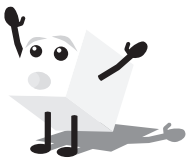
Inhaltsverzeichnis

3 Einführung in die objektorientierte Programmierung mit Java



3.1 Was lernen Sie in diesem Kapitel?

Java ist eine streng objektorientierte Programmiersprache und ohne Verständnis der grundlegenden Idee objektorientierter Programmierung (OOP) kommen Sie in Java nicht zurecht. Dieses Kapitel führt Sie in die fundamentalen theoretischen Konzepte, Ideen und Methodiken der objektorientierten Denkweise und vor allen Dingen deren konkrete Umsetzung in Java ein. Wobei in jedem Buch zu Java das Dilemma besteht, dass ein Programmierer für konkrete Praxis ebenfalls Kenntnisse über die Syntax der Sprache benötigt. Manche Java-Bücher beginnen deshalb mit der Syntax statt mit den OO-Konzepten. Ich bewerte allerdings das Verständnis des OO-Konzepts als die elementare Basis für einen Einstieg höher und möchte damit beginnen. Das ist auch aus meiner eigenen Historie beim Lernen von Java motiviert, denn ich bin von der prozeduralen auf die objektorientierte Programmierung umgestiegen und habe mich mit deren Konzepten sehr lange schwergetan. Mit den syntaktischen Strukturen hingegen kam ich sofort zurecht, da diese auch in prozeduralen Sprachen wie Pascal oder C ähnlich bis gleich sind. Und Java ähnelt von der Syntax her fast allen Sprachen, die zur C-Sprachfamilie zählen (wie etwa C, Perl, JavaScript, PHP etc.).

Die Syntax mit Details wie Datentypen und Variablen, Operatoren, Literalen, verschiedenen Anweisungstypen (Kontrollfluss-Anweisungen und Schleifen) etc. folgt im nächsten Kapitel, denn für ein funktionierendes Programm ist sie unabdingbar. In den Beispielen dieses Kapitels wird die Syntax aber natürlich benutzt, damit wir überhaupt etwas praktisch vorführen können. Allerdings werden die Beispiele grundsätzlich eine einfache Syntax besitzen und nur die OO-Konzepte anschaulich verdeutlichen.

Auf den folgenden Seiten erfahren Sie also Genaues zum grundsätzlichen Programmaufbau bei Java (Klassenbildung) und der praktischen Erzeugung von Objekten, dem Umgang mit Vererbung und Assoziationen sowie Modifizierern etc. Und warum sich OOP durchgesetzt hat und wie sie sich von klassischer Programmierung abgrenzt.

Etwas komplexere Konzepte der objektorientierten Programmierung verlagern wir dann aber hinter das folgende Syntax-Kapitel, um diesen beiden Säulen der Java-Programmierung gleichermaßen gerecht zu werden.

3.2 Objektorientierte Programmiersprachen

Die Geschichte der Programmierung geht von den theoretischen Ansätzen auf den Beginn des 19. Jahrhunderts zurück. Allerdings begann die praktische Umsetzung in der heute darunter verstandenen Form erst nach den Arbeiten von Konrad Zuse und der Von-Neumann-Maschine um 1945. Seit dieser Zeit wurde eine große Anzahl verschiedener Programmiersprachen entwickelt, die sich bestimmten Sprachfamilien zuordnen lassen. So werden die heute wichtigsten Programmiersprachen von der Syntax her meist zur C-Familie gezählt, die wiederum auf eine Sprache namens Algol zurückgeht.

3.2.1 Generationen

Programmiersprachen werden aber nicht (nur) bezüglich der Syntax klassifiziert, sondern ebenfalls historisch in verschiedene Generationen eingeteilt:

1. Die historisch erste Generation ist die **Maschinensprache** mit all ihren Facetten. Maschinensprache ist, wie bereits erörtert, explizit plattformabhängig bis hinunter zur Prozessorebene.

2. Als die zweite Generation wird **Assembler** gesehen. Assembler verwendet anstelle von numerischen Binärcodes symbolische Bezeichner (Mnemonics) für Anweisungen, die in genau einen Maschinenbefehl umgesetzt werden. Auch Assembler ist wie erwähnt plattformabhängig.

3. Mit der dritten Generation beginnen die **höheren Programmiersprachen**. Diese unterstützen Algorithmen und die Verwendung von Schlüsselwörtern, die der englischen Sprache entliehen sind und in Form von Text Anweisungen formulieren. Höhere Programmiersprachen sind weitgehend anwendungsneutral und (als Quelltext) plattformunabhängig. Diese Generation der Programmiersprachen beginnt Mitte der 50er Jahre mit Fortran, Cobol und Algol-60. Später kamen unter anderem Pascal, Modula-2, C und Basic hinzu.

4. Die vierte Generation der Programmiersprachen bezeichnet **anwendungsbezogene (applikative) Sprachen**. Solche Programmiersprachen ergänzen Techniken der dritten Generation um Sprachmittel für relativ komplexe, anwendungsbezogene Operationen. Dies sind beispielsweise Zugriffe auf Datenbanken (zum Beispiel mit SQL) oder die Gestaltung von grafischen Benutzeroberflächen (GUI).

5. Als fünfte Generation der Programmiersprachen werden solche Sprachen gesehen, die das relativ neutrale Beschreiben von Sachverhalten und Problemen erlauben und den genauen Problemlösungsweg nicht exakt vorgeben. Diese Sprachen werden im Rahmen der künstlichen Intelligenz eingesetzt.

3.2.2 Die Generation OO

Wenn Sie sich nun die Sprachgenerationen ansehen, werden Sie sich vielleicht fragen, wo objektorientierte Sprachen einzusortieren sind? Die Antwort ist einfach – sie passen nicht in dieses Generationenmodell. Sie werden daher oft als eigenständige **OO-Generation** bezeichnet, die sich bis Anfang der 70er Jahre und teilweise noch weiter zurückverfolgen lässt. Sehr frühe Vertreter mit objektorientiertem Denkansatz sind Lisp oder Logo. In den 70er Jahren entstand als wichtigster Vertreter erster objektorientierter Sprachen **Smalltalk**, das sich in mehreren Zyklen entwickelte und als einer der geistigen Väter von Java gilt. In den 80er Jahren entstanden mehrere OO-Sprachen, die bestehende, nicht objektorientierte Konzepte um objektorientierte Techniken erweiterten. Zum Beispiel wurde C um C++ erweitert und tritt seitdem meist als Paar auf (**C/C++**). C++ gibt es nicht als eigenständige, von C vollkommen

losgelöste Sprache. Seit Anfang der 90er Jahre entstanden eine Reihe moderner eigenständiger Programmiersprachen, die sich ausdrücklich als rein objektorientierte Programmiersprachen verstehen und mit prozeduralen Erblasten vollkommen brechen. Java zählt explizit dazu. Oder auch die meisten Sprachen unter dem .NET-Konzept von Microsoft, wie etwa C#. Diese Programmiersprachen unterstützen im Allgemeinen die wichtigsten Konzepte der objektorientierten Programmierung – allerdings in unterschiedlichem Maße.

3.2.3 Bessere Softwarequalität durch OOP

Das zentrale Problem bei der Entwicklung komplexer Softwaresysteme ist, eine möglichst hohe Softwarequalität zu erreichen. Allgemein betrachtet man dabei sowohl die innere als auch die äußere Softwarequalität. Die innere Softwarequalität bezeichnet die Sicht des Entwicklers. OOP bietet durch die Möglichkeiten der Abstraktion, Hierarchiebildung, Kapselung von Interna, Wiederverwendbarkeit, Schnittstellenbildung und einige weitere Techniken hervorragende Ansätze zur Gewährleistung einer hohen inneren Softwarequalität. Die äußere Softwarequalität spiegelt die Sicht des Kunden wider. Dieser erwartet Dinge wie die Korrektheit, Stabilität, Anwendungsfreundlichkeit oder Erweiterbarkeit einer Software. Ein Paradigma der OOP ist, dass eine hohe innere Softwarequalität zu einer hohen äußeren Softwarequalität führt. Und Java bestätigt dieses Paradigma eindrucksvoll.

3.2.4 Kernkonzepte der Objektorientierung

Als die theoretischen Kernkonzepte der objektorientierten Softwareentwicklung werden in den meisten Abhandlungen folgende Punkte angeführt:

- **Objekte** dienen als Abstraktion eines realen Gegenstands oder Konzepts und verfügen über einen spezifischen Zustand und ein spezifisches Verhalten. Man bezeichnet sie auch als **Instanzen** von Klassen, die durch **Konstruktoren** erzeugt werden.
- **Klassen** dienen als Baupläne für Objekte. Sie repräsentieren die Objekttypen. Auf der anderen Seite fassen Sie alle relevanten Eigenschaften und Verhaltensweisen der repräsentierten Objekttypen zusammen (sie ***klassifizieren*** diese).
- **Attribute** versteht man als Beschreibung objekt- und klassenbezogener Daten. Attribute sind die einzelnen Dinge, durch die sich ein Objekt von einem anderen unterscheidet. Man nennt Attribute meist die **Eigenschaften** eines Objekts.
- **Methoden** versteht man als Beschreibung der objekt- und klassenbezogenen Funktionalität. Sie repräsentieren das, was Objekte tun.
- **Assoziationen**, **Kompositionen** und **Aggregationen** sind Mechanismen zum Ausdruck von Klassen- bzw. Objektbeziehungen, wobei diese Schlagworte in Java allgemein und insbesondere in diesem Buch mehr theoretische als praktische Bedeutung haben. Eine Assoziation beschreibt in der Regel eine Beziehung zwischen zwei (meistens) oder mehr Klassen. Eine ganz besondere Assoziation, die wir konkret praktisch verwenden und auch

nur unter diesem Begriff weiter ausführen wollen, ist die **Vererbung**. Diese beschreibt einen Mechanismus zum Generalisieren und Spezialisieren von Objekttypen. In der objektorientierten Programmierung ist die Aggregation ebenso eine besondere Art der Assoziation – aber zwischen den Objekten selbst. Sie beschreibt eine sogenannte schwache Beziehung zwischen Objekten. Ein Objekt wird hier Teil eines anderen, ganzen Objekts gesehen. Es kann aber auch ohne das umgebende Objekt existieren. Im Fall einer Komposition ist ein Objekt auch Teil eines anderen, ganzen Objekts. Es kann jedoch nicht ohne das umgebende Objekt existieren.

- Die **Polymorphie** dient zur flexiblen Auswahl geeigneter Methoden (und teils auch anderer Syntaxstrukturen) identischen Namens anhand der Argumentenliste bzw. der gesamten Methodenunterschrift.
- **Schnittstellen** beschreiben einen Mechanismus zur Strukturierung von Klassenbeziehungen.
- Über **abstrakte Klassen** kann man bestimmte Operationen in spezialisierten Klassen erzwingen.
- **Generische Klassen** dienen zur Darstellung von ganzen Klassenfamilien.

Der Begriff Member

Ein wichtiger Begriff bei Klassen ist **Member** (Mitglied). Das ist eine Verallgemeinerung der Klassenbestandteile – also der Attribute und Methoden.

Wahrscheinlich werden diese Schlagworte am Anfang für Sie kaum Bedeutung haben, aber wir werden diese mit Substanz füllen. Neben der Syntax sind diese Konzepte eben der wesentliche Teil dessen, was Sie zum erfolgreichen Lernen von Java beherrschen müssen, den wir deshalb ins Zentrum des Buches stellen. Zum Teil besprechen wir die eben aufgeführten Konzepte bereits in diesem Kapitel, zum Teil aber auch erst in den folgenden Kapiteln des Buches.

3.2.5 Vertragsbasierte Programmierung

Nun soll Ihnen noch kurz ein Verfahren vorgestellt werden, das über weite Strecken der OOP im Allgemeinen und Java im Besonderen zum Tragen kommt. In vielen Situationen werden sogenannte **Verträge** bzw. **Kontrakte** zwischen dem Ersteller eines Codes und dessen Verwender geschlossen. Entweder in eigenen Sprachkonstrukten oder durch geeignete Programmierung. Vertragsbasierte Programmierung oder Kontrakt-Programmierung legt zwischen den aufrufenden und den aufgerufenen Stellen fest, welche Bedingungen vor der Verwendung und nach der Verwendung eines Codesegments gelten müssen. Bedingungen beziehen sich etwa auf den Zustand des aufgerufenen Objekts, die übergebenen Parameter und die Ergebnisse. Insbesondere legt ein Kontrakt fest, welche Eingangs- und Ausgangsbedin-

gungen bei einem Methodenaufruf sowie welche invarianten Eigenschaften im aufgerufenen Objekt gegeben sein müssen.

Bei einem Kontrakt werden einzelne Bestandteile unterschieden:

- **Preconditions** legen die Parameter und den Anfangszustand des aufgerufenen Objekts fest.
- **Postconditions** legen die Ergebnisse und den Zustand des aufgerufenen Objekts nach Ausführung des Aufrufs fest.
- **Invariants** überprüfen die Konsistenz des Zustands des aufgerufenen Objekts.

Ein Kontrakt wird in Java immer zwischen dem Aufruferobjekt (dem Client) und dem aufgerufenen Objekt (dem Supplier) geschlossen. Der Aufrufer kümmert sich um die Einhaltung der Preconditions. Dies bedeutet zum Beispiel sinnvolle Werte für die Argumentwerte bei einem Methodenaufruf. Der Supplier kümmert sich darum, dass bei Beendigung des Methodenaufrufs die Postconditions erfüllt sind. Dies bedeutet etwa, dass ein erwarteter Rückgabewert auch geliefert wird oder eine Ausnahme nur dann von einer Methode ausgelöst werden kann, wenn dies in der Methodensignatur dokumentiert wird. Dabei befindet sich das aufgerufene Objekt vor und nach dem Aufruf der Methode in einem gültigen Zustand.

Nun wird Ihnen dieses Konzept sehr wahrscheinlich etwas abstrakt erscheinen. Aber ich möchte Sie insofern beruhigen und auf die konkreten praktischen Umsetzungen neugierig machen, dass diese Verträge die Programmierung in Java einfach, sicher und logisch machen. Und gerade in Java wird durch das Laufzeitsystem und den Compiler streng auf die Einhaltung von Verträgen geachtet.

Deshalb ist Java so logisch, sicher und stabil. Ein Programmierer, der eine bestimmte Verhaltensweise eines Objekts bereitstellen will, muss das garantieren und der Verwender kann zu 100 % sicher sein, dass der Programmierer seine Versprechungen auch einhält.

3.3 Rund um Objekte, Instanzen und Klassen

In der OOP wird alles, was Sie jemals im Quelltext notieren, als Objekt bzw. Klasse oder ein Objekt-/Klassenbestandteil (oder eine Spezialform davon) zu verstehen sein. Das gilt nicht für sogenannte hybride Sprachen, mit denen man zwar objektorientiert programmieren kann, die jedoch mit ihrer prozeduralen Vergangenheit nicht vollständig gebrochen haben. Als Beispiele seien C/C++, Delphi, PHP oder Visual Basic genannt. Dort finden Sie aufgrund der historischen Altlasten Techniken, die nicht in das OO-Konzept passen und die dort parallel zu OO-Techniken verwendet werden können. Sogenannte objektbasierende Sprachen wie JavaScript verwenden Objekte, realisieren aber die verbindlichen Konzepte der OOP nur teilweise. Aber versuchen wir erst einmal genauer zu klären, was Objekte sind.

3.3.1 Objekte in der realen Welt

Lösen wir uns kurz von der Programmierung und betrachten die reale Welt. Objekte kommen ständig in unserer Umgebung vor. Ein Mensch bedient sich eines Objekts, um eine Aufgabe zu erledigen. Man weiß dabei in der Regel nicht genau, wie das Objekt im Inneren funktioniert, aber man kann es bedienen (weiß also um die verfügbaren Methoden, um es verwenden zu können), kennt die Eigenschaften des Objekts (die Attribute) und weiß, wann welche Funktionalität zur Verfügung steht (kennt den Zustand). Die Reihe von Beispielen kann man beliebig lang ausdehnen, wir wollen es beim Auto, der Waschmaschine, einem Kugelschreiber oder – natürlich – der Kaffeemaschine (es geht ja um Java) belassen.

Verfolgen wir unser Gedankenmodell mit einem ausgewählten realen Gegenstand weiter. Wenn Sie einen (realen) Kugelschreiber in die Hand nehmen, können Sie die Eigenschaften dieses (realen) Objekts beschreiben. Seine Form, die Größe, die Farbe. Sie können alle Eigenschaften (Attribute) des Objekts aufführen, soweit es notwendig bzw. sinnvoll ist. Und Sie können die aktiven Fähigkeiten des Stifts beschreiben, soweit auch dieses notwendig bzw. sinnvoll ist (Methoden). Etwa, dass Sie mit dem Stift schreiben können.

Dabei ist es offensichtlich, dass Sie mit dem Kugelschreiber nur dann schreiben können (eine aktive Fähigkeit), wenn die Mine vorher ausgefahren wurde. Und ebenso klar ist, dass eine Mine nur dann herausgedrückt werden kann, wenn sie zu diesem Zeitpunkt eingefahren ist und umgekehrt. Offensichtlich kann sich ein reales Objekt in einem spezifischen **Zustand** befinden und bestimmte Fähigkeiten in Abhängigkeit davon bereitstellen oder deaktivieren.

Ebenso ist ein Objekt in unserer Wahrnehmung immer nur die Abstraktion eines realen Gegenstands. Auch wenn das erst einmal sehr theoretisch klingt, bedeutet es nur, dass man immer selektiv die wirklich vorhandenen Eigenschaften und Methoden eines Objekts wahrnimmt. Dabei spricht man von **Abstraktion**, weil zur Lösung eines Problems normalerweise weder sämtliche Aspekte eines realen Elements benötigt werden, noch überhaupt darstellbar oder wahrnehmbar sind. Irgendwo muss immer abstrahiert werden.

Nehmen wir uns wieder unser reales Referenzobjekt. Jeder Kugelschreiber hat auch ein spezifisches Gewicht, einen bestimmten inneren Aufbau aus Mine und Feder bis hin zu einer im Prinzip angebbaren Anzahl an Atomen, aus denen er besteht. Ebenso können Sie mit dem Kugelschreiber auch einen Nagel in die Wand schlagen oder darauf Musik machen. Aber in der Regel interessieren Sie weder diese gerade genannten Eigenschaften noch nutzen Sie diese Methoden, die über ein »natürliches« Verständnis des Objekts hinausgehen – wir abstrahieren den realen Gegenstand, indem wir ihn in der Wahrnehmung auf nützliche Dinge reduzieren.

Die objektorientierte Denkweise im Computerumfeld entspricht vollkommen dem Abbild der realen Natur und den Objekten, die wir dort wahrnehmen. Die Objektorientierung versucht einfach, diese aus der realen Welt so natürlichen Vorstellungen in die EDV zu übertragen.

3.3.2 Objektorientierte Denkweise in der Benutzerführung

Wenn Sie sich den Desktop einer grafischen Betriebssystemoberfläche ansehen, repräsentieren alle dort zu findenden Symbole Objekte, die zu nützlichen Dingen abstrahiert wurden. Ein Objekt wird also als ein Symbol (ein Icon) dargestellt und dem Anwender ein Menü (zum Beispiel ein Kontextmenü) bereitgestellt, das die Möglichkeiten des Objekts und seine Eigenschaften – möglicherweise abhängig von einem Zustand – zugänglich macht. Bei einer grafischen Benutzeroberfläche setzt man die Darstellung für so eine zustandsabhängige Verhaltensweise dadurch um, dass man zu einem bestimmten Zeitpunkt nicht verwendbare Fähigkeiten in einem Menü deaktiviert und grau darstellt. Die aktivierbaren Fähigkeiten und der Zugang zu den Eigenschaften werden normal dargestellt. Betrachten Sie nur einmal das Symbol des Papierkorbs auf Ihrem Desktop.

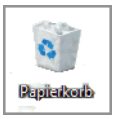


Abbildung 3.1: Das Symbol für ein Objekt mit spezifischen Informationen zu den relevanten Eigenschaften.

3.3.3 Objektorientierte Denkweise in der Programmierung

Objektorientierung kann nicht nur auf Anwendungsebene in einer GUI umgesetzt werden, sondern ebenso in der Programmierung. Unter einem Objekt stellt man sich in der EDV allgemein ein Softwaremodell vor, das ein Ding aus der realen Welt mit all seinen relevanten Eigenschaften und Verhaltensweisen beschreiben soll und das einem Anwender über ein Symbol oder einen Bezeichner zugänglich ist. Zum Beispiel das Objekt Drucker, Bildschirm oder Tastatur. Oder ein Objekt aus der realen Welt, das EDV-technisch abgebildet werden soll. Zum Beispiel ein Stift, ein Bankkonto, ein Bauernhof, ein Geldinstitut, ein Mensch. Auch Teile der Software selbst können ein Objekt sein. Ein Browser beispielsweise oder ein Teil davon – zum Beispiel ein Frame. Oder Teile der Verzeichnisstruktur eines Rechners. Zum Beispiel ein Ordner. Im Grunde ist in dem objektorientierten Denkansatz alles als Objekt zu verstehen, was sich eigenständig erfassen und ansprechen lässt.

Objektorientiert versus prozedural

In der althergebrachten Programmierung (ab der dritten Generation) programmiert man prozedural. Dieses bedeutet die Umsetzung eines Problems in ein Programm durch eine Folge von Anweisungen, die in einer festgelegten Reihenfolge auszuführen sind. Einzelne zusammengehörende Anweisungen werden dabei maximal in kleinere Einheiten von Befehlsschritten zusammengefasst (sogenannte Unterprogramme oder Module).

Ganz wichtig – die Datenebene und die Anweisungsebene kann aufgetrennt werden. Das Problem prozeduraler Programmierung ist, dass Änderungen in der Datenebene Auswirkungen auf die unterschiedlichsten Programmsegmente haben können und die Wiederverwend-

barkeit von Unterprogrammen sehr eingeschränkt ist, da oft Abhängigkeiten zu anderen Bestandteilen eines Programms existieren (beispielsweise zu globalen Variablen).

Objektorientierte Programmierung lässt sich im Vergleich zur prozeduralen Programmierung darüber abgrenzen bzw. definieren, dass zusammengehörende Anweisungen und Daten eine zusammengehörende, abgeschlossene und eigenständige Einheit bilden – eben Objekte!

Kernsatz der OOP

OOP hebt die Trennung von Daten- und Anweisungsebene auf!

Java setzt den objektorientierten Ansatz konsequenter um als viele vergleichbare Techniken. Java ist von der Philosophie her grundsätzlich objektorientiert.

Das hat massive Konsequenzen. Es gibt in Java beispielsweise keine globalen Variablen. Diese würden außerhalb von Objekten existieren und das ist explizit unmöglich.

Auch Strings und Arrays sind in Java Objekte. Ebenso werden Ereignisse, Ausnahmen oder Fehler durch spezifische Objekte repräsentiert.

Sogar ein Java-Programm selbst ist als Objekt zu verstehen. Es existiert im Hauptspeicher des Computers, solange das Programm läuft. In Java muss im Grunde nur eine Ausnahmesituation für die Aussage »Alles ist ein Objekt« beschrieben werden. Das sind sogenannte primitive Datentypen. Auf die hat Sun bei der Konzeption von Java nicht verzichtet, um prozedurale Programmierer nicht vollkommen vor den Kopf zu stoßen. Aber auch primitive Datentypen sind mittels sogenannter Wrapper-Klassen in das Konzept eingebunden. Damit werden für jeden primitiven Datentyp zugehörige Objekte bereitgestellt. Diese Details sollen allerdings zurückgestellt werden, denn primitive Datentypen zählen zu den syntaktischen Fragen, die später beantwortet werden.

Objekte und der Hauptspeicher

EDV-technisch sind Objekte bestimmte Bereiche im Hauptspeicher des Rechners, in denen zusammengehörige Informationen und Funktionalitäten gespeichert oder zugänglich gemacht werden.

3.3.4 Identifizieren Sie sich – Botschaften und Identifikatoren

Die Verwendung von Objekten in einer GUI erfolgt – wie wir besprochen haben – in der Regel über Kontextmenüs. Indem ein Anwender zum Beispiel mit der rechten Maustaste auf ein Objekt klickt und im Kontextmenü alle erlaubten Methoden und Eigenschaften angezeigt werden.

Im Rahmen eines Quelltextes muss ein Zugriff auf Objekte natürlich anders erfolgen. Man braucht einen Namen für das Objekt oder zumindest einen Stellvertreterbegriff, der ein Objekt repräsentiert (der **Identifikator**).

Egal, ob die Identität eines Objekts durch einen grafischen oder einen textnotierten Identifikator dargestellt wird – in der Regel ist der Identifikator eine Referenz (ein Zeiger bzw. Pointer) auf das tatsächliche Objekt. Meist handelt es sich bei einer solchen Referenz technisch um eine Hauptspeicheradresse, aber es kann auch anders umgesetzt werden. Das interessiert uns aus Sicht von Java aber grundsätzlich nicht. Wir müssen in Java nicht so tief in die Systemebene hinunter. Das ist ein zentraler Aspekt von Java und dessen virtueller Maschine.

Besonders wichtig ist im Zusammenhang mit der Verwendung von Objekten der Begriff der **Botschaften**. Damit man Objekte im Rahmen eines Quelltextes verwenden kann, tauschen sie sogenannte Botschaften (oder Nachrichten bzw. Messages) aus. In der strengen OOP werden ausschließlich Botschaften für die Kommunikation zwischen Objekten verwendet. Andere Kommunikationswege gibt es nicht.

Das Objekt, von dem man etwas will, erhält eine Aufforderung, eine bestimmte Methode auszuführen oder den Wert einer Eigenschaft zurückzugeben oder zu setzen. Das Zielobjekt versteht (hoffentlich) diese Aufforderung und reagiert entsprechend.

Punktnotation bzw. DOT-Notation

Die genaue formale Schreibweise solcher Botschaften ist in den meisten OO-Programmiersprachen nach dem folgenden Schema aufgebaut:

Empfängerobjekt [Methode oder Eigenschaft]

In den meisten OO-Sprachen trennt dabei ein Punkt die Bestandteile der Botschaft. Deshalb wird von der **Punktnotation** bzw. **DOT-Notation** gesprochen. Eine Botschaft, dass ein Objekt eine bestimmte Methode bereitstellen soll, sieht also meist so aus:

Empfängerobjekt.Methodenname(Argument)

Das Argument stellt in dem Botschaftsausdruck einen Übergabeparameter für die Methode dar.

Analoges gilt für die Verwendung von Objektattributen. In der Regel sieht das dann so aus:

Empfängerobjekt.Attributname

Die Anwendung der Punktnotation haben Sie in ***allen*** bisherigen Beispielen zu Java schon gesehen. Ich erinnere nur an die folgende Anweisung:

```
System.out.println("Hallo Welt");
```

Dem Objekt[1] `System` wird eine Nachricht geschickt, dass man dessen Unterobjekt `out` benötigt. Und diesem Unterobjekt `out` wird die Nachricht geschickt, dass man dessen Methode `println()` gern benutzen würde.

3.3.5 Ich denke, also bin ich

Aus programmiertechnischer Sicht ist es so, dass nicht nur der Programmierer weiß, was ein Objekt leistet. Auch das Objekt selbst weiß es und kann es nach außen dokumentieren. Es ist sich quasi seiner Existenz und seiner Fähigkeiten und Eigenschaften »bewusst«. Das wird in Java-Tools beispielsweise sehr umfangreich eingesetzt und äußert sich darin, dass der Compiler beim Übersetzen bereits vieles abfangen kann, was bei der Verwendung eines Objekts nicht erlaubt ist. Aber sogar schon vor der Kompilierung kann man diese Information nutzen. In geeigneten Entwicklungsumgebungen kann Ihnen ein Editor bereits Hilfe anbieten, indem er Ihnen bei einem Objekt alles anzeigt, was es leisten kann. Das haben Sie in den Beispielen mit Eclipse und NetBeans gesehen.

Und umgekehrt gilt, dass man von einem Objekt auch nur das fordern kann, was dort auftaucht.

In der OOP gibt es also eine Hilfe bzw. Kommunikation in zwei Richtungen. Der Programmierer sieht in einer geeigneten IDE sämtliche Eigenschaften und verfügbaren Methoden eines Objekts (samt dessen Zustand) und wird auf der anderen Seite sofort darauf hingewiesen, wenn er etwas vom Objekt anfordert, das dieses nicht bietet (falls zum Beispiel ein Schreibfehler bei einer Methode oder Eigenschaft gemacht wurde).

3.4 Klassen

Wenn nun Objekte die Basis der OOP sind und man in streng objektorientierten Sprachen wie Java ohne Objekte nichts machen kann – wie entstehen Objekte und was muss man als Programmierer konkret tun? Die Lösung heißt wie schon erwähnt Klassen (auch **Objekttyp** oder **abstrakter Datentyp** genannt) und darin enthaltene Konstruktoren. Diese müssen Sie schreiben und verwenden. Wenn man es genau betrachtet, besteht die gesamte Java-Programmierung daraus, Klassen zu schreiben und in ihnen Methoden und Eigenschaften zu definieren.

Erinnern Sie sich nur daran, dass Sie bei ***allen*** bisherigen Java-Beispielen das Schlüsselwort `class` verwendet haben. Das war der Beginn der Notation einer Klasse.

Das Schreiben einer Klasse, aber auch die Notation der darin enthaltenen Eigenschaften/Attribute und Methoden nennt man die **Deklaration** der Klasse.

1 Streng genommen ist `System` eine Klasse. Aber in Java kann man Klassen auch als Objekte betrachten – als Instanzen von Metaklassen. Das führen wir gleich noch aus.