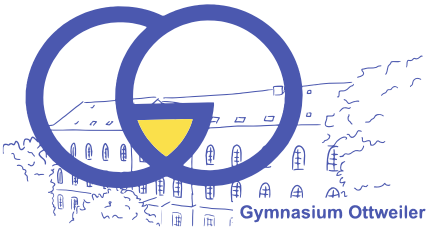
Kapitel 1

Turingmaschine, Algorithmus, Programm



Skript Informatik · Grundkurs

Die Informatik ist älter als der Computer. Das klingt nach einem Scherz, ist aber der Schlüssel zu diesem Kurs: Als 1936 geklärt wurde, was eine Maschine im Prinzip berechnen kann, gab es noch keine einzige. Gebaut wurde erst danach — und zwar nach Plänen, die dieser Klärung folgten.

Dieses Kapitel beginnt deshalb bei der Maschine, die nie gebaut wurde, und endet bei dem Werkzeug, das Sie in jedem weiteren Kapitel brauchen werden: dem Unterprogramm. Dazwischen ordnet es, was Sie aus der Einführungsphase mitbringen — und zwar in zwei Sprachen nebeneinander: **Python** und **Object Pascal** (Delphi/Lazarus). Beide sind im Saarland für das Abitur zugelassen, und wer dieselbe Idee zweimal geschrieben sieht, erkennt schneller, was an ihr die Idee ist und was bloß Schreibweise.

Leitidee: Zu den Marken am Seitenrand

Dieses Skript ist für den **Grundkurs** geschrieben; ohne Marke gilt also: verbindlicher Stoff nach dem Lehrplan Saarland vom Juli 2019. Was darüber hinausgeht, ist gekennzeichnet — **LK** steht nur im Lehrplan des Leistungskurses, **ZUSATZ** in keinem von beiden. Eine Marke an einer Überschrift gilt für den ganzen Abschnitt; einzelne Absätze tragen stattdessen einen schmalen Balken am Rand. Markiert ist nur, was *im Lehrplan* steht — was in einer Kursarbeit drankommt, entscheidet Ihre Lehrkraft.

Lernziele

Am Ende dieses Kapitels können Sie

- die Entstehung der Informatik aus dem Entscheidungsproblem in Grundzügen darstellen,
- den Aufbau einer Turingmaschine beschreiben und ihre Bestandteile benennen,
- eine gegebene Turingmaschine an einem Wort Schritt für Schritt ablaufen lassen,
- eine einfache Turingmaschine selbst angeben,
- die Church-Turing-These wiedergeben und erklären, warum sie keine beweisbare Aussage ist,
- die fünf Grundeigenschaften eines Algorithmus benennen und an Beispielen prüfen,
- das EVA-Prinzip auf eine Aufgabenstellung anwenden,
- Variablen, Datentypen und Zuweisungen in Python und Object Pascal sicher verwenden,
- Verzweigungen und alle drei Schleifenarten einsetzen und begründen, welche passt,
- Unterprogramme schreiben und dabei zwischen **Auftrag** (Prozedur) und **Anfrage** (Funktion) unterscheiden,
- Wert- und Referenzparameter unterscheiden und ihre Wirkung vorhersagen,
- erklären, warum lokale Variablen lokal sein sollen.

1.1 Woher die Informatik kommt

LK

> LK
>
> Die Abschnitte 1.1 und 1.2 gehören zum Themenfeld *Berechenbarkeit und ihre Grenzen* und stehen damit nur im Lehrplan des Leistungskurses. Für den Grundkurs sind sie nicht verbindlich — als Einstieg und zur Einordnung des Algorithmenbegriffs lohnen sie sich trotzdem.

Rechnen wollte man schon immer automatisieren. **Gottfried Wilhelm Leibniz** stellte 1673 das Modell einer Maschine vor, die alle vier Grundrechenarten beherrschen sollte — fertig wurde sie erst 1694. Er träumte von einer Sprache, in der sich Streitfragen durch Rechnen entscheiden lassen: *Calculemus* — lasst uns rechnen.

Gut anderthalb Jahrhunderte später entwarf **Charles Babbage** die *Analytical Engine*, eine Maschine aus Zahnrädern, die Programme von Lochkarten lesen sollte. Gebaut wurde sie nie, sie war für die Feinmechanik ihrer Zeit zu groß. Aber **Ada Lovelace** schrieb 1843 für diese Maschine auf, wie man die Bernoulli-Zahlen berechnet — ein Programm für einen Rechner, den es nicht gab, und das wohl erste der Geschichte. Sie bemerkte dabei als Erste, dass eine solche Maschine nicht nur Zahlen verarbeiten könnte, sondern alles, was sich als Zahl darstellen lässt: Musik zum Beispiel.

1.1.1 Hilberts Frage

1928 stellte der Mathematiker **David Hilbert** eine Frage, die wie reine Theorie klingt und die Informatik auslöste — das **Entscheidungsproblem**:

> Gibt es ein Verfahren, das für jede mathematische Aussage in endlich vielen Schritten entscheidet, ob sie allgemeingültig — und damit beweisbar — ist?

Hilbert rechnete mit *ja*. Er hielt die Mathematik für vollständig und entscheidbar und hielt es für eine Frage der Zeit, bis man das Verfahren fände. Sein berühmter Satz dazu: „Wir müssen wissen. Wir werden wissen."

Es kam anders. 1931 zeigte **Kurt Gödel**, dass jedes hinreichend starke formale System wahre Aussagen enthält, die es selbst nicht beweisen kann. Und 1936 beantworteten **Alonzo Church** und, unabhängig von ihm, **Alan Turing** das Entscheidungsproblem endgültig — mit *nein*.

Gymnasium Ottweiler

Leitidee: Das Problem hinter dem Problem

Um zu beweisen, dass es *kein* Verfahren gibt, muss man erst sagen, was ein Verfahren überhaupt ist. Für ein einzelnes Verfahren genügt es, es hinzuschreiben. Für die Aussage „es gibt keines" muss man über **alle denkbaren Verfahren** sprechen können.

Genau das war Turings eigentliche Leistung — nicht die Antwort, sondern der Begriff, der die Frage überhaupt stellbar machte.

1.1.2 Turings Antwort

Alan Turing war 24 Jahre alt, als 1936 seine Arbeit *On Computable Numbers, with an Application to the Entscheidungsproblem* erschien. Darin tat er etwas Ungewöhnliches: Er beobachtete, was ein **Mensch** tut, der mit Papier und Bleistift rechnet.

Ein solcher Mensch schaut auf eine Stelle des Papiers. Er erkennt dort ein Zeichen. Je nachdem, was er sieht und in welcher Verfassung er gerade ist — welchen Schritt der Rechnung er also im Kopf hat —, schreibt er etwas hin und wendet den Blick weiter. Mehr passiert nicht. Turing strich alles Überflüssige weg und behielt genau diesen Kern übrig. Das Ergebnis heißt heute **Turingmaschine**.

Mit diesem Begriff konnte er beweisen, was vorher nicht einmal formulierbar war: dass es Probleme gibt, die keine Maschine löst. Eines davon ist das **Halteproblem** (Abschnitt 4.4.4), und aus ihm folgt die Antwort auf Hilberts Frage.

1.1.3 Und dann wurde gebaut

Die Maschinen kamen nach der Theorie. **Konrad Zuse** stellte am 12. Mai 1941 in Berlin die **Z3** fertig, den ersten funktionsfähigen programmgesteuerten Rechner — er wusste von Turings Arbeit nichts. In den USA entstand bis 1946 der **ENIAC**, und **John von Neumann** beschrieb 1945 jene Bauweise, nach der praktisch jeder Rechner bis heute arbeitet: Programm und Daten im selben Speicher (Kapitel 6).

Turing selbst erlebte das kaum noch. Im Zweiten Weltkrieg war er in Bletchley Park an der Entzifferung der ENIGMA beteiligt (Kapitel 8). 1952 wurde er wegen seiner Homosexualität verurteilt, 1954 starb er. Die höchste Auszeichnung der Informatik, vergeben seit 1966, trägt seinen Namen.

Merke

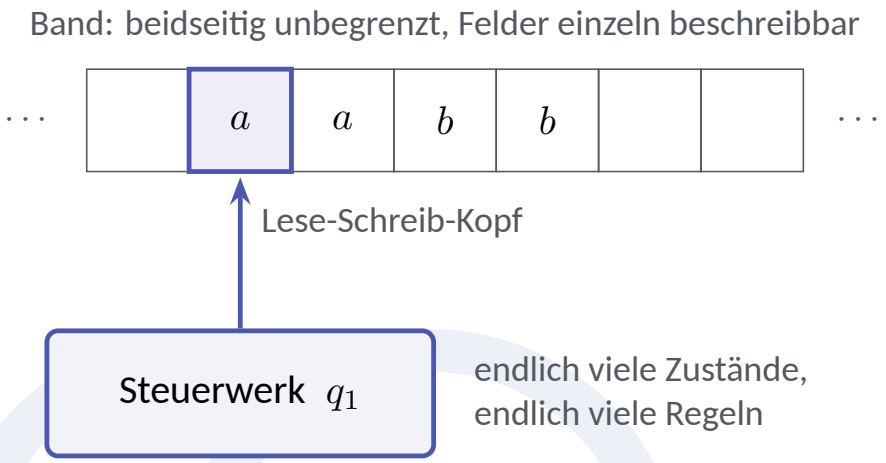
Die Informatik entstand nicht aus dem Bau von Rechnern, sondern aus der Frage nach den **Grenzen** des Rechnens. Deshalb ist sie eine Wissenschaft und nicht nur eine Technik: Ihre Kernaussagen gelten für jede Maschine — auch für die, die noch niemand gebaut hat.

1.2 Die Turingmaschine

LK

> LK
>
> Auch dieser Abschnitt ist LK-Stoff (*Berechenbarkeit und ihre Grenzen*). Im Grundkurs genügt die Grundidee: Es gibt ein formales Modell dafür, was „berechenbar" heißt.

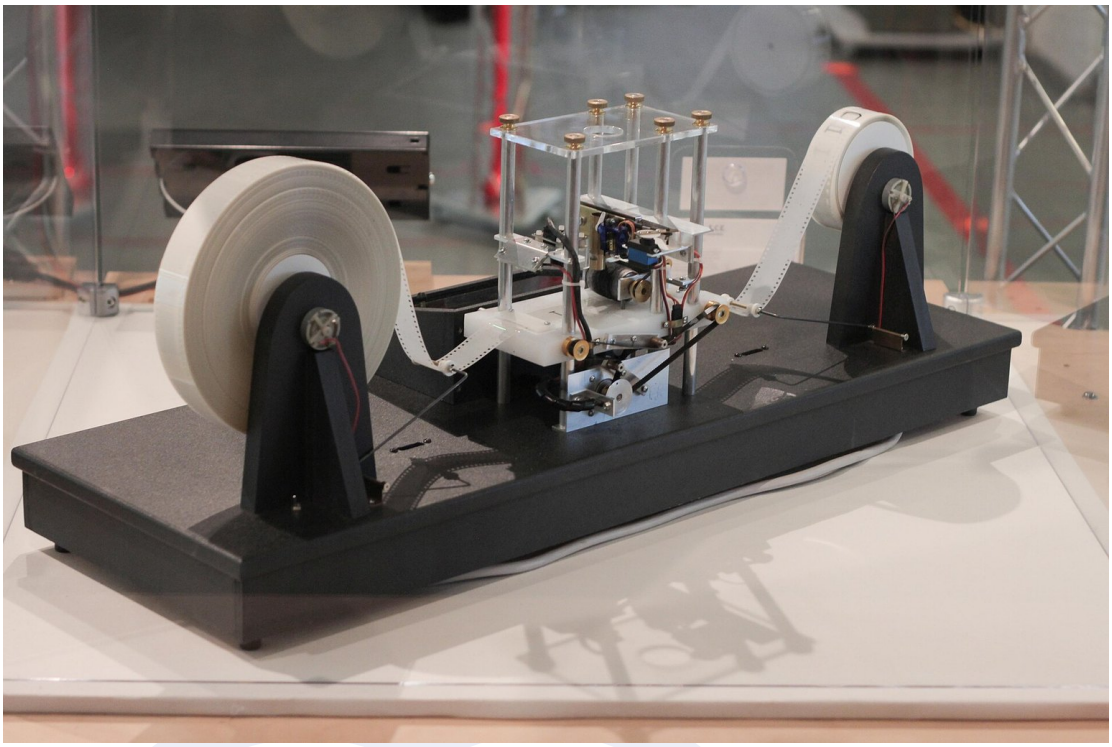
Die Maschine, die Turing beschrieb, ist absichtlich armselig. Sie besteht aus drei Teilen.



Aufbau einer Turingmaschine.

Gebaut hat man sie trotzdem — nicht, weil es nötig wäre, sondern weil es schön ist. Mike Davey stellte 2010 eine funktionierende Turingmaschine her: Links und rechts die Rollen mit dem Band, in der Mitte der Lese-Schreib-Kopf auf seinem Schlitten, darunter das Steuerwerk. Das Band ist tatsächlich unbegrenzt — es wird einfach nachgelegt.

Gymnasium Ottweiler



Turingmaschine, gebaut von Mike Davey (2010). Foto von 2012: Rocky Acosta, CC BY 3.0, via Wikimedia Commons.

Vergleichen Sie das Foto mit der Zeichnung darüber: Jedes Bauteil der Definition ist darin wiederzufinden — und nichts darüber hinaus.

Definition: Turingmaschine

Eine **Turingmaschine** besteht aus

- einem **Band**, das in Felder geteilt und nach beiden Seiten unbegrenzt ist. Jedes Feld trägt ein Zeichen oder ist leer (□).
- einem **Lese-Schreib-Kopf**, der auf genau einem Feld steht. Er kann das Zeichen dort lesen, es überschreiben und sich um ein Feld nach links oder rechts bewegen.
- einem **Steuerwerk**, das sich in einem von **endlich vielen Zuständen** befindet.

Das Verhalten legen **Regeln** fest. Eine Regel hat die Form

$$(\text{Zustand},\ \text{gelesenes Zeichen}) \longrightarrow (\text{zu schreibendes Zeichen},\ \text{Bewegung},\ \text{neuer Zustand})$$

mit der Bewegung L (links), R (rechts) oder N (stehen bleiben). Gibt es für die aktuelle Lage keine Regel, hält die Maschine an.

Merke

Endlich sind nur Zustände und Regeln. **Unbegrenzt ist allein das Band.** Genau darin liegt der Unterschied zum endlichen Automaten aus Kapitel 5: Dieser hat kein Band und muss sich alles im Zustand merken — und weil er nur endlich viele Zustände hat, kann er sich nur endlich viel merken.

1.2.1 Eine erste Maschine: unär addieren

Zahlen werden als Striche geschrieben: 3 ist 111. Auf dem Band steht eine Aufgabe wie 11+111, und am Ende soll dort das Ergebnis stehen.

Der Trick ist hübsch: Man ersetzt das + durch einen Strich — dann stehen alle Striche beieinander, aber es ist einer zu viel. Also läuft die Maschine ans Ende und löscht dort einen.

Zustand	liest	schreibt	Bew.	neu	Bedeutung
q_0	1	1	R	q_0	über die ersten Striche laufen
q_0	+	1	R	q_1	Pluszeichen zum Strich machen
q_1	1	1	R	q_1	weiter bis ans Ende
q_1	□	□	L	q_2	Ende erreicht, einen zurück
q_2	1	□	N	q_a	letzten Strich löschen, fertig

Beispiel: Ablauf für 11+111

Der Kopf steht jeweils auf dem eingerahmten Feld.

```
q0   [1] 1  +  1  1  1
q0    1 [1] +  1  1  1
q0    1  1 [+] 1  1  1
q1    1  1  1 [1] 1  1
q1    1  1  1  1 [1] 1
q1    1  1  1  1  1 [1]
q1    1  1  1  1  1  1 [□]
q2    1  1  1  1  1 [1] □
qa    1  1  1  1  1 [□] □
```

Acht Schritte, und auf dem Band stehen fünf Striche: $2 + 3 = 5$.

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Turingmaschine**.

1.2.2 Eine Maschine, die ein Automat nicht sein kann

In Kapitel 5 wird gezeigt, dass kein endlicher Automat die Wörter $a^n b^n$ — also ab, aabb, aaabbb, … — erkennen kann: Er müsste beliebig weit zählen, und dazu reichen endlich viele Zustände nicht.

Eine Turingmaschine schafft es, weil sie sich auf dem Band Notizen machen kann. Ihre Idee ist die eines Kindes, das Äpfel und Birnen paarweise zuordnet: **immer ein a und ein b zusammen durchstreichen.** Bleibt am Ende nichts übrig, waren es gleich viele.

Gestrichene a werden zu X, gestrichene b zu Y.

Zustand	liest	schreibt	Bew.	neu	Bedeutung
q_0	a	X	R	q_1	ein a streichen
q_0	Y	Y	R	q_3	kein a mehr da — Rest prüfen
q_0	□	□	N	q_a	leeres Wort, annehmen
q_1	a	a	R	q_1	nach rechts zum ersten b
q_1	Y	Y	R	q_1	über Gestrichenes hinweg
q_1	b	Y	L	q_2	zugehöriges b streichen
q_2	a	a	L	q_2	zurück nach links
q_2	Y	Y	L	q_2	zurück nach links
q_2	X	X	R	q_0	am letzten X — nächste Runde
q_3	Y	Y	R	q_3	nur noch Y erlaubt
q_3	□	□	N	q_a	sauber aufgegangen, annehmen

Beispiel: Ablauf für aabb

```
q0  [a] a  b  b        q1   X  X  Y [b]
q1   X [a] b  b        q2   X  X [Y] Y
q1   X  a [b] b        q2   X [X] Y  Y
q2   X [a] Y  b        q0   X  X [Y] Y
q2  [X] a  Y  b        q3   X  X  Y [Y]
q0   X [a] Y  b        q3   X  X  Y  Y [□]
q1   X  X [Y] b        qa   angenommen
```

Dreizehn Schritte. Zweimal ist die Maschine hin- und hergelaufen — einmal für jedes Paar.

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Turingmaschine**.

Probieren Sie im Bedienfeld auch `aab` und `abb` — die Maschine bleibt dann stehen, ohne q_a zu erreichen.

Stolperstein

Fehlt eine Regel, hält die Maschine einfach an, **ohne** anzunehmen. Bei `aab` etwa steht die Maschine irgendwann in q_1 vor einem $\square$ — dafür gibt es keine Zeile in der Tabelle. Das Wort ist damit abgelehnt. Ablehnen heißt also nicht „Nein sagen", sondern nur „nicht in q_a enden".

1.2.3 Die Church-Turing-These

Turing und Church hatten unabhängig voneinander zwei ganz verschiedene Begriffe von „berechenbar" entwickelt — Turing seine Maschine, Church den λ-Kalkül. Dann stellte sich heraus: Beide umfassen genau dieselben Probleme. Dasselbe gilt für jedes weitere Modell, das seither vorgeschlagen wurde, und für jede Programmiersprache, die je gebaut wurde.

Merke: Church-Turing-These

Alles, was überhaupt im intuitiven Sinn **berechenbar** ist, kann eine Turingmaschine berechnen.

Das ist **kein Satz, sondern eine These**, und sie lässt sich grundsätzlich nicht beweisen: Auf der einen Seite steht ein exakter Begriff, auf der anderen das Alltagswort „berechenbar". Zwischen beiden kann man keinen Beweis führen. Widerlegen ließe sie sich dagegen sofort — durch ein Verfahren, das offensichtlich rechnet und das keine Turingmaschine nachahmen kann. In neunzig Jahren hat niemand eines gefunden.

Merke

Die These hat eine unbequeme Folge, die diesen Kurs durchzieht: Zeigt man, dass eine Turingmaschine ein Problem nicht lösen kann, dann kann **kein Rechner** es lösen — kein heutiger, kein zukünftiger, egal wie schnell. Das ist die Grundlage von Abschnitt 4.4.

Arbeitsauftrag 1

1. Lassen Sie die Additionsmaschine auf `111+1` laufen und schreiben Sie das Protokoll wie im Beispiel auf.
2. Was tut die Additionsmaschine, wenn auf dem Band gar kein `+` steht? Begründen Sie mit der Tabelle.
3. Geben Sie eine Turingmaschine an, die ein Band aus Nullen und Einsen von links nach rechts durchläuft und dabei jede `0` durch eine `1` ersetzt und umgekehrt.
4. Lassen Sie die zweite Maschine auf `ab` und auf `abb` laufen. Bei welchem Schritt genau scheitert `abb`?
5. Warum braucht die zweite Maschine zwei verschiedene Streichzeichen `X` und `Y`? Was ginge schief, wenn beide `X` hießen?

Selbstcheck

Können Sie in einem Satz sagen, was eine Turingmaschine kann, was ein endlicher Automat nicht kann — und woran das liegt?

1.3 Was ein Algorithmus ist

Turings Maschine war der Versuch, den Begriff des Verfahrens so weit zu entkleiden, dass nur das Wesentliche übrigbleibt. Für die tägliche Arbeit braucht man eine handlichere Fassung. Nicht jede Folge von Anweisungen ist ein Algorithmus: Ein Kochrezept kommt der Sache nahe, ein Zuruf wie „mach das mal ordentlich" nicht.

Definition: Algorithmus

Eine Folge von Anweisungen heißt **Algorithmus**, wenn sie fünf Eigenschaften erfüllt:

Allgemeingültigkeit — sie löst eine ganze Klasse von Problemen, nicht einen Einzelfall.

Ausführbarkeit — jede Anweisung ist verständlich formuliert und tatsächlich ausführbar.

Eindeutigkeit — an jeder Stelle steht fest, was als Nächstes geschieht.

Endlichkeit — die Beschreibung selbst ist ein endlicher Text.

Terminiertheit — nach endlich vielen Schritten liegt ein Ergebnis vor.

Beispiel: Verletzte Eigenschaften

Anweisung	verletzt
„Addiere 3{,}4 und grün."	Ausführbarkeit
„Addiere drei oder vier."	Eindeutigkeit
„Buchstabiere *Ferien*, bis du ein *y* liest."	Terminiertheit
„Berechne 3 + 5 + 1{,}23."	Allgemeingültigkeit

Die fünfte Eigenschaft ist die heikelste. Ob ein gegebenes Programm für jede Eingabe terminiert, lässt sich im Allgemeinen **nicht** entscheiden — kein Programm der Welt kann das zuverlässig prüfen. Warum das so ist, klärt Abschnitt 4.4.

Formulieren Sie eine Anweisung, die die Eigenschaft Endlichkeit verletzt:

Gymnasium Ottweiler

1.4 Das EVA-Prinzip

Fast jedes Programm zerfällt in drei Teile:

E	Eingabe	Tastatur, Datei, Sensor, Netz
V	Verarbeitung	hier wird das Problem gelöst
A	Ausgabe	Bildschirm, Datei, Drucker, Netz

Der Teil, über den nachgedacht werden muss, ist **V**. Ein- und Ausgabe sind Handwerk; sie wechseln mit der Technik, die Verarbeitung bleibt. Deshalb halten wir sie in diesem Skript knapp und schreiben Programme meist als reine Verarbeitung — eine Funktion, die etwas hereinbekommt und etwas zurückgibt.

Merke

Trennen Sie Ein-/Ausgabe von der Verarbeitung. Ein Unterprogramm, das sortiert, soll sortieren — und nicht nebenbei etwas auf den Bildschirm schreiben. Sonst ist es nur an der einen Stelle brauchbar, an der es entstanden ist.

Gymnasium Ottweiler

1.5 Variablen und Datentypen

Eine **Variable** ist ein benannter Platz im Speicher. Der **Datentyp** legt fest, was dort stehen darf und was man damit rechnen kann.

Typ	Python	Object Pascal	Beispielwert
ganze Zahl	`int`	`Integer`	`42`
Kommazahl	`float`	`Real`	`3.14`
Zeichenkette	`str`	`String`	`'Hallo'`
einzelnes Zeichen	`str` (Länge 1)	`Char`	`'A'`
Wahrheitswert	`bool`	`Boolean`	`True` / `true`

Der auffälligste Unterschied: In Pascal **muss** jede Variable vor ihrer Verwendung in einem `var`-Block deklariert werden, in Python nicht.

Python

```python
name = "Ada"
alter = 36
groesse = 1.52
volljaehrig = alter >= 18
```

Object Pascal

```pascal
var
  name: String;
  alter: Integer;
  groesse: Real;
  volljaehrig: Boolean;
begin
  name := 'Ada';
  alter := 36;
  groesse := 1.52;
  volljaehrig := alter >= 18;
end;
```

Stolperstein

In Pascal ist `=` der **Vergleich** und `:=` die **Zuweisung**. In Python ist `=` die Zuweisung und `==` der Vergleich. Wer von einer Sprache in die andere wechselt, schreibt diesen Fehler mindestens einmal — rechnen Sie damit.

Pascal ist **streng typisiert**: `alter := 'Ada'` lehnt der Compiler ab, bevor das Programm überhaupt läuft. Python prüft erst zur Laufzeit. Beides hat seinen Preis — der Compiler nörgelt früh, Python lässt Sie weiterlaufen und stürzt später ab.

1.6 Verzweigungen

Beispiel: Würfelspiel

Der Rechner würfelt gegen den Menschen. Bei 1 bis 3 gewinnt der Mensch, bei 4 bis 6 der Rechner.

Python

```python
import random

def wuerfeln():
    augen = random.randint(1, 6)
    if augen >= 4:
        gewinner = "Rechner"
    else:
        gewinner = "Mensch"
    return augen, gewinner
```

Object Pascal

```
procedure Wuerfeln(var augen: Integer; var gewinner: String);
begin
  augen := Random(6) + 1;
  if augen >= 4 then
    gewinner := 'Rechner'
  else
    gewinner := 'Mensch';
end;
```

Stolperstein

Vor dem `else` steht in Pascal **kein** Semikolon. Das Semikolon trennt Anweisungen — und `if … then … else …` ist *eine* Anweisung.

1.6.1 Mehrfachauswahl

Hängt die Entscheidung an *einem* Wert mit vielen Fällen, wird die `if`-Kette unleserlich. Dafür gibt es die Mehrfachauswahl.

Object Pascal

```
case punkte of
  15, 14, 13: note := 'sehr gut';
  12, 11, 10: note := 'gut';
   9,  8,  7: note := 'befriedigend';
   6,  5,  4: note := 'ausreichend';
   3,  2,  1: note := 'mangelhaft';
           0: note := 'ungenügend';
else
  note := 'keine gültige Punktzahl';
end;
```

Der geprüfte Ausdruck heißt **Selektor** und muss in Pascal ganzzahlig, ein Zeichen oder ein Wahrheitswert sein. Bereiche lassen sich mit `10..12:` abkürzen.

Python kennt seit Version 3.10 `match`, in der Schule genügt meist ein Wörterbuch:

```
note = {15: "sehr gut", 14: "sehr gut", 13: "sehr gut",
        12: "gut", 11: "gut", 10: "gut"}.get(punkte, "…")
```

Arbeitsauftrag 2

6. Schreiben Sie ein Programm, das zu einer Oberstufennote in Punkten (0 bis 15) den Notentext ausgibt — in beiden Sprachen.
7. Schreiben Sie ein Programm, das bei einer Division durch Null eine Fehlermeldung ausgibt, statt abzustürzen.

Gymnasium Ottweiler

1.7 Schleifen

Definition: Schleife

Eine **Schleife** ist ein Programmbaustein, der eine Anweisungsfolge mehrfach ausführt. Wir unterscheiden drei Arten:

- **Zählschleife** (`for`) — die Zahl der Durchläufe steht vorher fest.
- **Kopfgesteuerte Schleife** (`while`/`while do`) — vor jedem Durchlauf wird geprüft; sie kann *keinmal* laufen.
- **Fußgesteuerte Schleife** (`repeat until`) — erst nach dem Durchlauf wird geprüft; sie läuft **mindestens einmal**.

Python kennt die fußgesteuerte Schleife nicht; man baut sie mit `while True:` und `break` nach.

Beispiel: Summe von a bis b

Python

```python
def summe(a, b):
    ergebnis = 0
    for zahl in range(a, b + 1):
        ergebnis = ergebnis + zahl
    return ergebnis
```

Object Pascal

```
function Summe(a, b: Integer): Integer;
var
  zahl, ergebnis: Integer;
begin
  ergebnis := 0;
  for zahl := a to b do
    ergebnis := ergebnis + zahl;
  Result := ergebnis;
end;
```

Für $a = 5$ und $b = 10$ ergibt sich $5 + 6 + 7 + 8 + 9 + 10 = 45$.

Stolperstein

Python gibt einen Wert mit `return` zurück — und beendet die Funktion damit sofort. Pascal hat kein `return`: Dort wird dem Ergebnis ein Wert **zugewiesen**, und die Funktion läuft weiter bis zum `end`. Dafür gibt es in Object Pascal die vordefinierte Variable `Result`:

```
  Result := ergebnis;
```

Klassisches Pascal erlaubt stattdessen auch die Zuweisung an den Funktionsnamen (`Summe := ergebnis;`). Beides ist in Delphi und Lazarus erlaubt; in diesem Skript steht durchgehend `Result`, weil es in allen Übersetzermodi funktioniert und beim Lesen klarer zeigt, dass hier ein Rückgabewert gesetzt und nicht die Funktion erneut aufgerufen wird. Soll die Funktion sofort verlassen werden, dient dazu `Exit`.

Stolperstein

In Python läuft `range(a, b)` **bis ausschließlich** `b`. In Pascal läuft `for zahl := a to b` **bis einschließlich** `b`. Daher das `+ 1` im Python-Beispiel. Das ist die häufigste Ursache für einen Durchlauf zu wenig.

1.7.1 Welche Schleife wann?

Situation	passende Schleife
„zehnmal" oder „für jedes Element"	`for`
„solange noch nicht fertig", evtl. nie	`while`
„bis der Benutzer `q` drückt" — mindestens einmal	`repeat until`

Die Gefahr liegt immer bei der Abbruchbedingung: Wird sie nie wahr, entsteht eine **Endlosschleife** und das Programm reagiert nicht mehr.

Arbeitsauftrag 3

8. Berechnen Sie die Fakultät $n! = 1 \cdot 2 \cdot \ldots \cdot n$ mit einer Schleife, in beiden Sprachen.
9. Ein Kapital k wird jährlich mit p Prozent verzinst. Geben Sie den Kontostand für jedes Jahr aus, bis er sich verdoppelt hat. Welche Schleifenart passt — und warum nicht die `for`-Schleife?
10. Schreiben Sie eine Schleife, die absichtlich nicht terminiert, und erklären Sie in einem Satz, welche Grundeigenschaft aus Abschnitt 1.3 sie verletzt.

Gymnasium Ottweiler

1.8 Unterprogramme

Ein Programm aus einem Stück ist ab etwa fünfzig Zeilen nicht mehr zu überblicken. Die Gegenmaßnahme ist alt und wirksam: Man zerlegt es in benannte Teile, die man einzeln verstehen, einzeln testen und mehrfach verwenden kann.

Definition: Unterprogramm

Ein **Unterprogramm** ist eine benannte, abgeschlossene Anweisungsfolge, die von anderer Stelle aus aufgerufen werden kann. Zwei Bauarten:

- Eine **Prozedur** führt etwas aus und gibt keinen Wert zurück. Sie ist ein **Auftrag**.
- Eine **Funktion** berechnet einen Wert und gibt ihn zurück. Sie ist eine **Anfrage**.

Diese Unterscheidung ist keine Formalie. Sie taucht in Kapitel 2 unverändert wieder auf — dort heißen die Unterprogramme *Methoden*, und ein Objekt bekommt entweder einen Auftrag („ändere deine Farbe") oder eine Anfrage („welche Farbe hast du?").

Merke

Auftrag oder Anfrage? Fragen Sie: Will ich etwas *wissen* oder etwas *bewirken*? Wissen → Funktion. Bewirken → Prozedur. Ein Unterprogramm, das beides tut, ist meist eines zu wenig.

In Python gibt es nur `def`; eine Funktion ohne `return` liefert `None` und ist damit faktisch eine Prozedur. Pascal unterscheidet die beiden im Schlüsselwort.

Anfrage (Funktion)

```python
def flaeche(breite, hoehe):
    return breite * hoehe
```

```pascal
function Flaeche(breite, hoehe: Real): Real;
begin
  Result := breite * hoehe;
end;
```

Auftrag (Prozedur)

```python
def ausgeben(text):
    print(">>> " + text)
```

```pascal
procedure Ausgeben(text: String);
begin
  WriteLn('>>> ', text);
end;
```

1.8.1 Parameter: Wert oder Referenz

Beim Aufruf werden **Argumente** an die **Parameter** übergeben. Entscheidend ist, *wie*.

Definition: Wert- und Referenzparameter

Bei einem **Wertparameter** erhält das Unterprogramm eine **Kopie**. Änderungen daran bleiben drinnen.

Bei einem **Referenzparameter** erhält es die **Variable selbst**. Änderungen wirken nach außen. In Pascal schreibt man dafür `var` vor den Parameter.

Beispiel: Tauschen — einmal falsch, einmal richtig

```
1  procedure TauschFalsch(a, b: Integer);      { Wertparameter }
2  var hilf: Integer;
3  begin
4    hilf := a;  a := b;  b := hilf;
5  end;
6
7  procedure TauschRichtig(var a, b: Integer); { Referenzparameter }
8  var hilf: Integer;
9  begin
10   hilf := a;  a := b;  b := hilf;
11 end;
```

Nach `TauschFalsch(x, y)` stehen `x` und `y` unverändert da: Getauscht wurden die Kopien. Nur ein einziges Wort unterscheidet die beiden Fassungen.

Python kennt kein `var`. Dort gilt: Zahlen, Zeichenketten und Wahrheitswerte verhalten sich wie Wertparameter, Listen und Objekte wie Referenzparameter.

```
1 def haenge_an(liste):         # liste ist eine Liste
2     liste.append(99)          # wirkt nach außen
3
4 def ersetze(liste):
5     liste = [99]              # wirkt NICHT nach außen
```

Stolperstein

Im zweiten Fall zeigt der Name `liste` danach auf eine neue Liste — die alte bleibt unberührt. Verändern Sie das übergebene Objekt (`append`, `sort`, Zuweisung an `liste[i]`), oder geben Sie ein Ergebnis zurück. Eine Zuweisung an den Parameternamen selbst erreicht den Aufrufer nie.

1.8.2 Lokal und global

Eine im Unterprogramm deklarierte Variable ist **lokal**: Sie entsteht beim Aufruf und verschwindet am Ende. Eine außerhalb deklarierte Variable ist **global** und überall sichtbar.

Globale Variablen sind bequem und darum gefährlich: Wenn jedes Unterprogramm an derselben Variablen schreiben darf, lässt sich ein Fehler nicht mehr eingrenzen. Jede Stelle im Programm kommt als Verursacher in Frage.

Merke

Ein Unterprogramm soll alles, was es braucht, über seine Parameter bekommen und alles, was es liefert, über seinen Rückgabewert abgeben. Dann — und nur dann — kann man es isoliert testen.

Diese Regel ist der Kern dessen, was Kapitel 2 das **Geheimnisprinzip** nennt.

Arbeitsauftrag 4

11. Schreiben Sie eine Funktion `ggt(a, b)` nach dem euklidischen Algorithmus — in beiden Sprachen.
12. Schreiben Sie eine Prozedur, die eine Zahl in Worten ausgibt (`3` → `drei`), für 0 bis 9. Begründen Sie, warum hier eine Prozedur und keine Funktion naheliegt — und wann das Gegenteil besser wäre.
13. Was gibt das folgende Programm aus? Begründen Sie, ohne es laufen zu lassen.

```python
def f(x, liste):
    x = x + 1
    liste.append(x)
    return x

a = 5
b = []
f(a, b)
print(a, b)
```

14. Überführen Sie `TauschRichtig` nach Python. Was müssen Sie ändern — und warum?

Selbstcheck

Können Sie ohne nachzusehen sagen, welche der drei Schleifenarten mindestens einmal läuft? Und können Sie an einem eigenen Beispiel erklären, warum ein Wertparameter beim Tauschen versagt?

Gymnasium Ottweiler