Andreas Hofmann

# Programmieren lernen

Grundlagen für Studium und Beruf – praxisnah und sprachunabhängig

mitp

Andreas Hofmann

# Programmieren lernen

## Grundlagen für Studium und Beruf – praxisnah und sprachunabhängig

mitp

**Bibliografische Information der Deutschen Nationalbibliothek**
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über https://portal.dnb.de/opac.htm abrufbar.

ISBN 978-3-7475-1019-3

1. Auflage 2025

www.mitp.de

E-Mail: mitp-verlag@lila-logistik.com
Telefon: +49 7953 / 7189 - 079
Telefax: +49 7953 / 7189 - 082



Lektorat: Janina Vervost
Sprachkorrektorat: Christine Hoffmeister
Covergestaltung: Christian Kalkert
Bildnachweis: © Denis Yevtekhov / stock.adobe.com
Satz: III-satz, Kiel, www.drei-satz.de

# Inhaltsverzeichnis

# Einleitung

Das Programmieren ist zwar noch eine relativ junge Disziplin – etwa im Vergleich zur Fotografie oder dem Maschinenbau –, allerdings ist die Informationstechnologie ein sehr schnelllebiges Gebiet. Das Programmieren von Computerprogrammen ist also von diesem Standpunkt aus gesehen schon recht alt. Verändert hat sich dennoch recht wenig. Das Ziel ist immer noch dasselbe, nämlich einer Maschine beizubringen, bestimmte Aufgaben zu erfüllen. Zwar haben sich die Methoden weiterentwickelt, die Grundbausteine eines Programms sind aber immer noch dieselben und auch der grundlegende technische Mechanismus (siehe Abschnitt 1.2 »Wie funktioniert das Programmieren von Computern?«) ist gleich geblieben.

Die Entwicklungen auf dem Gebiet der Software-Programmierung haben, neben neuen technischen Möglichkeiten der Maschinen, lediglich ein schnelleres und leichteres Arbeiten des Programmierers zur Folge. Selbst die künstliche Intelligenz (KI), die gerade in aller Munde ist, revolutioniert das Programmieren – zumindest vorerst – nicht. Künstliche Intelligenz ist, anders als der Name vermuten lässt, nämlich gar nicht so intelligent und auch nichts anderes als ein von Menschen erschaffenes Programm.

Wer heute Programmieren lernen will, muss also immer noch beinahe dieselben Grundlagen lernen wie jemand, der vor einigen Jahrzehnten programmieren gelernt hat.

## Das Ziel dieses Buchs

Dieses Buch richtet sich an Programmieranfänger und ist als Einstieg für jene gedacht, die von Grund auf Programmieren lernen wollen. Der Fokus liegt dabei auf der Vermittlung der grundlegenden Konzepte und Bausteine, die Sie als Programmierer oder Programmiererin beherrschen müssen, nicht auf einer bestimmten Programmiersprache.

Es dient aber auch als Orientierung für die weiteren Schritte und Themen auf ihrem Lernpfad und als Entscheidungshilfe für die Wahl einer oder mehrerer Programmiersprachen.

Sehen Sie dieses Buch als ersten Schritt auf dem Weg zum Programmierer. Es ist kein »Komplettwerk für Programmierer«, denn so etwas gibt es nicht. Das Programmieren von Software ist eine derart umfangreiche Disziplin, dass alle dafür nötigen Themen unmöglich in einem Buch abgedruckt werden können. Bücher können ohnehin nur gewisse Grundlagen vermitteln. Die Fähigkeit zu programmieren erwirbt man viel mehr durch die Praxis und die Erfahrung, die man sammelt, wenn man Programmcode schreibt. Dafür erhalten Sie mit diesem Buch eine solide Grundlage.

Das Buch beinhaltet keine fertigen Lösungen für bestimmte Problemstellungen und Aufgaben. Diese finden Sie in sogenannten *Cookbooks* (Kochbüchern) oder noch umfangreicher und aktueller im Internet.

Auch wenn dieses Buch keinerlei Vorwissen zum Thema »Programmierung« erfordert, so setzt es doch ein gewisses IT-Basiswissen voraus oder benötigt zumindest eine gewisse IT-Affinität und Lernbereitschaft, um sich das notwendige Wissen aus anderen Quellen anzueignen.

## Programmieren lernen

Wie lernt man den nun Programmieren? Wie bereits erwähnt, ist das Programmieren bzw. Schreiben von Computerprogrammen eine komplexe Disziplin, denn das Erlernen einer Programmiersprache macht nur einen geringen Teil aus. Wichtiger sind die grundlegenden Konzepte, die die Programmierung ausmacht. Möchte man Programmierer werden, ist es viel essenzieller, eine gewisse Denkweise zu erlernen. Einen guten Programmierer zeichnet insbesondere eine hohe Problemlösungskompetenz aus.

Natürlich stellt das Erlernen einer Programmiersprache eine nicht zu unterschätzende Hürde für jeden Anfänger dar, haben Sie diese aber erst einmal gemeistert, lassen sich weitere Sprachen mit erheblich weniger Aufwand erlernen. Man muss eine Sprache auch nicht bis ins kleinste technische Detail kennen oder jeden Befehl in- und auswendig beherrschen, um erfolgreich Computerprogramme zu schreiben.

Programmieren ist keine Inseldisziplin, es ist nur ein Werkzeug, um Probleme zu lösen. Dafür ist, je nach Art des Problems, aber noch weiteres Wissen und Können erforderlich. Sie können die bestausgestattete Werkstatt Ihr Eigen nennen, ohne das nötige Fachwissen werden Sie kein Auto reparieren können. Nehmen Sie sich also etwas Zeit und lernen Sie, wie Sie Problemstellungen mit dem Schreiben von Programmen lösen. Mit diesem Buch liefere ich Ihnen die Grundlagen und die notwendige Denkweise hierfür.

## Aufbau des Buchs

In diesem Buch finden Sie nicht nur Grundwissen zur Programmierung, sondern auch die nötigen Grundbausteine, um selbst ein Computerprogramm zu schreiben. Die entsprechenden Kapitel zielen auf ein allgemeines Verständnis ab, ohne an eine konkrete Programmiersprache gebunden zu sein. Dennoch soll die Praxis nicht zu kurz kommen. Im Praxisteil erfahren Sie, wie Sie das Gelernte in ein tatsächliches Stück Software verwandeln. Auch wenn ich hierfür auf eine konkrete Programmiersprache zurückgreifen muss, so können Sie die Techniken in jeder beliebigen anderen Programmiersprache ausprobieren.

- **Theorie (Kapitel 1 bis 4):** Sie erfahren zuerst, wie das Programmieren von Maschinen ganz grundsätzlich funktioniert. Anschließend lernen Sie die Basiselemente eines Computerprogramms kennen sowie weitere Methoden und Konzepte, die in der modernen Softwareentwicklung unerlässlich sind.
- **Praxis (Kapitel 5 bis 7):** Sie lernen anschließend, wie der Arbeitsprozess eines Programmierers aussieht, welche Werkzeuge und Abläufe notwendig sind und wie Sie sich das Programmieren einfacher gestalten können. Zum Abschluss zeige ich Ihnen, wie Sie das Gelernte in tatsächlichen Quellcode umsetzen.

Kapitel 2 »Daten abbilden und miteinander in Beziehung setzen«, Kapitel 3 »Den Programmfluss kontrollieren« und Kapitel 4 »Objektorientierte Programmierung« bauen aufeinander auf. Kapitel 6 »Programmieren in der Praxis« setzt das erworbene Wissen aus Kapitel 2 bis Kapitel 4 in die Praxis um, daher empfehle ich Ihnen, diese Kapitel in der vorgesehenen Reihenfolge zu lesen.

Viele Themen der Programmierung sind miteinander verknüpft und lassen sich nicht immer eindeutig in einer gewissen Reihenfolge beschreiben. Sie finden an den entsprechenden Stellen immer Querverweise auf die passenden Abschnitte dieses Buchs. Manchmal kann es auch sinnvoll sein, nach einem Kapitel nochmals zu vorangegangenen Abschnitten zurückzukehren und diese mit mehr Wissen erneut zu beleuchten.

Die Kapitel 1 »Einführung in die Welt der Computerprogramme«, Kapitel 5 »Werkzeuge für die Programmierung« sowie Kapitel 7 »Welche Programmiersprache ist die richtige für mich?« liefern Zusatzwissen, die zum Erlernen der Programmierung nicht direkt erforderlich, aber durchaus hilfreich und wissenswert sind. Sie können sie in beliebiger Reihenfolge lesen.

### Konventionen in diesem Buch

Beim Programmieren dreht sich alles um Programmcode. Dieser besteht aus speziellen Schlüsselwörtern und Befehlen, die im Text durch eine `eigene Schriftart` hervorgehoben sind.

```
Programmcode, Befehle und spezielle Ausdrücke können aber auch in einem
eigenen Absatz stehen.
```

Mehrere Befehle, die zusammengehören bzw. einen Codeabschnitt darstellen, sind ebenfalls in einem eigenen Absatz formatiert und mit Zeilennummern versehen (siehe Listing). Auch moderne Quellcode-Editoren verwenden nummerierte Zeilen, da das den Verweis auf einzelne Befehle erleichtert. Speziell in Fehlerfällen findet sich eine Zeilennummer schneller als ein bestimmter Befehl.

```
1  namespace HelloWorld
2  {
3      internal class Program
4      {
5          static void Main(string[] args)
6          {
7              Console.WriteLine("Hello, World!");
8          }
9      }
10 }
```

**Listing 1:** So wird zusammengehörender Quellcode im Buch formatiert.

Hin und wieder finden Sie im formatierten Quellcode ergänzende Kommentare, diese werden beginnend mit zwei Schrägstrichen (Slashes) gekennzeichnet. In etwa so: `//ich bin ein Kommentar`.

Kommentare helfen, bestimmte Stellen im Quellcode näher zu erläutern bzw. deren Auswirkung zu zeigen, ohne im Fließtext auf jedes Detail eingehen zu müssen.

Manchmal würden Codeabschnitte aus mehreren Zeilen Code bestehen, aber nicht immer sind alle davon relevant für ein Beispiel bzw. würden von den eigentlichen Befehlen ablenken. Nicht relevante Zeilen und Abschnitte werden in den Beispielen ausgelassen und durch `...` ersetzt.

Manchmal werden Beispiele laufend erweitert, auch hier ist es nicht sinnvoll, immer wieder den kompletten Code abzudrucken, also werden die Stellen aus vorherigen Listings zusammengekürzt und ebenfalls durch `...` ersetzt.

### Mehr Wissen

In diesen Boxen finden Sie Zusatzwissen, das zu einem gewissen Themenbereich gehört, aber für Anfänger noch zu speziell oder einfach weniger relevant ist. Die Themen in diesen Boxen werden von mir nur grob umrissen und können als Grundlage für eigene Recherchen dienen.

Auch Sachverhalte, die sich in den einzelnen Programmiersprachen stärker unterscheiden oder bei bestimmten Sprachen eine Ausnahme darstellen, finden Sie in diesen Boxen.

Kapitel 1

# Einführung in die Welt der Computerprogramme

Bevor ich Sie mit den Grundlagen und Konzepten der Programmierung vertraut mache, erhalten Sie in diesem Kapitel eine allgemeine Einführung zum Thema Programmieren. Dazu gehört ein kurzer Abriss aus der Geschichte, hier erfahren Sie, wie es zur Erfindung von Computerprogrammen kam und wie sich Programme und Programmiersprachen über die Jahrzehnte entwickelt haben. Danach erkläre ich Ihnen kurz, wie Programmiersprachen und Programmcode funktioniert und anschließend bekommen Sie eine Übersicht über die Einsatzgebiete von Computerprogrammen sowie über die gängigen Programmiersprachen.

## 1.1 Eine kleine Reise durch die Geschichte der Programmierung

Auch wenn wir heute auf Computern programmieren, überlegt man kurz, wird klar, dass ein Computer nur funktioniert, weil darauf ein Programm läuft, das das Arbeiten mit dem Computer erst möglich macht. Programmiersprachen, Programme und das Schreiben eben jener wurde also vor dem modernen Computer erfunden. Die Programmierung von Maschinen ist älter, als viele denken.

Vor den Computern gab es Rechenmaschinen und vor den Rechenmaschinen gab es nur mechanische Maschinen. Diese Maschinen waren in der Lage, bestimmte Aufgaben oder Arbeitsschritte automatisch zu erledigen. Die Funktion basiert auf deren Bauweise bzw. der mechanischen Konstruktion. Die Rede ist von industriellen Maschinen wie etwa Stanzen und Pressen, aber auch komplexeren Konstruktionen, wie etwa mechanischen Webstühlen. Aber egal wie komplex die Arbeitsschritte auch waren, die Ausführung erfolgte ohne jegliche Logik. Zwar konnten Werkzeuge und Aufsätze getauscht werden, dafür war aber ein menschliches Eingreifen erforderlich. Um andere Arbeitsschritte auszuführen, benötigte man eine andere mechanische Konstruktion. So konnten die ersten mechanischen Webstühle immer nur ein bestimmtes Muster weben.

Zu Beginn des 19. Jahrhunderts erfand der Franzose Joseph-Marie Jacquard den programmierbaren Webstuhl. Und auch wenn dieser noch keine Rechenmaschine war, folgte dieser dem Grundprinzip der elektronischen Datenverarbeitung: Ein-

gabe – Verarbeitung – Ausgabe (EVA). Über Lochstreifen konnte dem Webstuhl das zu webende Muster übermittelt werden.

Die erste mechanische Rechenmaschine wurde 1837 von Charles Babbage erfunden. Als erste mathematisch-logische Programmierung gilt die Vorschrift zur Berechnung von Bernoulli-Zahlen[1] von Ada Lovelace, die als erste Programmiererin der Geschichte gilt.

Zu Beginn des 20. Jahrhunderts wurden weitere Rechenmaschinen entworfen und dafür wiederum Vorschriften zum Lösen mathematischer Probleme entwickelt. Als Meilensteine gelten die vom deutschen Bauingenieur Konrad Zuse erfundenen und gebauten Rechenmaschinen. In den Vierzigerjahren des 20. Jahrhunderts griff der österreichisch-ungarische Mathematiker John von Neumann einige Ideen Zuses auf und beschrieb mit der Von-Neumann-Architektur ein Referenzmodell für Computer. Diese bildet die Grundlage für die meisten der heute bekannten Computer.

Nach dem Zweiten Weltkrieg entstanden, in den USA, die ersten höheren Programmiersprachen FORTRAN, Lisp und COBOL. Die an der Entwicklung von COBOL beteiligte Grace Hopper entwickelte auch den ersten Compiler (Abschnitt 1.2.1 »Kompilierte Programmiersprachen«). In den Sechzigern und Siebzigern des vergangenen Jahrhunderts wurde eine Vielzahl weiterer Programmiersprachen entwickelt. Grundlage war der technische Fortschritt in der Entwicklung von Computer-Hardware. Viele dieser Sprachen sind längst vergessen, andere werden noch heute verwendet oder entwickelten sich zu einigen der aktuellen Sprachen weiter. Nennenswerteste Vertreter sind die Sprachen BASIC, welche durch die ersten leistbaren Heimcomputer der Siebziger populär wurde, sowie C, welche 1972 für das neue Betriebssystem Unix entwickelt wurde.

Ende der Siebzigerjahre stellte das US-Verteidigungsministerium erschrocken fest, dass über 450 teils nicht standardisierte Programmiersprachen in deren Projekten genutzt wurden. Der Versuch, eine Sprache zu finden, die alle Anforderungen des US-Militärs erfüllte, scheiterte. Das Militär entwickelte daraufhin eine eigene Sprache, Ada, benannt nach Ada Lovelace.

1983 stellte Bjarne Stroustrup C++ vor, eine Erweiterung von C, die jene neuen Konzepte enthielt, die seit der Erfindung von C Einzug in die Computerprogrammierung gehalten hatten. Das Wichtigste hierbei ist die objektorientierte Programmierung, bei der die Architektur eines Programms sich an jenen Objekten der wirklichen Welt anlehnt, die die Aufgabenstellung des Programms betreffen (siehe Kapitel 4 »Objektorientierte Programmierung«).

1 Als Bernoulli-Zahlen wird eine Reihe von rationalen Zahlen bezeichnet, die in verschiedenen mathematischen Bereichen Anwendung finden.

Die schnelle Ausbreitung des Internets stellte eine ganze eigene Herausforderung dar, denn plötzlich mussten Inhalte auf eine Vielzahl verschiedener Endgeräte angepasst werden. Der Erfolg des Internets begründet sich unter anderem auf HTML – keine Programmiersprache, sondern eine Auszeichnungssprache –, welche die Gestaltung von Webinhalten übernahm, und PHP, der Programmiersprache für Serverlogik und dynamische/interaktive Inhalte.

1995 folgte die plattformunabhängige, objektorientierte Programmiersprache Java und 2001 die von Microsoft beauftragte Sprache C#.

Die nachfolgenden Entwicklungen brachten Sprachen hervor, die entweder Schwachstellen in den bereits verbreiteten Sprachen beheben sollten oder die für spezielle Einsatzgebiete gedacht waren (mehr dazu siehe Kapitel 7 »Welche Programmiersprache ist die richtige für mich?«).

## 1.2 Wie funktioniert das Programmieren von Computern?

Was ist Programmieren eigentlich? In der deutschen Sprache spricht man vom Programmieren, wenn man sein Haushaltsgerät oder ein anderes elektronisches Gerät konfiguriert. Wer alt genug ist, hat vielleicht schon mal einen Videorekorder »programmiert«. Unter Software- oder Computerprogrammierung versteht man aber das Schreiben von Befehlsabläufen, die einen Computer anweisen, bestimmte Aufgaben durchzuführen.

Wichtig zu erwähnen ist, dass der Computer bzw. genauer gesagt dessen Prozessor die Programmiersprache, in der die Befehlsabläufe geschrieben sind, gar nicht versteht. Sie dient dazu, die Befehlsabläufe, sogenannte Algorithmen, dem menschlichen Verständnis möglichst nahekommend zu formulieren. Die Befehle, die mit einer bestimmten Programmiersprache geschrieben werden, müssen erst in einen Maschinencode übersetzt werden, um vom Computer ausgeführt werden zu können.

**Hinweis**

Die folgende Erklärung zur Programmierung von Computern, die Sie hier finden, soll ein grundlegendes Verständnis für die Thematik schaffen. Der tatsächliche Vorgang ist sehr technisch und komplex und eine Erläuterung aller Details passt nicht in den Rahmen dieses Buchs. Sollte Sie das Thema interessieren, finden Sie dazu online umfangreiche Erläuterungen. Auch in der Fachliteratur gibt es ein umfangreiches Angebot an Werken zu diesem Thema, etwa Code von Charles Petzold, ebenfalls erhältlich beim mitp-Verlag.

### 1.2.1 Kompilierte Programmiersprachen

Diese Art von Programmiersprachen verwenden einen sogenannten *Compiler* (deutsch: Übersetzer). Dabei handelt es sich um ein Computerprogramm, das den Quellcode, der in einer bestimmten Programmiersprache geschrieben wurde, in vom Prozessor verständlichen Maschinencode übersetzt (kompiliert).

Der Maschinencode enthält Befehle, die vom jeweiligen Prozessor, der den Code ausführt, verstanden werden. Unterschiedliche Hardware kann dabei unterschiedliche Befehlssätze aufweisen. Beim Kompilieren wird der Code auch optimiert, das heißt, er wird gekürzt, anders angeordnet und durchläuft weitere Änderungen, die das Programm schneller machen und weniger Speicher verbrauchen lassen.

```
1  int main() {
2      int a = 4;
3      int b = 1;
4      int c = a + b;
5      return c;
6  }
```

**Listing 1.1:** Simpler Programmcode, der in C, Java, C# und anderen Sprachen geschrieben worden sein könnte

```
1   55
2   48 89 E5
3   C7 45 FC 04
4   C7 45 F8 01
5   8B 45 F8
6   8B 55 FC
7   01 D0
8   89 45 F4
9   8B 45 F4
10  5D
11  C3
```

**Listing 1.2:** So könnte der Maschinencode für das Programm aus Listing 1.1 aussehen.

Ein simples Programm wie das aus Listing 1.1 ist selbst für Menschen ohne Programmiererfahrung halbwegs verständlich, die Variablen a und b werden addiert und das Ergebnis in die Variable c gespeichert, welche an die aufrufende Stelle zurückgegeben wird. Maschinencode (Listing 1.2) hingegen ist für einen Menschen nicht mehr lesbar. Natürlich wäre es möglich, Maschinencode zu lernen, das Unterfangen wäre aber eher mühsam. Dazu kommt noch die Tatsache, dass der Maschinencode je nach Plattform ganz unterschiedlich ausfallen kann.

Sie sehen also, es handelt sich um ein eher sinnloses Unterfangen. Dank des Compilers können Programmierer den Code mithilfe von Programmiersprachen in einem relativ einfach lesbaren Format schreiben und diesen dann für verschiedene Plattformen übersetzen lassen, ohne den Code für jede Plattform einzeln anpassen zu müssen. Bekannteste Vertreter kompilierter Programmiersprachen sind C bzw. C++.

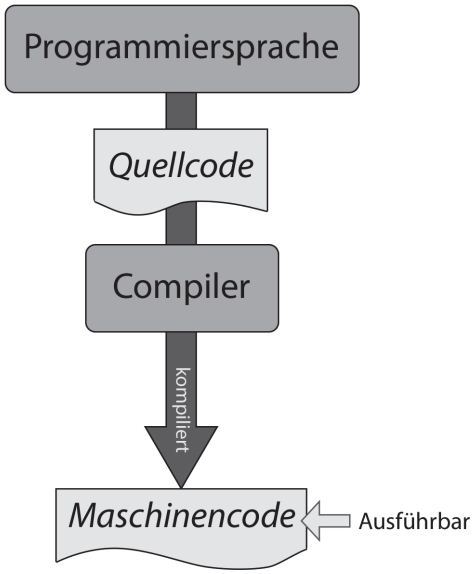


**Abb. 1.1:** Schematischer Ablauf bei kompilierten Programmiersprachen

Bis zur Entwicklung des ersten Compilers durch Grace Hopper Mitte der Fünfzigerjahre war das direkte Schreiben von Maschinencode allerdings die einzige Möglichkeit, Programme zu entwickeln. Damals gab es aber auch weniger verschieden Plattformen und Funktionalitäten der damaligen Computer – und somit war die Anzahl an Befehlen geringer als heute.

### 1.2.2 Interpretierte Programmiersprachen

Technisch etwas anders sieht es bei den sogenannten *interpretierten Sprachen* aus. Diese werden nicht direkt in Maschinencode übersetzt, ehe das Programm ausgeführt werden kann. Der Programmcode bleibt grundsätzlich, wie er ist. Erst beim Ausführen des Programms wird er an eine spezielle Software – den sogenannten *Interpreter* – übergeben. Dieser interpretiert den geschriebenen Code in Echtzeit und kennt alle Anweisungen der Programmiersprache, die er dann als Maschinencode ausführt. Bekannte Vertreter von interpretierten Sprachen sind Python und JavaScript.

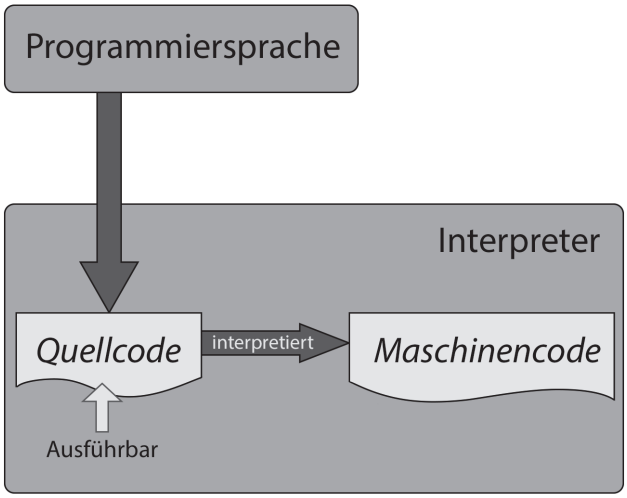


**Abb. 1.2:** Schematischer Ablauf bei interpretierten Programmiersprachen

### 1.2.3 Kompilieren vs. Interpretieren

Der Unterschied zwischen den beiden Vorgängen kann etwas verwirrend sein. Am besten hilft hier ein analoges Beispiel. Denken Sie an eine Gebrauchsanleitung oder eine Aufbauanleitung und nehmen Sie an, diese ist in einer Sprache geschrieben, die Sie nicht verstehen. Sie haben zwei Möglichkeiten, die Anweisungen der Anleitung umzusetzen:

- Option A: Es gibt bereits eine Übersetzung der Anleitung. Diese wurde im Vorfeld von einer Person (Compiler) angefertigt, die sowohl die Sprache der Originalanleitung als auch Ihre Sprache beherrscht.
- Option B: Sie kennen jemanden (Interpreter), der sowohl Ihre Sprache spricht als auch die Sprache der Anleitung versteht, und bitten denjenigen, die Anleitung direkt für Sie zu übersetzen.

Vereinfacht ausgedrückt, machen beide Methoden dieselbe Arbeit, aber zu einem anderen Zeitpunkt (vor dem Ausführen des Programms vs. während dem Ausführen) und an einem anderen Ort (Rechner des Entwicklers vs. Rechner des Anwenders).

Natürlich haben beide Methoden gewisse Vor- und Nachteile. Diese sind für Anfänger eventuell noch schwieriger zu verstehen als die unterschiedlichen Funktionsweisen der beiden Methoden. Aber gerade als Anfänger sollten Sie sich auch nicht allzu viele Gedanken darüber machen. Ihr Fokus sollte auf dem Lernen der Grundlagen liegen, dem Üben und dem Sammeln von Erfahrung durch das Schreiben von Code.

Dennoch möchte ich Ihnen im Folgenden eine vereinfachte Übersicht über Vor- und Nachteile der beiden Methoden bieten.

An dieser Stelle sollte noch erwähnt werden, dass, auch wenn von interpretierten bzw. kompilierten Programmiersprachen die Rede ist, diese Übersetzungsmethoden keine Eigenschaft der Sprache selbst sind, sondern davon abhängen, wie die Sprache implementiert, also technisch umgesetzt wird. Es gibt Programmiersprachen, die sowohl kompiliert als auch interpretiert werden können. Python ist hier ein gutes Beispiel. Primär als Skriptsprache eingesetzt, wird Python interpretiert. Durch die wachsende Beliebtheit und die breit gefächerten Einsatzmöglichkeiten gibt es für diverse Plattformen auch Python-Compiler.

### Just-in-time-Compiler

Für immer mehr Sprachen stehen Just-in-time-Compiler zur Verfügung. Diese speziellen Compiler übersetzten Code bzw. die Änderungen am Code in Echtzeit. Sie vereinen den flexiblen Entwicklungszyklus von interpretierten Sprachen mit der schnellen Ausführung von kompilierten Sprachen.

Ausgeliefert wird immer noch kompilierter Code. Durch die Echtzeitübersetzung muss der Code aber nicht nach jeder Änderung neu kompiliert werden, um ihn zu testen. JIT-Compiler gehen so weit, dass das Programm auch nicht mehr beendet werden muss, um Änderungen am Code vorzunehmen. Das ist besonders bei sehr komplexen Programmen hilfreich, wo es mitunter einige Anwendungsschritte und Zeit benötigt, um die getätigte Änderung sichtbar zu machen. Auch bei Fehlern, die nur schwierig zu reproduzieren sind, helfen JIT-Compiler enorm.

## Vorteile kompilierter Programmiersprachen

Kompilierte Programme sind in der Regel schneller, da das Übersetzen in Maschinencode schon vor dem Ausführen passiert. Der Anwender erhält das bereits übersetzte Programm und benötigt dafür auch keine zusätzlichen Ressourcen.

## Nachteile kompilierter Programmiersprachen

Das Kompilieren erfordert einen zusätzlichen Zeit- und Ressourcenaufwand zwischen dem Schreiben und Ausführen des Programms. Dieser Aufwand kann bei großen Programmen stark anwachsen und verlangsamt den Entwicklungszyklus. Code muss immer wieder getestet werden, aber bevor das geschehen kann, muss der Code kompiliert werden. Dasselbe gilt für die Fehlersuche. Um den Code zu verändern, muss das Programm gestoppt und nach der Änderung wieder kompiliert werden.

Bei großen Softwareprojekten arbeiten mehrere Entwickler am Code. Dieser wird dann nur noch zu Testzwecken am jeweiligen Rechner des Entwicklers kompiliert. Das Kompilieren des finalen Programms, ehe dieses produktiv genutzt werden kann, erfolgt auf eigenen Servern. Je mehr Code kompiliert werden muss, umso mehr Ressourcen muss für diese Server bereitgestellt werden.

Ein weiterer Nachteil ist, dass ein Compiler immer plattformabhängig ist. Soll ein Programm auf verschiedenen Plattformen laufen, muss der Code für jede Plattform separat kompiliert werden.

### Vorteile interpretierter Programmiersprachen

Interpretierte Programmiersprachen sind in der Regel etwas flexibler. Da der Interpreter das Übersetzen auf der Zielplattform übernimmt, ist der Code selbst immer plattformunabhängig und kann von jeder Plattform genutzt werden. Interpreter erlauben bzw. vereinfachen gewisse Programmierparadigmen und -funktionen (z.B.: Reflection und dynamic typing), deren Erklärung aber über den Rahmen dieses Buchs hinausgeht.

### Nachteile interpretierter Programmiersprachen

Primärer Nachteil ist die langsamere Ausführungszeit von Programmen, die erst interpretiert werden müssen. Außerdem muss der Anwender einen Interpreter für die verwendete Sprache installiert haben.

Ein weiterer Nachteil ist, dass gewisse Fehler, die Kompilierfehler (compiler error), wie der Name schon verrät, nicht vorab gefunden werden. Normalerweise werden sie zum Zeitpunkt des Kompilierens erkannt und führen zu einem Abbruch des Kompilierens. Der Interpreter aber arbeitet eine Codezeile nach der anderen ab und vergisst, was in den Zeilen zuvor passiert ist. Fehler treten dann zum Zeitpunkt des Ausführens auf und lassen das Programm abstürzen oder führen zu ungewolltem Verhalten bzw. falschen Ergebnissen. Wird Code nicht gewissenhaft getestet, bevor er ausgeliefert wird, fallen Fehler erst beim Anwender auf und das Programm ist nicht ausführbar.

Moderne Programmierumgebungen (IDEs, siehe Abschnitt 5.2 »Die Entwicklungsumgebung«) bieten diverse Funktionen, um das Schreiben von Code zu erleichtern. Diese können für interpretierte Sprachen eingeschränkt oder gar nicht zur Verfügung stehen.

## 1.2.4 Bytecode und Laufzeitumgebungen

Da Computerprogramme lange nicht mehr nur mathematische Probleme lösen, sondern komplexe Aufgaben übernehmen sollen, wie Dateien zu erstellen und zu manipulieren, oder über das Netzwerk zu kommunizieren, reicht ein Compiler heute nicht mehr aus. Als Programmierer wollen Sie aber den Maschinencode für

diese Aufgaben (meist) nicht selbst schreiben. Höhere Programmiersprachen bieten eine Vielzahl und Funktionen und Schnittstellen zu allen möglichen Hardwarekomponenten eines Computers, wie etwa grafische Ausgabe, Tonwiedergabe, Arbeits- und Festplattenspeicher oder eben Netzwerkschnittstellen. Damit Ihr Programm diese Funktionen nutzen kann, muss der Compiler entweder all diese Funktionen zusammen mit Ihrem Quellcode in das Programm integrieren oder der erzeuge Code wird in einer sogenannten *Laufzeitumgebung* (Runtime Environment) ausgeführt.

Hierbei wird der Quellcode nicht direkt in Maschinencode kompiliert, sondern in sogenannten *Bytecode*. Der Bytecode liegt in binärer Form (Folgen von Nullen und Einsen) vor, er ist also vom Menschen nicht mehr lesbar. Er ist immer noch plattformunabhängig, hat aber bereits einige Optimierungen durchlaufen. Der Bytecode wird von der Laufzeitumgebung entweder zu Maschinencode kompiliert oder direkt interpretiert.

Je nach Betriebssystem sind Laufzeitumgebungen für einige Sprachen bereits integriert. Andere Laufzeitumgebungen müssen Sie selbst nachinstallieren. Das trifft im Übrigen auch dann zu, wenn Sie ein fertiges Programm einfach nur ausführen wollen und nicht selbst programmieren. Die bekanntesten Laufzeitumgebungen sind das Java Runtime Environment (JRE) oder das .Net-Framework für C# und weitere verwandte Sprachen.

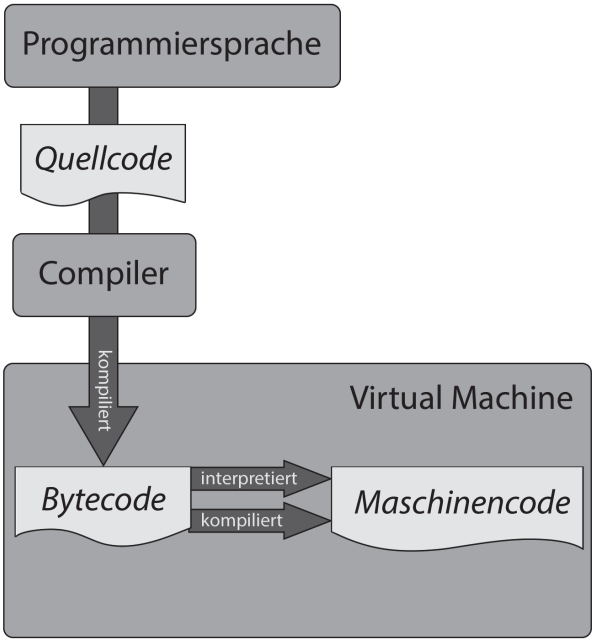


**Abb. 1.3:** Schematischer Ablauf bei Programmiersprachen mit Laufzeitumgebung

Kapitel 2

# Daten abbilden und miteinander in Beziehung setzen

In Kapitel 1 habe ich erläutert, wie die Programmierung entstanden ist und wie sie sich entwickelt hat. Ich habe auch in groben Zügen beschrieben, wie die Programmierung unter Verwendung von Programmiersprachen funktioniert.

In diesem Kapitel geht es darum, welche Elemente und Grundkonzepte in der Programmierung verwendet werden – und das möglichst unabhängig von einer bestimmten Programmiersprache. So können Sie sich wichtige Grundkenntnisse der Programmierung aneignen, ohne sich bereits auf eine Programmiersprache festzulegen.

## 2.1 Mit einer Maschine kommunizieren

Natürliche Sprachen bestehen aus einem Alphabet, also einer endlichen Sammlung aus Zeichen, die dazu genutzt werden, um Wörter zu bilden. Wörter haben eine gewisse Bedeutung und diese Bedeutung wird vom Menschen festgelegt. Durch Anwendung einer Grammatik können Wörter verwendet werden, um zu kommunizieren. Menschen, die dasselbe Alphabet, dieselbe Wortsammlung und dieselbe Grammatik verwenden, also dieselbe Sprache sprechen, verstehen diese Form der Kommunikation.

Mit Programmiersprachen verhält es sich ähnlich. Das Programmieren einer Maschine kann als Form der Kommunikation angesehen werden, wenn auch viel eingeschränkter. Durch das Programmieren weist man eine Maschine an, sich auf gewisse Weise zu verhalten. Eine Hochsprache (wie etwa C++ oder Java) ist dazu aber nicht zwingend erforderlich, denn die Anweisungen an die Maschine können auch direkt in Maschinensprache erfolgen. Hochsprachen dienen dazu, das Verfassen von Anweisungen zu vereinfachen und in einer für den Menschen leichter verständlichen Form zu ermöglichen.

Die Anweisungen, die eine Maschine versteht, sind natürlich viel limitierter als die Wörter, die ein Mensch versteht. Ein Mensch interpretiert Gesprochenes und kann in der Kommunikation auch Informationen erfassen, die nicht gesagt oder geschrieben wurden. Sprachliche Kommunikation kann daher auch trotz gemeinsamer Sprache unterschiedlich interpretiert werden. Bei Maschinen ist das

anders, Maschinen interpretieren die Befehle, die Sie beherrschen, immer gleich. Die Formulierung der Anweisungen muss daher sehr exakt erfolgen. Die Regeln dazu sind in der jeweiligen Syntax, quasi der Grammatik einer Programmiersprache, festgelegt. In erster Linie bestimmt die Syntax, welche Zeichen im Alphabet vorhanden sind und welche Wörter und Befehle in der Programmiersprache daraus gebildet werden können. Diese Wörter nennt man *Schlüsselwörter*. Jede Programmiersprache hat eine bestimmte Anzahl an Schlüsselwörtern. Diese haben eine bestimmte Bedeutung und dürfen nur in bestimmten Kontexten verwendet werden. Die Syntax legt auch fest, in welcher Kombination die Schlüsselwörter eine gültige Anweisung ergeben (ähnlich der Grammatik natürlicher Sprachen).

### Mehr Wissen – Syntax vs. Semantik

Nachdem die Syntax formal richtige Anweisungen garantiert, die von der Maschine interpretiert werden kann, sorgt die Semantik dafür, dass die Anweisungen auch Sinn ergeben. Ähnlich wie ein grammatisch richtiger Satz absolut keinen Sinn ergeben kann (z.B. Peter ist ein verheirateter Junggeselle.), kann das auch mit syntaktisch richtigen Anweisungen der Fall sein. Hochsprachen wie Java oder C# haben allerdings eine derart strikte Syntax, dass kaum semantische Fehler passieren können. Semantikfehler werden vom Compiler üblicherweise auch als Syntaxfehler ausgewiesen.

Programmiersprachen haben aber auch noch andere Gemeinsamkeiten mit natürlichen Sprachen. So wie die Sprachen, die wir Menschen verwenden, um untereinander zu kommunizieren, können Programmiersprachen in einem Sprachbaum dargestellt werden. Wie natürliche Sprachen entwickeln sich Programmiersprachen weiter und neue Sprachen entstehen auf Basis bestehender Sprachen. So wie menschliche Sprachen lokale Ausprägungen haben, können Programmiersprachen verschiedene sogenannte *Implementierungen* haben.

Diese Gemeinsamkeiten helfen Ihnen auch beim Erlernen von Programmiersprachen. Die erste Programmiersprache zu erlernen, mag noch schwerer fallen. Lernen Sie aber eine weitere Sprache, wird Ihnen das schon leichter von der Hand gehen. Je ähnlicher sich die Sprachen sind, umso schneller erlernt man sie.

Vergessen Sie aber nicht, dass es beim Programmieren nicht ausschließlich darum geht, eine Programmiersprache syntaktisch zu beherrschen. Viel wichtiger sind die grundlegenden Konzepte der Programmierung, die Sie hier in diesem Buch finden.

## 2.2 Informationen speichern

Es wird Ihnen kaum gelingen, ein brauchbares Programm zu schreiben, ohne Informationen zwischenzuspeichern. Nur so können Sie mit einem Programm mehr erreichen, als nur einfache Rechnungen zu lösen.

### 2.2.1 Variablen

Jede Programmiersprache stellt Ihnen dafür sogenannte *Variablen* zur Verfügung. Variablen kennen Sie vielleicht noch aus der Schule, genauer gesagt aus dem Mathematikunterricht. Dort stehen sie für eine bestimmte Zahl in einer Gleichung (z.B. x = 2 + y). Variablen in der Programmierung sind aber etwas anderes.

Variablen helfen Ihnen, Informationen zu speichern und wieder abzurufen. Das können Zahlen, Zeichen (Zahlen, Buchstaben, Sonderzeichen) oder Zeichenketten (alle möglichen Kombinationen von Zeichen, Wörter, Sätze oder ganze Texte), aber auch viele andere Inhalte (Daten, die nicht im Klartext vorliegen und daher nicht vom Menschen direkt gelesen werden können) sein. Genau genommen lässt sich alles, was sich digital repräsentieren lässt, auch speichern.

Denken Sie an Briefkästen oder Postfächer. Diese besitzen eine eindeutige Adresse und werden mit einem Namen verknüpft. Diese Kombination bleibt gültig, solange der Träger des Namens auch Besitzer des Postfachs ist. In den Postfächern können Briefe, Wurfzettel und Pakete hinterlegt werden. Briefe können Texte, Dokumente oder andere Kleinigkeiten enthalten. Inhalt von Paketen kann alles Mögliche sein.

Stellen Sie sich jetzt ein gigantisches Regal mit vielen Einlagefächern vor, alle gleich groß und jedes Einlagefach hat eine eindeutige Adresse. Möchten Sie Information bzw. Daten abspeichern, dann werden diese in einem oder auch mehreren Einlagefächern abgelegt. Für Sie als menschlicher Programmierer wäre es aber sehr mühsam, müssten Sie sich merken, wie diese Adressen lauten. Diese bestehen nämlich aus einer alphanumerischen Zeichenkette (siehe Listing 2.1). Selbst wenn Sie sich diese Adressen notieren und mit Hinweisen versehen, würden Sie diese sehr bald durcheinanderbringen, zumal Einlagefächer auch geleert und anderwärtig verwendet werden. So ein Programm zu schreiben, macht auch wenig Spaß. Wenn Sie sich an die kleine Geschichtsstunde aus Kapitel 1 erinnern, war das in der grauen Vorzeit der Programmiergeschichte aber durchaus der Fall. Glücklicherweise müssen Sie sich damit nicht mehr abmühen.

```
0x7ffe5367e044
```

**Listing 2.1:** Beispiel einer Speicheradresse

Damit Sie sich nicht merken müssen, wo Ihre Daten abgelegt wurden und wie viele Einlagefächer diese belegen, können Sie einfach einen Namen verwenden, hinter dem sich alles Weitere verbirgt. Anhand des Namens können Sie die gespeicherte Information abrufen oder auch manipulieren. Sie haben es bestimmt schon erraten, diese Namen sind die Variablen, um die sich dieser Abschnitt dreht.

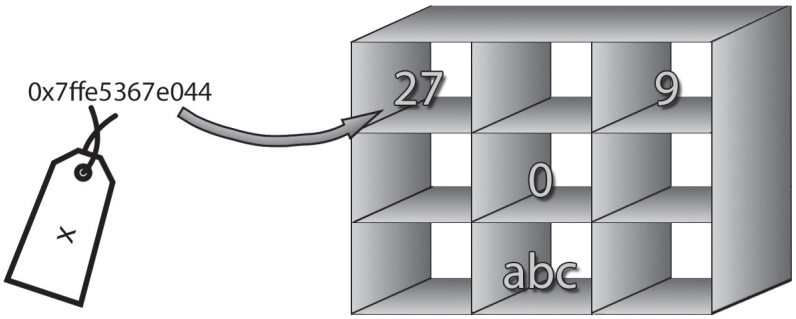


**Abb. 2.1:** Die Variable mit Namen »x« enthält den Wert 27.

Die Variable dürfen Sie dabei weitestgehend frei benennen. In der Welt der Computerprogrammierung hat sich die Konvention verbreitet, Variablennamen mit Kleinbuchstaben zu beginnen (z.B. `name`). Handelt es sich um ein zusammengesetztes Wort, wird jedes weitere Hauptwort mit einem Großbuchstaben begonnen (z.B. `geburtsJahr`). Diese Schreibweise wird auch Camelcase genannt.

Konventionen dienen dazu, eine einheitliche Darstellung des Codes zu erreichen, das erleichtert das Lesen und hilft Ihnen, sich auch in fremden Code schneller zurechtzufinden. Mehr zu den sogenannten *Coding Conventions* erfahren Sie in Abschnitt 5.3.2 »Code-Style-Guides und Coding-Standards«.

Schlüsselwörter einer Programmiersprache können nicht als Variablenname verwendet werden. Auch Leerzeichen sind nicht erlaubt. Zahlen können Sie innerhalb der Bezeichnung gerne verwenden, auch Unterstriche sind erlaubt, ansonsten sollten Sie von Sonderzeichen absehen, da diese üblicherweise nicht erlaubt sind.

**Tipp**

Umlaute können je nach Entwicklungsumgebung akzeptiert werden, Sie sollten aber von deren Verwendung absehen. Ändert sich die Entwicklungsumgebung oder wollen Sie den Code mit jemandem teilen oder auf einem anderen Gerät verwenden, kann es durchaus sein, dass Umlaute nicht akzeptiert werden.

Da auch in der Welt der Programmierung Englisch die Standardsprache ist, gewöhnen Sie sich doch gleich an, englische Bezeichnungen zu vergeben.

### Mehr Wissen – Speichermanagement

Das Regal mit den Einlagefächern in meiner kleinen Analogie repräsentiert den Arbeitsspeicher, denn dort spielen sich alle relevanten Aktionen Ihres Programms ab. Damit Sie sich nicht abmühen müssen, freie Einlagefächer zu finden und die richte Anzahl an Fächern zu reservieren und diese auch wieder zu leeren, wenn sie nicht mehr benötigt werden, bieten die meisten Hochsprachen ein System zur Speicherverwaltung. Dieses ist Teil der Laufzeitumgebung, arbeitet im Hintergrund und kümmert sich um alle Speicherbelange Ihres Programms.

Anders sieht die Sache aber beispielsweise bei der Programmiersprache C/C++ aus. Dort sind Sie selbst dafür verantwortlich, genügend Einlagefächer zu reservieren und diese vor allem wieder zu leeren, wenn diese nicht mehr benötigt werden. Vergessen Sie Letzteres, kann Ihnen bzw. Ihrem Programm sehr schnell der Speicher ausgehen. Dazu kommt noch eine der Besonderheiten von C++, die Zeiger, das sind Referenzen auf den Speicher. Mit den Details zu Zeigern werde ich Sie in diesem Buch nicht belästigen, Sie müssen nur wissen, dass Sie die Speicheradresse, auf die ein Zeiger zeigt, jederzeit ändern können. Bei Variablen geht das nicht. Solange eine Variable gültig ist, verweist sie auf eine gewisse Anzahl an Einlagefächern, sie können lediglich deren Inhalt in gewissem Rahmen ändern.

Zeiger aber können Sie problemlos auf ein anderes Einlagefach zeigen lassen. Sind Sie unachtsam, enthält das Fach nicht den Beginn der gewünschten Daten oder es enthält keine gültigen Daten, so wie Sie sie erwarten. Ihr Programm erzeugt dann eine Fehlermeldung oder stürzt ab. Es geht aber noch schlimmer, so könnten zwar gültige Daten enthalten sein (Sie erwarten eine Ganzzahl und erhalten auch eine), allerdings sind es die falschen Daten und Ihr Programm arbeitet mit diesen falschen Daten weiter und kommt auch zu einem Ergebnis, aber eben nicht dem richtigen. Oder Sie überschreiben Daten, die Sie an anderer Stelle benötigen. Sie sehen also, C++ ist nicht die beste Wahl für eine Einsteigersprache.

## 2.2.2 Datentypen

Im vorherigen Abschnitt habe ich im Zuge der Einlagefächer-Analogie zwei Dinge erwähnt. Erstens: Es können verschiedene Elemente abgespeichert werden: Zahlen, Zeichen, Zeichenketten und binäre Daten. Zweitens: Unterschiedliche Daten benötigen unterschiedlich viele Einlagefächer. Wenn Sie den »Mehr Wissen«-Kasten gelesen haben, wissen Sie auch schon, dass Sie sich bei den meisten Programmiersprachen nicht darum kümmern müssen, wie viele Einlagefächer – also Speicher – Sie benötigen. Daraus ergeben sich zwei Fragen: Woher weiß das Pro-