

Einführung in die Informatik

Johannes Buchmann

15. März 2005

Inhaltsverzeichnis

1	Einleitung	9
2	Syntax und Semantik	13
2.1	Einführung	13
2.2	Mengen	13
2.3	Zahlen	14
2.4	Relationen	15
2.5	Funktionen	16
2.6	Zeichen und Worte	18
2.7	Terme	19
2.8	Zahldarstellung	20
2.9	Syntax von Termen	21
2.9.1	Typen	21
2.9.2	Literale	21
2.9.3	Operatoren	22
2.9.4	Vollständig geklammerte Terme	24
2.10	Syntax unvollständig geklammerter Terme	26
2.11	Semantik von U-Termen	27

3	Einfache Java-Programme	31
3.1	Der Rahmen	31
3.2	Anweisungen	32
3.2.1	Deklaration von Variablen	32
3.2.2	Deklaration von Konstanten	34
3.2.3	Zuweisungen	34
3.2.4	if-Anweisungen	34
3.2.5	switch-Anweisungen	36
3.2.6	while-Anweisung	37
3.2.7	for-Anweisung	38
3.2.8	break-Anweisung	39
3.2.9	continue-Anweisung	39
3.3	Arrays	40
3.4	Übungen	41
4	Algorithmen und ihre Effizienz	43
4.1	Algorithmen	43
4.2	Effizienz	44
4.3	Übungen	49
5	Einführung in die Objektorientierung	51
5.1	Einführung	51
5.2	Klassen und Objekte	53
5.2.1	Importieren von Packages und Klassen	53
5.2.2	Die Klasse <code>Rectangle</code>	54
5.2.3	Deklaration	54
5.2.4	Konstruktion	55

5.2.5	Attribute	55
5.2.6	Weitere Konstruktoren	56
5.2.7	Methoden	57
5.2.8	Implementierung	58
5.2.9	Implementierung der Variablen	59
5.2.10	Implementierung von Konstruktoren	59
5.2.11	Implementierung von Methoden	60
5.2.12	Datenabstraktion	61
5.3	Die Mouse-Events	61
5.4	Wie funktioniert das Applet?	62
5.5	Nach oben links Zeichnen	64
5.5.1	Vererbung	65
5.6	Ein komplexeres Beispiel	65
5.6.1	Zeichnen mit der <code>Graphics</code> -Klasse	66
5.6.2	Die Klassenhierarchie <code>Shapes</code>	66
5.6.3	Polymorphie	67
5.6.4	Die Unterklassen	68
5.6.5	Variablen des <code>DrawShapes</code> -Applet	69
5.6.6	Implementierung von <code>init</code>	70
5.6.7	Implementierung von <code>MousePressed</code>	70
5.6.8	Implementierung von <code>MouseDragged</code>	70
5.6.9	Implementierung von <code>MouseReleased</code>	70
5.6.10	Implementierung von <code>paint()</code>	70
5.6.11	Implementierung von <code>update()</code>	70

6	Verkettung	73
6.1	Motivation	73
6.2	Verkettete Listen	73
6.3	Pointer	74
6.4	Lineare verkettete Listen	75
7	Rekursion	81
7.1	Berechnung von Quadratsummen	81
7.1.1	Iterative Lösung	81
7.1.2	Analyse der Quadratsummenprogramme	83
7.2	Umkehr der Knoten in einer verketteten Liste	84
7.3	GGT	85
7.4	Der Turm von Hanoi	85
7.5	Merge-Sort	86
8	Modularität und Datenabstraktion	89
8.1	Einleitung	89
8.2	Warteschlangen	89
8.3	Anwendung von Warteschlangen	90
8.4	Implementierungen von Warteschlangen	91
8.5	Warteschlangen auf der Basis von verketteten Listen	92
8.6	Warteschlangen auf der Basis von Arrays	92
8.7	ComparisonKey	93
8.8	Klasse PQItem	94
8.9	Übungen	94

9	Stacks und Queues	95
9.1	Stacks	95
9.2	Kellerautomaten	95
9.2.1	Start	98
9.2.2	Zeichen lesen und pushen	98
9.2.3	Auswerten ohne Lesen	98
9.2.4	Auswerten bei schießender Klammer	99
9.2.5	Eingabe zuende	99
9.2.6	Endauswertung	99
9.3	ADT Stack	100
9.4	Stack-Implementierungen	100
9.5	Implementierung von Rekursion mit Stacks	100
9.6	Queues	101
9.7	Anwendungen von Queues	102
9.8	Übungen	102
10	Verifikation	103
10.1	Formaler Rahmen	103
10.2	Korrektheit von Methodenaufrufen	104
10.3	Schnelle Exponentiation	107
10.4	Selbstüberprüfung von Methoden	110
10.5	Übungen	110

11 Objektorientiertes Design	111
11.1 Die Idee	111
11.2 Objektorientiertes Design	111
11.3 Abstrakte Datentypen und Klassen	112
11.4 Vererbung	114
11.5 Mehrfachvererbung	114

Kapitel 1

Einleitung

Ziel dieser Vorlesung ist es, in die Informatik einzuführen. Insbesondere befassen wir uns mit der *Programmierung* von Computern. Viele Studierende haben schon Vorkenntnisse. Sie kennen Programmiersprachen und Computer, teilweise sogar besser als wir, die wir diese Vorlesung machen. Wir wollen ihr Wissen vertiefen und theoretisch fundieren.

Programmieraufgaben können sehr komplex sein. Ein Beispiel sind Programme, die Internet-Banking erlauben. So etwas ist eigentlich ein System von Programmen. Teile laufen bei der Bank und verwalten die Kundendaten. Teile laufen in Geschäften. Teile laufen bei Endkunden zu Hause. Welche Eigenschaften sollen solche komplexen Programme haben? Wir erläutern einige wichtige Eigenschaften.

Korrektheit Das Programm tut das, was es soll. Internet-Banking-Programme müssen z.B. die richtigen Summen überweisen oder abbuchen. Zuerst muß genau festgelegt werden, was das Programm tun soll. Man nennt das *Spezifikation*. Hat man die Spezifikation, dann muß man kontrollieren, ob das Programm sie erfüllt. Man kann versuchen, das mathematisch zu beweisen. Das nennt man *Verifikation*. Oder man konstruiert geeignete Testfälle. Testen komplexer Programme ist eine Wissenschaft für sich.

Testbarkeit Da man niemals sicher sein kann, ob ein komplexes Programm korrekt ist, muß man es so konstruieren, das es leicht getestet werden kann.

Robustheit Das Programm soll sich auch dann noch vernünftig verhalten, wenn es falsch bedient wird oder wenn andere Probleme auftreten. Wenn z.B. ein Benutzer eines Internet-Banking-Programms vergißt, seinen Namen einzugeben, darf das Programm nicht einfach abstürzen, sondern muß nach dem Namen fragen.

Erweiterbarkeit Es muß leicht sein, die Funktionalität des Programms zu erweitern. Wenn eine Bank in ihr Internet-Banking-Programm die Verwaltung von Wertpapierdepots einbeziehen will, muß das möglich sein, ohne daß das ganze Programm neu geschrieben wird.

Wiederverwendbarkeit Programmeinheiten müssen auch in anderen Zusammenhängen, in denen dieselbe Funktionalität gebraucht wird, verwendbar sein. In unserem Internet-Banking-Programm werden z.B. Überweisungsdaten verschlüsselt. Die Verschlüsselungsprogramme müssen z.B. auch für die Verschlüsselung von Email-Verkehr einsetzbar sein.

Kompatibilität Programme müssen mit unterschiedlichen Umgebungen verträglich (kompatibel) sein. Sie müssen z.B. ihre Ergebnisse auf verschiedenen Druckern oder Bildschirmen ausgeben können. Sie müssen also

Effizienz Programme müssen das, was sie tun, möglichst schnell tun. Wenn das Banking-Programm z.B. Zinseszinsberechnungen durchführt, soll der Benutzer darauf nicht einen halben Tag warten müssen.

Benutzerfreundlichkeit Der Benutzer muß das Programm leicht benutzen können.

Portabilität Das Programm soll leicht an neue Hard- und Softwareumgebungen angepaßt werden können. Wenn sich viele Benutzer des Internet-Banking-Programms einen neuen schnelleren Computer kaufen, dann muß die Bank ihnen ein angepaßtes Programm geben. Das soll nicht schwer zu machen sein.

Diese Qualitätsfaktoren konkurrieren teilweise miteinander. Je portabler ein System ist, desto weniger kann es die speziellen Eigenschaften eines besonderen Computers ausnützen und desto weniger effizient kann es werden. Bei der Systemkonstruktion muß man also zwischen den verschiedenen Qualitätskriterien abwägen. Interbanking-Programme sollten eher portabel sein. Echtzeitsteuerungen von Autos sollten eher effizient sein.

Wir werden in dieser Vorlesung eine Einführung in die formalen Methoden geben, die es ermöglichen, obige Ziele zu erreichen. Wir werden zeigen, wie man in der Sprache der Logik beschreibt, was korrekte Programme sind und was sie tun. Wir werden zeigen, wie man die Effizienz einfacher Programme analysiert und ihre Korrektheit überprüft. Wir werden zeigen, wie man die Methode der Objektorientierung benutzt, um große Programmsysteme zu konstruieren. Dabei werden wir als illustrierende Programmiersprache JAVA verwenden.

Die Methoden, die zur Verwendung kommen, sind mathematische, naturwissenschaftlich und ingenieurwissenschaftlich.

Ich veranschauliche das an einem Beispiel. Eins der Gebiete, das mich besonders interessiert, ist die Computersicherheit. Ein Problem aus diesem Gebiet: Alice und Bob wollen geheime elektronische Nachrichten austauschen.

Ein Ingenieur erhält den Auftrag, Alice und Bob dafür ein geeignetes System zur Verfügung zu stellen. Er konstruiert die Lösung. Er wählt die Computer und deren Verbindung aus. Er legt die Protokolle fest. Er programmiert eine Bedienungsführung.

Jetzt kommt der Naturwissenschaftler. Er experimentiert mit dieser Verbindung und versucht, geheime Nachrichten zu entschlüsseln. Vielleicht gelingt es ihm.

Jetzt könnte der Ingenieur nachbessern. Besser ist es, den Mathematiker zu fragen. Er macht ein Sicherheitsmodell. In seiner exakten Sprache beschreibt er, was es heißt, daß das System sicher ist. In seinem Modell entwirft er Verfahren, deren Sicherheit er zu beweisen versucht.

Die Ideen des Mathematikers benutzt der Ingenieur in seiner neuen Realisierung. Der Naturwissenschaftler untersucht die neuen Verfahren theoretisch und experimentell.

Auch in der Programmierung spielen die drei Aspekte eine wichtige Rolle. Ingenieure erfinden Programmiersprachen, die Menschen besonders einfach benutzen können. Mathematiker beschreiben, was korrekte Programme sind und was sie tun. Mathematiker und Ingenieure erfinden Lösungen von Programmieraufgaben. Die Lösungen werden mathematisch analysiert und experimentell untersucht.

Ein guter Informatiker sollte in allen drei Bereichen gut sein.

Kapitel 2

Syntax und Semantik

2.1 Einführung

2.2 Mengen

Mengen werden nicht formal definiert, weil das zu kompliziert ist.

Wir verstehen unter einer Menge eine Zusammenfassung von unterschiedlichen Dingen, die *Elemente* der Menge heißen.

Beispiele sind die Menge M aller geraden Zahlen. Die Elemente dieser Menge sind 2, 4, 6, 8, ... Wir schreiben dafür

$$M = \{2, 4, 6, 8, \dots\}.$$

Ist eine Zahl n eine gerade Zahl, also ein Element der Menge der natürlichen Zahlen, so schreiben wir

$$n \in M$$

und sagen “ n ist ein Element von M ”. Ist n aber keine natürliche Zahl, dann schreiben wir $n \notin \mathbb{N}$ und sagen “ n ist kein Element von M ”.

Eine andere Menge ist die Menge der Menschen mit dem Nachnamen Buchmann.

Eine grundlegende Zahlenmenge ist die Menge der natürlichen Zahlen. Sie heißt immer \mathbb{N} und ist

$$\mathbb{N} = \{1, 2, 3, 4, 5, 6, 7, \dots\}.$$

Eine andere Weise, die Menge der geraden Zahlen zu schreiben ist

$$\{n \in \mathbb{N} : n \text{ ist durch } 2 \text{ teilbar}\}.$$

Dies ist die Menge aller natürlichen Zahlen n mit der Eigenschaft, daß n durch 2 teilbar ist.

Allgemein ist

$$\{m \in M : m \text{ hat Eigenschaft } e\}$$

Die Menge aller derjenigen Elemente aus M , die die Eigenschaft e haben. Manchmal schreibt man auch einfach

$$\{m : m \text{ hat Eigenschaft } e\}$$

Das macht man wenn klar ist, aus welcher Menge m kommen soll.

Wir definieren jetzt die Vereinigung und den Durchschnitt zweier Mengen M und N .

Der Durchschnitt von M und N ist die Menge aller Elemente, die sowohl in M als auch in N sind. Sie wird $M \cap N$ geschrieben.

Ist

$$M = \{n \in \mathbb{N} : n \geq 10\}$$

und

$$N = \{n \in \mathbb{N} : n \text{ ist durch drei teilbar}\}$$

dann ist

$$M \cup N = \{3, 6, 9, 10, 11, 12, \dots\}$$

und

$$M \cap N = \{12, 15, 18, \dots\}$$

2.3 Zahlen

Wir führen einige Zahlenmengen ein, die häufig benutzt werden.

Die Menge der *natürlichen Zahlen* ist

$$\mathbb{N} = \{1, 2, 3, 4, 5, 6, \dots\}.$$

Die Menge der *ganzen Zahlen* ist

$$\mathbb{Z} = \{0, \pm 1, \pm 2, \pm 3, \pm 4, \pm 5, \pm 6, \dots\}.$$

Die Menge der *rationalen Zahlen* ist

$$\{p/q : p \in \mathbb{Z}, q \in \mathbb{N}\}.$$

Die Menge der *reellen Zahlen* wird mit \mathbb{R} bezeichnet und in der Analysis definiert.

2.4 Relationen

1. Definition Sei M eine Menge und sei $R \subset M \times M$ eine Relation auf M .

1. R heißt reflexiv, wenn mRm gilt für alle $m \in M$.
2. R heißt transitiv, wenn aus aRb und bRc stets aRc folgt für alle $a, b, c \in M$.
3. R heißt symmetrisch, wenn aus aRb stets bRa folgt für alle $a, b \in M$.
4. R heißt antisymmetrisch, wenn aus aRb und bRa stets $a = b$ folgt für alle $a, b \in M$.
5. Ist R reflexiv, transitiv und symmetrisch, dann heißt R Äquivalenzrelation. Für $m \in M$ heißt $[m]$ Äquivalenzklasse von R und $a \in [m]$ Vertreter von $[m]$.
6. Ist R reflexiv, transitiv und antisymmetrisch, so heißt R (partielle) Ordnung auf M .

2. Example Kongruenz.

3. Example Sei M eine Menge. Dann bezeichnet 2^M die Menge aller Teilmengen von M . Sie wird *Potenzmenge* von M genannt. Die Relation

$$R = \{(A, B) \in 2^M \times 2^M : A \subset B\}$$

ist eine partielle Ordnung auf 2^M .

4. Theorem Sei M eine Menge und $R \subset M^2$ ein symmetrisch und transitiv. Dann gilt für alle $a, b \in M$ entweder $[a] = [b]$ oder $[a] \cap [b] = \emptyset$.

Proof: Sei $a, b \in M$ und gelte $[a] \cap [b] \neq \emptyset$. Wir müssen zeigen, daß $[a] = [b]$ gilt.

Da $[a] \cap [b] \neq \emptyset$ gilt, gibt es ein $c \in [a] \cap [b]$. Wähle ein solches.

Wir zeigen zuerst, daß $[a] \subset [b]$ gilt. Dazu wählen wir $d \in [a]$ und zeigen $d \in [b]$ also bRd . Es gilt

1. aRd , weil $d \in [a]$,
2. cRa , weil $c \in [a]$ und R symmetrisch ist,
3. bRc , weil $c \in [b]$.

Aus der Transitivität folgt bRd , wie behauptet.

Die Inklusion $[b] \subset [a]$ zeigt man genauso.

Aus $[a] \subset [b]$ und $[b] \subset [a]$ folgt $[a] = [b]$, weil \subset eine Ordnung auf M ist. ■

5. Definition Sei S eine Menge von Mengen und sei M eine Menge. Die Menge M ist die disjunkte Vereinigung der Mengen aus S , wenn die Mengen in S paarweise disjunkt sind und wenn M die Vereinigung der Mengen in S ist.

6. Example Sei $S = \{\{0, 1\}, \{2\}, \emptyset\}$. Die Mengen in S sind paarweise disjunkt. Die Menge $M = \{0, 1, 2\}$ ist ihre disjunkte Vereinigung.

7. Corollary Sei M eine Menge und sei R eine Äquivalenzrelation auf M . Dann ist M die disjunkte Vereinigung der Äquivalenzklassen bezüglich R .

Proof: Übung. ■

8. Example Betrachte die Menge M aller Menschen und darauf die Relation, die aus allen Paaren von Menschen besteht, die am gleichen Tag Geburtstag haben. Das ist eine Äquivalenzrelation. Für jeden Geburtstag gibt es eine Äquivalenzklasse, die aus allen Menschen besteht, die an diesem Tag Geburtstag haben. Die Äquivalenzklassen sind paarweise disjunkt, weil Menschen aus verschiedenen Äquivalenzklassen an verschiedenen Tagen Geburtstag haben. Ihre Vereinigung ist die Menge aller Menschen, weil jeder Mensch an einem Tag Geburtstag hat.

2.5 Funktionen

Wir besprechen als nächstes Funktionen.

9. Definition Eine partielle Funktion ist ein Tripel $f = (A, B, R)$. Dabei sind A, B Mengen und R ist eine Relation zwischen A und B mit der Eigenschaft, daß für alle $a \in A$ die Menge $[a]$ höchstens ein Element enthält. Die Menge A heißt Argumentbereich von f und B heißt Wertebereich von f . Der Definitionsbereich von f ist $\{a \in A : |[a]| = 1\}$. Gehört a zum Definitionsbereich von f und ist $[b] = [a]$, so schreibt man $b = f(a)$ und bezeichnet $f(a)$ als Wert von f an der Stelle a . Man schreibt auch $f : A \rightarrow B$.

Eine (totale) Funktion ist eine partielle Funktion, deren Argumentbereich und Definitionsbereich gleich ist, die also für alle Argumente definiert ist.

10. Example Betrachte die partielle Funktion, die einem Paar ganzer Zahlen (x, y) den Wert $1/(x^2 - 2y^2 - 1)$ zuordnet, sofern der Nenner nicht Null ist. Dies ist eine partielle Funktion mit Argumentbereich \mathbb{Z}^2 , Wertebereich \mathbb{Q} . Man kann den Definitionsbereich nicht a priori angeben aber man kann für jedes einzelne Paar (x, y) prüfen, ob (x, y) zum Definitionsbereich der Funktion gehört.

11. Example Betrachte die partielle Funktion, die jedem Tripel (a, b, c) von ganzen Zahlen den kürzesten von Null verschiedenen Vektor (x, y) zuordnet, für den $ax^2 + bxy + cy^2 = 1$ gilt, falls es einen solchen Vektor gibt. Es ist noch nicht einmal klar, wie man testet, ob (a, b, c) zum Definitionsbereich gehört.

12. Example Die partielle Funktion, die jeder natürlichen Zahl ihr Quadrat zuordnet, ist total.

13. Definition Sind $f = (A, B, R)$ und $g = (B, C, Q)$ partielle Funktionen, dann heißt

$$g \circ f = (A, C, \{(a, c) : \text{Es gibt ein } b \in B \text{ mit } aRb \text{ und } bRc\})$$

die Komposition von f und g .

14. Example Sei f die Funktion, die einem Menschen sein Geburtsdatum zuordnet und sei g die Funktion, die einem Datum den entsprechenden Wochentag zuordnet. Dann ordnet $g \circ f$ einem Menschen seinen Geburtstag zu.

15. Definition Sei $f : A \rightarrow B$ eine partielle Funktion.

Sie heißt injektiv, falls zu verschiedenen Elementen aus dem Definitionsbereich immer verschiedene Funktionswerte gehören. Sind also x_1 und x_2 verschiedene Elemente des Definitionsbereichs von f , so gilt $f(x_1) \neq f(x_2)$.

Sie heißt surjektiv, falls für alle $y \in B$ ein x in Definitionsbereich von f existiert mit $f(x) = y$.

Ist f injektiv und surjektiv, so heißt f bijektiv.

16. Example Betrachte die Funktion f , die einem Paar (p, q) von Primzahlen ihr Produkt pq zuordnet. Sie ist injektiv, weil die Primfaktorzerlegung einer natürlichen Zahl eindeutig ist. Sie ist aber nicht surjektiv.

Wir beschreiben jetzt Umkehrfunktionen.

17. Example Betrachte die Funktion f aus Example 16. Deren Umkehrfunktion ordnet einer natürlichen Zahl ihre beiden Primfaktoren zu, falls sie nur zwei Primfaktoren hat.

18. Theorem Sei $f = (A, B, R)$ eine injektive partielle Funktion. Dann ist $f^{-1} = (B, A, Q)$ mit $Q = \{(b, a) : (a, b) \in R\}$ eine partielle Funktion.

2.6 Zeichen und Worte

Wenn man eine Sprache beschreiben will, braucht man Zeichen, aus denen die Wörter und Sätze der Sprache aufgebaut sind. Unter einem *Alphabet* versteht man einfach eine nicht leere Menge.

19. Example Beliebte Alphabete sind $\{0, 1\}$ oder $\{a, b, c, d, \dots, z\}$. Aber auch $\{\text{true}, \text{false}\}$ ist ein Alphabet.

Im folgenden sei Σ ein Alphabet.

20. Definition Sei Σ ein Alphabet.

1. Die Elemente von Σ heißen Zeichen, Buchstaben oder Symbole von Σ .
2. Als Wort oder String über Σ bezeichnet man eine endliche Folge von Zeichen aus Σ oder das sogenannte leere Wort ϵ .
3. Die Länge eines Wortes w über Σ ist die Anzahl seiner Zeichen. Sie wird mit $|w|$ bezeichnet. Das leere Wort hat die Länge 0.
4. Die Menge aller Worte über Σ einschließlich des leeren wird mit Σ^* bezeichnet.
5. Sind $v, w \in \Sigma^*$, dann ist der String $vw = v \circ w$ den man durch Hintereinanderschreiben von v und w erhält, die Konkatenation von v und w . Insbesondere ist $v \circ \epsilon = \epsilon \circ v = v$.
6. Sind u, v, w und $z = uvw$ Wörter über Σ , dann heißt u Präfix und v Suffix von z .
7. Eine Sprache über Σ ist eine Teilmenge von Σ^* .

21. Example Sei $\Sigma = \{0, 1\}$. Dann ist 001 ein Wort der Länge 3 über Σ . Es gibt 8 Wörter der Länge 3 über Σ .

Ist $v = 00$ und $w = 010$, dann ist $v \circ w = 00010$.

Ist $z = 001001001$ so sind 00, 001 und z selbst Suffixe von z und 001, 01 und v selbst sind Präfixe von z .

Die Menge aller Wörter über $\{0, 1\}$, die gleich viele Nullen und Einsen enthalten, ist eine Sprache über $\{0, 1\}$. Eine andere Sprache ist die Binärkodierung aller deutschen Sätze.

2.7 Terme

Beim Programmieren verwendet man Anweisungen von der Form

$$x = 3 + 4 * 5 + 6 * (2 + 3) \quad (2.1)$$

Der Variablen x wird dabei der Wert des Ausdrucks $3 + 4 * 5 + 6 * (2 + 3)$ zugewiesen. Ein solcher Ausdruck heißt *arithmetischer Ausdruck* oder *Term*.

Will man den Wert des Ausdrucks aus (2.1) ausrechnen, so formt man den Ausdruck um:

$$\begin{aligned} 3 + 4 * 5 + 6 * (2 + 3) &= 3 + 4 * 5 + 6 * 5 \\ &= 3 + 20 + 30 \\ &= 23 + 30 \\ &= 53. \end{aligned}$$

Dabei muß man aber wissen, daß Punktrechnung vor Strichrechnung geht. Man kann den Term aber auch vollständig klammern:

$$x = (3 + (4 * 5) + (6 * (2 + 3))) \quad (2.2)$$

Die Menge aller solcher Terme bildet eine Sprache.

Im folgenden beschreiben wir formal die Syntax und Semantik dieser Sprache. Wir geben also an, wie korrekte Terme aussehen und was sie bedeuten, d.h. was das Ergebnis ihrer Auswertung ist.

Man kann den Term aus (2.1) auch in Polnischer Notation schreiben:

$$2, 3, +, 6, *, 4, 5, *, +, 3, +$$

Bei der Auswertung geht man den Term von links nach rechts durch. In der Übung wird die Syntax und Semantik von Termen in Polnischer Notation besprochen.

2.8 Zahldarstellung

Eine Dezimalzahl ist

$$129$$

Sie bedeutet

$$1 * 100 + 2 * 10 + 9 * 1 = 1 * 10^2 + 2 * 10^1 + 9 * 10^0.$$

Dieselbe Zahl kann man auch binär schreiben als

$$128 + 1 = 1 * 2^7 + 0 * 2^6 + 0 * 2^5 + 0 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 10000001.$$

Man kann sie auch oktal schreiben als

$$2 * 8^2 + 1 = 201$$

Schließlich kann man sie auch hexadezimal schreiben, also zur Basis 16. Dann wird 129 zu

$$8 * 16 + 1 = 81$$

Will man 28 hexadezimal darstellen, erhält man

$$28 = 1 * 16 + 12 * 1.$$

Für 12 braucht man eine neue Ziffer. Man setzt

Ziffer	A	B	C	D	E	F
Bedeutung	10	11	12	13	14	15

Die Großbuchstaben können auch gegen die Kleinbuchstaben ersetzt werden. Dann ist die Hexadezimalentwicklung von 28 einfach 1C.

Solche Entwicklungen sind zu jeder natürlichen Basis g möglich und eindeutig. Es gilt nämlich der folgende Satz:

22. Theorem *Sei g eine natürliche Zahl, $g > 2$. Für jede natürliche Zahl a gibt es eine eindeutig bestimmte natürliche Zahl k und eine eindeutig bestimmte Folge*

$$(a_1, \dots, a_k) \in \{0, \dots, g-1\}^k$$

mit $a_1 \neq 0$ und

$$a = \sum_{i=1}^k a_i g^{k-i}. \quad (2.3)$$

23. Definition Die Folge aus Theorem 22 heißt g -adische Entwicklung von a . Ihre Länge ist $k = \lfloor \log_g a \rfloor + 1$. Falls $g = 2$ ist, heißt die Folge aus Theorem 22 Binärentwicklung von a . Falls $g = 16$ ist, heißt die Folge Hexadezimalentwicklung von a .

24. Example Die Folge 10101 ist die Binärentwicklung der Zahl $2^4 + 2^2 + 2^0 = 21$. Wenn man Hexadezimaldarstellungen aufschreibt, verwendet man für die Ziffern 10, 11, ..., 15 die Buchstaben A, B, C, D, E, F, G . So ist $A1C$ die Hexadezimaldarstellung von $10 * 16^2 + 16 + 12 = 294$.

Die Länge der Binärentwicklung einer natürlichen Zahl wird auch als ihre *binäre Länge* bezeichnet. Die binäre Länge von 0 wird auf 1 gesetzt. Die binäre Länge einer ganzen Zahl ist die binäre Länge ihres Absolutbetrags. Die binäre Länge einer ganzen Zahl a wird auch mit $\text{size } a$ bezeichnet.

2.9 Syntax von Termen

2.9.1 Typen

Wir legen zuerst fest, daß in den Termen Konstanten mit folgenden Typennamen vorkommen:

int, long, float, double, boolean, char.

Es gibt auch noch die Typen short, byte. Die spielen aber eine Sonderrolle.

Dies Typen werden in Java verwendet. Wir haben aber noch nicht gesagt, was die Typennamen bedeuten, obwohl wir es uns schon denken können. Die Menge der Typen bezeichnen wir mit S .

2.9.2 Literale

Zu allen Typen gibt es Konstantennamen, also Bezeichner für Konstanten. Bei den Grundtypen heißen die Konstantennamen *Literale*. In dem Term

$$3 + 4 * 5 + 6 * (2 + 3)$$

kommen zum Beispiel die Literale 2,3,4,5,6 vor.

Hier ist eine Liste der Literale für den Typ int. Für $i \in \mathbb{N}$ setzen wir

$$M_i = \{-2^i, \dots, 2^i - 1\}$$

Literale	Beispiel
Dezimalentwicklungen von Zahlen in M_{32}	1261
Oktalentwicklungen von Zahlen in M_{32} mit führender 0	01261
Hexadezimalentwicklungen von Zahlen in M_{32} mit führendem 0x oder 0X	0xbad

Literale für long werden fast genauso gebildet. Man nimmt statt M_{32} die Menge M_{64} und hängt an alle Dezimal-, Oktal-, oder Hexadezimalentwicklungen ein L dran.

Um zu testen, welche Literale für jeden Typ erlaubt sind (und was sie bedeuten) kann man folgendes Java-Programm verwenden

```
public class test {
    public static void main(String[] args) {
        int x = 198;
        System.out.print(x);
        System.out.print("\n");
    }
}
```

Literale vom Typ double und float beschreiben Dezimalbrüche mit Exponenten. Sie sehen wie folgt aus.

1. Dezimalbrüche, z.B. 123.654, allgemein $a.b$ mit $a, b \in \{0, 1, \dots, 9\}^*$. Dabei unterliegen a und b aber noch Größenbeschränkungen, die ich aber nicht herausfinden konnte.
2. Dezimalbrüche gefolgt von Exponenten, z.B. $123.654e - 3$ oder $123.654E4$, allgemein den oder dEn wobei d ein Dezimalbruch und n eine ganze Zahl ist.

Literale vom Typ float sind Literale vom Typ double mit angehängtem f oder F . Es gelten auch andere Größenbeschränkungen als für double.

Literale vom Typ boolean sind true und false. Literale vom Typ char sind von der Form 'x', wobei x ein einzelnes Unicode-Zeichen ist.

2.9.3 Operatoren

In Termen darf man auch Operatoren verwenden. In einfachen Java-Programmen sind das die Operatoren

*, /, %, +, -, ==, !=, <, <=, >, >=, &&, ||, !

Dies sind sämtlich *Infixoperatoren*, weil sie zwischen die Operanden geschrieben werden. Eine andere Art von Operatoren sind die *Präfixoperatoren*, z.b.

min, max, log, sin, sqrt

Man schreibt z.B.

min(3,2), log 10.

Präfixoperatoren werden vor die Operanden geschrieben. Die Operanden werden eingeklammert.

Für jeden Operator muß man festlegen, auf welchen Typen er arbeiten soll und von welchem Typ sein Ergebnis sein soll. Der Operator $*$ nimmt z.B. zwei Operanden vom Typ `int` und macht daraus ein `int`. Oder er nimmt zwei Operanden vom Typ `double` und macht daraus ein `double`. Es ist sogar möglich, zwei Operanden vom Typ `char` zu multiplizieren. Aber es ist nicht möglich zwei Operanden vom Typ `boolean` zu multiplizieren.

Wir beschreiben die Typfestlegung formal. Wir bezeichnen mit Ω die Menge der Operatoren, die in den Termen vorkommen dürfen. Zur Typfestlegung benutzen wir eine partielle Funktion

$$\varphi : \Omega \times S^* \rightarrow S.$$

Sie ordnet einem Operator und einer Folge von Typen einen Ergebnistyp zu.

Hier ist eine Tabelle, die einige Werte dieser partiellen Funktion φ angibt:

$\omega \in \Omega$	$s \in S^*$	$\varphi(\omega, s)$
+	int int	int
+	float float	float
*	double double	double
&&	boolean boolean	boolean
<	int int	boolean
max	float float	float

Da $(+, \text{int}, \text{int})$ zum Definitionsbereich von φ gehört ist (int, int) eine Signatur des Operators $+$.

Im allgemeinen wird die Signatur eines Operators folgendermaßen definiert.

25. Definition Sei $\omega \in \Omega$. Eine Signatur von ω ist ein String $s \in S^*$ für den (ω, s) zum Definitionsbereich von φ gehört.

Kann ein Operator als Infixoperator verwendet werden, so wird er immer als Infixoperator verwendet. Seine Signaturen sind sämtlich in S^2 . Die Menge der Infixoperatoren wird mit Ω_I bezeichnet.

Auch die Konstantennamen, also die Literale werden als Operatoren aufgefaßt. Die Signatur einer Konstante ω ist ϵ und es gibt keine andere Signatur. Der Typ von ω ist $\varphi(\omega, \epsilon)$.

Es gibt also insgesamt Konstanten, Infixoperatoren und Präfixoperatoren. Bei Präfixoperatoren sind die Signaturen $\neq \epsilon$.

2.9.4 Vollständig geklammerte Terme

Wir beschreiben jetzt die vollständig geklammerten Terme. Dies geschieht induktiv.

1. Jeder Konstantenname (Literal) vom Typ s ist ein Term vom Typ s .
2. Wenn ω ein Infixoperator mit der Signatur $(s_1 s_2)$ ist und wenn t_i ein Term vom Typ s_i ist, $i = 1, 2$, dann ist $(t_1 \omega t_2)$ ein Term vom Typ $\varphi(\omega, s_1 s_2)$.
3. Wenn ω ein Präfixoperator mit der Signatur $(s_1 s_2 \dots s_k)$ ist und wenn t_i ein Term vom Typ s_i ist, $i = 1, 2, \dots, k$, dann ist $\omega(t_1, t_2, \dots, t_k)$ ein Term vom Typ $\varphi(\omega, s_1 s_2 \dots s_k)$.

26. Example Wir zeigen die Erzeugung des Terms $((3 + 4) < \max(3, 4))$.

1. 3 und 4 sind Literale vom Typ int , also Terme vom Typ int .
2. $(3 + 4)$ ist ein Term vom Typ int , weil $+$ ein Infixoperator der Signatur (int int) ist, und weil $\varphi(+, \text{int int}) = \text{int}$ ist.
3. $\max(3, 4)$ ist ein Term vom Typ int , weil \max ein Präfixoperator von der Signatur (int int) und $\varphi(\max, \text{int int}) = \text{int}$ ist.

Man kann die Ableitung des Terms $(3 + 4) < \max(3, 2)$ auch nachträglich rekonstruieren.

Dabei ent der Ableitungsbaum des Terms.

Ableitungsbaum.

27. Exercise Ableitungsbaum und Ableitung zu gegebenem Term.

Es ist jetzt immer noch möglich, daß ein und derselbe Term auf unterschiedliche Weise erzeugt wird und das darum sein Typ nicht eindeutig ist. Wir werden jetzt zeigen, daß das unmöglich ist.

28. Theorem *Jeder vollständig geklammerte Term hat einen eindeutig bestimmten Ableitungsbaum und damit auch einen eindeutig bestimmten Typ.*

Proof: Als Beweismethode wird die vollständige Induktion benutzt.

Sie erlaubt es, Behauptungen für alle natürlichen Zahlen (ab einer bestimmten Stelle) zu beweisen.

Wir beweisen die Behauptung durch Induktion über die Anzahl der im Term vorkommenden Operationszeichen. Wir beweisen also genauer gesagt folgende Behauptung:

Für alle natürlichen Zahlen n gilt: Ist t ein Term, in dem n Operationszeichen vorkommen, so ist der Ableitungsbaum von t eindeutig, es gibt also keine zwei verschiedenen Konstruktionen von t , die zu unterschiedlichen Typen führen.

Induktionsverankerung: Zuerst wird die Behauptung für die erste natürliche Zahl bewiesen, für die sie gelten soll. In diesem Fall ist das $n = 1$. Terme, die nur einen Operator enthalten, sind Konstanten. Sie haben offensichtlich einen eindeutig bestimmten Ableitungsbaum.

Induktionsschritt: Wir machen jetzt die Induktionsannahme, daß $n > 1$ ist und daß die Behauptung für alle natürlichen Zahlen $i < n$ gilt. Wir beweisen die Behauptung dann für n .

Wir nehmen also einen Term t , der genau $n > 1$ Operationszeichen enthält. Dieser Term ist keine Konstante. Er ist also durch einen der beiden Konstruktionsschritte entstanden.

Es gibt zwei Möglichkeiten.

Der Term ist von der Gestalt

$$\omega(t_1, \dots, t_k)$$

mit Termen t_1, \dots, t_k und einem Präfixoperator ω . Dann wurde die dritte Regel verwendet, um den Term zu konstruieren und nicht die zweite. Die Terme t_i , $1 \leq i \leq k$ enthalten weniger Operationszeichen. Ihr Ableitungsbaum ist eindeutig. Also ist auch der Ableitungsbaum von t eindeutig. (Bild)

Der Term ist von der Gestalt

$$(t_1 \omega t_2)$$

mit Termen t_1, t_2 und einem Infixoperator ω . Dann wurde die zweite Regel verwendet, um den Term zu konstruieren und nicht die dritte. Die Terme t_i , $1 \leq i \leq 2$ enthalten weniger Operationszeichen. Ihr Ableitungsbaum ist eindeutig. Also ist auch der Ableitungsbaum von t eindeutig. ■

2.10 Syntax unvollständig geklammerter Terme

Wenn man die vielen Klammern nicht will, muß man für die Infixoperatoren Prioritäten festlegen.

Hier ist ein Beispielterm, bei dem man die Prioritäten der Operatoren ablesen kann.

$$2 + 3 * 4 < 6 \&\& 2 > 4 || true.$$

Vollständig geklammert sieht das

29. Exercise Vollständige Klammerung erzeugen zu gegebenem unvollständig geklammertem Term.

30. Exercise Geben Sie einen ungeklammerten Term an, in dem nur true, false, —, && vorkommt und der je nach Priorität von — und && ein verschiedenes Ergebnis hat.

Lösung: true || false && false

Testen Sie damit die Priorität in Java.

Folgende Tabelle beschreibt die Prioritäten der in unseren Beispielen benutzten Infixoperatoren:

Operator	Priorität
*, /, +, -	3
i, i=, i, i=	2
&&	1
	0

Im allgemeinen braucht man eine Funktion

$$\pi : \Omega_I \rightarrow \{0, 1, \dots, |\Omega_I| - 1\}.$$

Für $\omega \in \Omega_I$ heißt $\pi(\omega)$ die *Priorität* von ω . Wir setzen ferner

$$p = |\Omega_I|.$$

Das ist die allerhöchste Priorität. Mit der werden Präfixoperatoren ausgewertet. In dem Term

$$\max(3, 4) * 2 + 3 < 4$$

wird natürlich erst das max ausgewertet.

Es folgen die Regeln zur Konstruktion unvollständig geklammerter Terme. Jedem Term t wird auch eine Priorität $\pi(t)$ zugeordnet.

1. Jeder Konstantenname (Literal) vom Typ s ist ein Term vom Typ s und von der Priorität p .
2. Wenn t ein Term vom Typ s ist, dann ist (t) ein Term vom Typ s und von der Priorität p .
3. Wenn ω ein Infixoperator mit der Signatur $(s_1 s_2)$ ist und wenn t_i ein Term vom Typ s_i ist, $i = 1, 2$, und wenn $\pi(t_1) \geq \pi(\omega)$ und $\pi(t_2) > \pi(\omega)$, dann ist $t_1 \omega t_2$ ein Term vom Typ $\varphi(\omega)$ und von der Priorität $\pi(\omega)$. dann ist $(t_1 \omega t_2)$ ein Term vom Typ $\varphi(\omega, s_1 s_2)$ und mit der Priorität $\pi(\omega)$.
4. Wenn ω ein Präfixoperator mit der Signatur $(s_1 s_2 \dots s_k)$ ist und wenn t_i ein Term vom Typ s_i ist, $i = 1, 2, \dots, k$, dann ist $\omega(t_1, t_2, \dots, t_k)$ ein Term vom Typ $\varphi(\omega, s_1 s_2 \dots s_k)$ und von der Priorität p .

Wie bei den vollständig geklammerten Termen kann man auch für die unvollständig geklammerten Terme einen Ableitungsbaum hinschreiben. Wir werden im nächsten Abschnitt beweisen, daß dieser Ableitungsbaum durch den Term eindeutig bestimmt ist.

2.11 Semantik von U-Termen

Wir wollen jetzt beschreiben, wie man Termen einen Wert zuordnet. Damit erklären wir also die Semantik von Termen.

Zuerst brauchen wir für jeden Typ eine sogenannte *Trägermenge*. Sie enthält die Konstanten dieses Typs. Es gilt

$$\begin{aligned}
 M(\text{int}) &= M_{32} \\
 M(\text{boolean}) &= \{\text{true}, \text{false}\} \\
 M(\text{float}) &= \mathbb{R} \\
 M(\text{double}) &= \mathbb{R} \\
 M(\text{char}) &= \{\text{alle Unicode-Zeichen}\}.
 \end{aligned}$$

Man sieht daran, daß die Syntax schon im Hinblick auf die Semantik erzeugt wurde.

Als nächstes definieren wir eine Interpretationsfunktion, die die Operatorzeichen interpretiert.

Diese Funktion heißt g .

Für jeden Konstantennamen ω ist $g(\omega)$ die entsprechende Konstante. $g(\omega)$ gehört zur Trägermenge des Typs von ω .

So ist z.B. $g(123L) = 123$.

Für jeden Operator ω , der keine Konstante ist, interpretiert g die Wirkung des Operators. Der $+$ -Operator macht aus zwei ints deren Summe, wieder ein int, sofern sie Summe in M_{31} ist. $g(+, \text{int int})$ ist also eine partielle Funktion. Sie ordnet einem Paar von Konstanten vom Typ int dessen Summe zu, sofern die Summe erklärt ist.

Allgemein ist für einen Operator ω und eine Signatur $s_1 s_2 \dots s_k$ von Ω der Funktionswert $g(\omega, s_1 \dots s_k)$ eine partielle Funktion

$$M(s_1) \times \dots \times M(s_k) \rightarrow M(\varphi(\omega, s_1 \dots s_k)).$$

Vielleicht weniger bekannt ist die Interpretation der boolean-Operatoren. Die folgenden Tabellen interpretieren das logische Oder $||$ und das logische Und $\&\&$. Es sind die Tabellen der Funktionen $g(\&\&, \text{boolean boolean})$ und $g(||, \text{boolean boolean})$

x	y	$x\&\&y$	$x y$
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Die Bedeutung eines Terms t wird mittels seiner Konstruktion festgelegt. Sie wird mit $g(t)$ bezeichnet. Sie wird folgendermaßen definiert

1. Für Konstanten ω liegt $g(\omega)$ bereits fest.
2. Sei ω ein Infixoperator und $t = t_1 \omega t_2$ ein daraus konstruierter Term ist. Der Term t_i sei vom Typ s_i , $i = 1, 2$. Dann ist $g(t) = g(\omega, s_1 s_2)(g(t_1), g(t_2))$.
3. Sei ω ein Präfixoperator und $t = \omega(t_1, t_2, \dots, t_k)$ ein daraus gebildeter Term. Sei t_i ein term vom Typ s_i , $1 \leq i \leq k$. Dann ist $g(t) = g(\omega, s_1 \dots s_k)(g(t_1), \dots, g(t_k))$.

Wie kann man diese Definition benutzen, um einen fertigen Term auszuwerten? Man bestimmt den Ableitungsbaum des Terms und wertet den Term dann von unten nach oben aus.

Wir erläutern das an einem Beispiel. Der Term ist

$$t = 1 + 2 - 3 + 4.$$

Alle Operatoren in dem Term haben Priorität 3. Was war die letzte Konstruktionsvorschrift, die bei der Erzeugung dieses Terms verwendet wurde? Es war die zweite Konstruktionsvorschrift. Und worauf wurde die angewandt? Aus das am weitesten links stehende $+$, weil nur so links ein Term steht, der höhere Priorität hat als $+$,

nämlich die Konstante 1, die die Priorität 5 hat. Man kommt dadurch also zu dem Ableitungsbaum.

Ableitungsbaum.

Den wertet man von unten nach oben aus gemäß der obigen Regeln.

Wäre nicht gefordert, daß der linke Term eine höhere Priorität hat als der zur Debatte stehende Operator, dann wäre der Ableitungsbaum nicht eindeutig, sondern könnte auch so aussehen:

Alternativer Ableitungsbaum.

Dessen Wert wäre dann ein anderer: Auswertung.

Daher ist die Eindeutigkeit des Ableitungsbaums also wirklich wichtig.

Deswegen beweisen wir also jetzt diese Eindeutigkeit und geben gleichzeitig ein Konstruktionsverfahren für den Ableitungsbaum und damit eine Methode zur Auswertung an.

31. Theorem *Der Ableitungsbaum eines unvollständig geklammerten Terms ist eindeutig bestimmt.*

Proof: Um den Satz zu beweisen brauchen wir den Begriff des ungeklammerten Infixoperators. Wir nennen also einen Infixoperator ω *ungeklammert* in einem Term t , wenn die Anzahl der öffnenden Klammern links von ω in t genauso gross ist wie die Anzahl der schließenden Klammern links von ω .

Wir beweisen die Behauptung durch vollständige Induktion. Genauer beweisen wir folgende Aussagen:

Sei t ein Term, der n Operatoren enthält. Dann gilt folgendes:

1. Sein Ableitungsbaum ist eindeutig.
2. Seine Priorität ist entweder p , wenn t keine ungeklammerten Infixoperatoren enthält oder die Priorität des ungeklammerten Infixoperators von niedrigster Priorität.

$n = 1$: t ist eine (vielleicht eingeklammerte) Konstante. Daher gelten die Behauptungen.

$\leq n \rightarrow n$: Sei t ein Term mit n Operationszeichen und gelte die Behauptung für alle Terme mit weniger als n Operationszeichen.

Wir können annehmen, daß t nicht eingeklammert ist. Ist $t = \omega(t_1, \dots, t_k)$, so ist die Behauptung ebenfalls wahr. Ist t nicht von dieser Form, dann muß t mittels der dritten Regel entstanden sein. Daher enthält t ungeklammerte Infixoperatoren. Welche Eigenschaften hat der Infixoperator, auf den die dritte Regel angewendet wurde?

Er muß bei den ungeklammerten Infixoperatoren von niedrigster Priorität sein. Andernfalls gibt es einen ungeklammerten Infixoperator von niedrigerer Priorität. Dann ist

$$t = t_1 \omega t_2.$$

Darin enthält entweder t_1 oder t_2 einen ungeklammerten Infixoperator von niedrigerer Priorität als t . Nach Induktionsannahme ist damit die Priorität von t_1 oder t_2 niedriger als die Priorität von ω und das geht nicht wegen der Konstruktionsvorschrift.

Er muß der ungeklammerte Infixoperator mit niedrigster Priorität sein, der am weitesten rechts steht. Das Argument ist dasselbe wie oben. ■

Kapitel 3

Einfache Java-Programme

In diesem Kapitel erklären wir die Syntax und Semantik einfacher Java-Programme.

3.1 Der Rahmen

Einfache Java-Programme sind von folgender Gestalt

```
public class Simple {
    public static void main (String args[])
        < Block >
}
```

Wir interessieren uns im Augenblick nicht für den Rahmen sondern nur für den Block. Der wird nämlich ausgeführt.

Ein Beispiel:

```
public class Simple {
    public static void main (String args[]) {
        int x;
        x = 2;
        int i;
        for (i=1; i <= 10; i = i+2) {
            x = x+x;
        }
        System.out.print(x);
        System.out.print("\n");
    }
}
```

Ein Block ist von der Form

$$\text{Block} = \{\text{Anweisungsfolge}\}$$

Eine Anweisungsfolge ist von der Form

$$\text{Anweisungsfolge} = s_1; s_2; \dots; s_k$$

Hierin sind s_i Anweisungen für $1 \leq i \leq k$. Anweisungen werden jetzt erklärt. Die Anweisungen werden nacheinander abgearbeitet.

3.2 Anweisungen

3.2.1 Deklaration von Variablen

Wir haben bereits gesehen, daß es in Java die primitiven Datentypen `int`, `long`, `float`, `double`, `char`, `boolean` gibt.

Eine Deklaration einer Variablen mit Namen `x` vom Typ `int` sieht so aus:

```
int x
```

Sie bindet den Namen `x` an eine Speicherstelle, die eine Konstante vom Typ `int` speichern kann.

Allgemein sieht eine Deklaration so aus

```
<Typ> <Name>
```

Dabei ist `Typ` einer der erwähnten Typen. Später kann man auch selbst noch Typen erzeugen. Dann können auch die eigenen Typen hier erscheinen. `Name` ist ein Wort über dem Unicode-Alphabet mit Ausnahme der reservierten Wörter und der bereits in Deklarationen verwendeten Namen. Reserviert ist z.B. `int`. Eine aller Liste reservierten Wörter findet man in in den VL-Infos.

Sie erzeugt eine Variable vom Typ `Typ` und bindet die an eine Speicherzelle, die eine Konstante vom Typ `Typ` enthalten kann.

Damit man Variablen auch in Terme einbauen kann, geht man so vor: Für jeden Typ $s \in S$ führt man zusätzlich einen neuen Typ (`var, s`) ein. Wir haben also jetzt neben den Typen

int, char, boolean, float, double, long

die Typen

(var,int), (var,char), (var,boolean), (var,float), (var,double),
(var,long).

Mit der Deklaration

```
int x
```

entsteht das Literal x vom Typ (var,int). Allgemein entsteht mit der Deklaration

<Typ> <Name>

das Literal <Name> vom Typ (var,Typ). Die Literale vom Typ (var,Typ)

Die Operatoren erhalten jetzt noch zusätzliche Signaturen. Hat ein Operator ω die Signatur (s_1, \dots, s_k) , dann erhält er jetzt zusätzlich alle Signaturen (s'_1, \dots, s'_k) , wobei $s'_i \in \{s_i, (\text{var}, s_i)\}$ ist für $1 \leq i \leq k$. Das sind also 2^k Signaturen. Der Typ des Ergebnisses ändert sich nicht.

Wenn also die Deklarationen

```
int a;  
int b;
```

in einem Block vorgekommen sind, dann ist $a + b$ ein unvollständig geklammerter Term.

Der Wert eines solchen Terms ist folgendermaßen definiert.

1. Der Wert einer Variablen ist die Konstante, die im Speicherplatz steht, der dieser Variablen zugeordnet ist.
2. Den Wert eines Terms erhält man auf dieselbe Weise

3.2.2 Deklaration von Konstanten

Eine Konstante vom Typ `int` deklariert man mit

```
final int x = 5;
```

Dadurch wird die Konstante `x` mit dem Wert 5 erzeugt.

Im allgemeinen sehen Konstantendeklarationen so aus:

```
final <Typ> <Name> = <Term vom Typ Typ>
```

Dadurch wird eine Konstante vom Typ `Typ` erzeugt.

3.2.3 Zuweisungen

Eine Zuweisung ist eine Anweisung von der Form

$$i \text{ Variable vom Typ } s; = \text{ Term vom Typ } s \text{ oder } (\text{var}, s).$$

also z.B.

```
x = 4 + y
```

wenn `x` und `y` Variable vom Typ `int` sind. Der Term auf der linken Seite wird ausgewertet und in die Speicherzelle geschrieben, die dieser Variablen zugeordnet ist.

3.2.4 if-Anweisungen

Im folgenden verstehen wir unter *Bedingung* einen Term vom Typ `boolean` oder `(var, boolean)`.

Eine *if*-Anweisung sieht z.B. so aus

```
if(n % 2 == 0){
    n = n+1;
}
else {
    n = n+2;
}
```

Wenn n gerade ist, wird n durch $n + 1$ ersetzt, andernfalls durch $n + 2$.

Allgemein sieht die einfachste if-Anweisung so aus:

```
if (<Bedingung>){
  <Anweisungsfolge>
}
```

Wenn <Bedingung> den Wert `true` hat wird Block ausgeführt.

Man kann auch

```
if (<Bedingung>){
  <Anweisungsfolge 1>
}else{
  <Anweisungsfolge 2>
}
```

verwenden.

Diese Anweisung macht dasselbe wie

```
if (<Bedingung>){
  <Anweisungsfolge 1>
}
if (!<Bedingung>){
  <Anweisungsfolge 2>
}
```

Man kann schließlich auch schreiben

```
if (<Bedingung 1>){
  <Anweisungsfolge 1>
}else if (<Bedingung 2>){
  <Anweisungsfolge 2>
}else if (<Bedingung 3>){
  <Anweisungsfolge 3>
...
}else if (<Bedingung k>){
  <Anweisungsfolge k>
}else{
  <Anweisungsfolge k+1>
}
```

Es wird die erste Bedingung gesucht, die `true` ist. Wenn es eine solche gibt und dies die *i*-te Bedingung ist, wird Block *i* ausgeführt. Wenn es keine solche gibt, wird Block *k*+1 ausgeführt.

Ein Beispiel ist

```
int x;
x = 4;
if(x == 3){
    System.out.println(3);
} else if (x == 4){
    System.out.println(4);
} else {
    System.out.println(0);
}
```

Dieses Programm produziert die Ausgabe 4.

3.2.5 switch-Anweisungen

Eine weitere Auswahlmöglichkeit bietet das `switch`-statement. Hier kommt ein Beispiel:

```
int x;
x = 4;
switch(x) {
case 3:
    System.out.println(3);
case 4:
    System.out.println(4);
case 5:
    System.out.println(5);
default:
    System.out.println(0);
}
```

Hier wird der erste Fall gesucht, der zutrifft. Danach werden alle Anweisungen ausgeführt, die folgen, also alle Anweisungen in allen Fällen. Die Ausgabe ist also

```
4
5
0
```

Allgemein ist die switch Anweisung von der Form

```
switch(<Term vom Typ int>) {
  case <Konstante 1 vom Typ int>
    Anweisungsfolge
  case <Konstante 2 vom Typ int>
    Anweisungsfolge
  ...
  case <Konstante k vom Typ int>
    Anweisungsfolge
  default
    Anweisungsfolge
}
```

Darin wird der erste erfüllte Fall gesucht und dann werden alle Anweisungen ausgeführt, die danach folgen.

3.2.6 while-Anweisung

Erst ein Beispiel

```
x = 1;
while (x < 4) {
  x = x+1;
}
System.out.println(x);
```

Dies gibt 4 aus.

Allgemein sieht die while-Anweisung so aus:

```
while (<Bedingung>){
  <Anweisungsfolge>
}
```

Solange die Bedingung erfüllt ist, wird Block ausgeführt.

Eine Variante ist

```
do{
  <Anweisungsfolge>
}while(<Bedingung>)
```

Block wird ausgeführt, bis die Bedingung erfüllt ist.

3.2.7 for-Anweisung

Erst ein Beispiel

```
int x;
for (x = 1; x <= 10; x = x+1) {
    x = x+2;
}
System.out.println(x);
```

Dieses Programm gibt 11 aus. Es funktioniert genauso wie

```
int x;
x = 1;
while (x <= 10) {
    x = x+2;
    x = x+1;
}
System.out.println(x);
```

Allgemein sieht eine for-Anweisung so aus

```
for( <Initialisierung>; <Bedingung>; <Inkrement> ){
    <Anweisungsfolge>
}
```

Hierin sind Initialisierung und Inkrement Zuweisungen.

Die for-Anweisung macht dasselbe wie

```
<Initialisierung>;
while ( <Bedingung> ) {
    <Anweisungsfolge>
    <Inkrement>;
}
```

3.2.8 break-Anweisung

Ein Beispiel

```
int x;
x = 4;
switch(x) {
case 3:
    System.out.println(3);
    break;
default:
    System.out.println(0);
}
```

Die `break`-Anweisung verläßt die `for`-, `while`-, `do while`- oder `switch`-Anweisung, in der sie vorkommt. Sind solche Schleifen ineinander geschachtelt, so wird in der nächst äußere hinter der verlassenen Schleife weitergemacht. Will man mehrere Schachtelungsebenen verlassen, braucht man labels wie im folgenden Beispiel:

```
outerLoop:
while (i <= 0) {
    while (j <= 0) {
        if (k == 0) break outerLoop;
    }
}
```

3.2.9 continue-Anweisung

Ein Beispiel

```
while (i <= 0) {
    if (k == 0) continue;
    k = k+2;
    i = i+1;
}
```

Wenn $k = 0$ ist, überspringt das Programm die restlichen Anweisungen der `while`-Schleife.

3.3 Arrays

In vielen Programmen braucht man Vektoren, Matrizen oder sogar mehrdimensionale Matrizen. In Java wie in vielen anderen Programmiersprachen benutzt man dafür Arrays.

Das Programm

```
public class Simple {
    public static void main (String args[]) {
        int[] A;
        A = new int[2];
        A[0] = 1;
        A[1] = 2;
        System.out.println(A[0]);
        System.out.println(A[1]);
    }
}
```

1. Deklariert **A** als Variable, dessen Typ ein eindimensionales `int`-Array ist.
2. Erzeugt ein neues eindimensionales `int`-Array der Länge 2 und weist **A** die Adresse dieses Arrays zu.
3. Besetzt die nullte Komponente von **A** mit 1 und den ersten mit 2 (Die Zählung fängt immer mit 0 an).
4. Gibt die beiden Komponenten von **A** aus.

Die Komponenten eines Arrays kann man benutzen wie eine Variable desselben Typs.

Wenn man ein Array einem anderen zuweist wie in

```
int [] A;
int [] B;
A = new int[5];
B = A;
```

dann zeigt die Variable **B** auf dasselbe Array wie **A**. Es gibt also nur ein Array, auf das beide zeigen. Verändert man eine Komponente in **A**, so wird dieselbe Komponente in dem Array **B** verändert.

Entsprechendes gilt für mehrdimensionale Arrays.

3.4 Übungen

32. Exercise Geben Sie an, wie die Ausgabe des folgenden Java-Programms aussieht.

```
public class Simple {
    public static void main (String args[]) {
        int i;
        int k;
        i = -10;
        k = 2;
        while (i <= 0) {
            i = i+1;
            if (k == 0) {
                k = k+2;
                continue;
            }
            k = k-2;
            System.out.println(i);
        }
    }
}
```

33. Exercise Wie kann man ohne Verwendung der `switch`-Anweisung ein Programm schreiben, das dasselbe tut wie folgendes Programm:

```
int x;
x = 4;
switch(x) {
    case 3:
        System.out.println(3);
    case 4:
        System.out.println(4);
    case 5:
        System.out.println(5);
    default:
        System.out.println(0);
}
```

34. Exercise Welche Werte gibt das folgende Programm aus?

```
public class Simple {
    public static void main (String args[]) {
```

```

    int[] A;
    int[] B;
    A = new int[2];
    B = new int[2];
    A[0] = 1;
    A[1] = 2;
    B = A;
    A[0] = 2;
    System.out.println(B[0]);
    System.out.println(B[1]);
}
}

```

35. Exercise Das Sieb des Erathostenes kann alle Primzahlen p die kleiner als eine vorgegebene natürliche Zahl sind, berechnen. Das funktioniert so. Man macht eine Tabelle mit allen Zahlen zwischen 2 und n . Die erste Zahl 2 ist eine Primzahl. Sie wird in eine Primzahlliste geschrieben und gelöscht. Die anderen zahlen der Tabelle werden durch 2 geteilt. Alle Zahlen, bei denen die Division aufgeht, können keine Primzahlen sein. Sie werden gelöscht. Das ist das Sieb mit 2. Die erste übriggebliebene Zahl ist 3, eine Primzahl. Sie wird in die Primzahlliste geschrieben und gelöscht. Die restlichen Zahlen werden durch 3 geteilt. Alle Zahlen, bei denen die Division aufgeht, können keine Primzahlen sein. Sie werden gelöscht. Das ist das Sieb mit 3. Es folgt das Sieb mit 5, 7, 11. Am Schluß hat man in der Primzahlliste alle Primzahlen $\leq n$ stehen und die Tabelle ist leer.

1. Schreiben sie ein Programm, das das Sieb des Erathostenes implementiert und die Liste aller Primzahlen $\leq n$ in ein Array schreibt.
2. Was ist die größte Primzahl, für die man das Sieb wirklich anwenden muß?
3. Schreiben Sie ein Programm, daß in ein zweidimensionales Array alle Primzahlzwillinge $\leq n$ schriebe, also alle Paare von Primzahlen mit Differenz 2.

Kapitel 4

Algorithmen und ihre Effizienz

4.1 Algorithmen

Algorithmen sind Lösungen von Berechnungsproblemen. Wir illustrieren das an einer Reihe von Beispielen.

Problem: Wie ordnet man eine Folge von natürlichen Zahlen der Größe nach? Genauer: Eingabe für den Algorithmus ist eine Folge von k natürlichen Zahlen. Ausgabe ist die aufsteigend geordnete Folge.

Algorithmus: Suche erst die kleinste Zahl und lösche sie, dann die zweitkleinste usw.

Problem: Finde den größten gemeinsamen Teiler zweier natürlicher Zahlen x und y , also die größte natürliche Zahl, die sowohl x als auch y teilt. Eingabe sind also zwei natürliche Zahlen x und y . Ausgabe ist der ggT dieser Zahlen.

Beispiel: Der ggT von 6 und 3 ist 3. Der ggT von 10 und 9 ist 1.

Algorithmus:

1. Vertausche x und y , wenn $x < y$ ist. Danach ist $x \geq y$.
2. Probiere für alle Zahlen $g = y, y - 1, y - 2$ aus, ob sie sowohl x als auch y teilen. Die erste Zahl g , die das tut ist der ggT von x und y .

Um das nächste Problem zu schildern, brauchen wir den begriff des Polynoms. Ein Polynom ist z.B.

$$p(x) = x^2 + 2x - 3.$$

Das ist ein Polynom in der Variablen x . Die Koeffizienten sind 1, 2 und -3 . Dieses Polynom hat die ganzzahlige Nullstelle $x = 1$.

Ein anderes Polynom ist

$$p(x, y, z) = xy + 2xz + yz^2.$$

Das ist ein Polynom in drei Variablen, nämlich x, y, z . Die Koeffizienten sind 1, 2, 1. Dieses Polynom hat die ganzzahlige Nullstelle $(x, y, z) = (1, -1, 1)$.

Man kann auch ein Polynom in 5 Variablen hinschreiben. dann werden die Variablen eben durchnummeriert. Ein Beispiel ist

$$p(x_1, x_2, x_3, x_4, x_5) = x_1 + x_2x_3^2 + x_4x_5^3.$$

Dieses Polynom hat die ganzzahlige Nullstelle $(0, 0, 0, 0, 0)$.

Problem: Eingabe ist ein Polynom $p(x_1, \dots, x_k)$ mit ganzzahligen Koeffizienten. Ausgabe ist `true`, falls das Polynom eine ganzzahlige Nullstelle hat und `false`, wenn es keine hat.

Man kann beweisen, daß es einen solchen Algorithmus nicht gibt. Man kann also kein Programm schreiben, dem man das Polynom eingibt und das dann entscheidet, ob es eine ganzzahlige Nullstelle hat. Für spezielle Polynome geht das aber wohl.

4.2 Effizienz

Eine wichtige Frage ist, wie schnell Algorithmen arbeiten.

Wir analysieren die Algorithmen aus dem letzten Abschnitt.

Zuerst der Sortieralgorithmus. Wir müssen uns erst überlegen, welche Operationen Zeit kosten und welche wir berücksichtigen wollen, weil wir ja nicht so genau wissen, wie der konkrete Computer, auf dem der Algorithmus hinterher abläuft, arbeitet.

Wir entscheiden uns dafür, daß wir nur die Vergleichsoperationen zählen wollen. Die Frage ist also, wieviele Vergleichoperationen das oben vorgeschlagene Verfahren braucht.

Dazu formulieren wir das Verfahren etwas genauer.

```
public class Sort {
    public static void main (String args[]) {
        int[] unsorted = {2,3,5,4,7};
        int[] sorted;
        int i,j,len,index;
        n = unsorted.length;
        sorted = new int[len];
```

```

for (i = 0; i < n; i++) {
    // Suche nach der ersten Komponente ungleich 0
    j = 0;
    while (unsorted[j] == 0)
        j++;
    // Der i-te Eintrag von sorted wird gesetzt.
    sorted[i] = unsorted[j];
    index = j;
    // Der kleinste Wert wird gefunden
    for (j = 0; j < n; j++) {
        if (unsorted[j] > 0 && sorted[i] > unsorted[j]){
            sorted[i] = unsorted[j];
            index = j;
        }
    }
    // der verwendete Wert wird auf 0 gesetzt.
    unsorted[index] = 0;
}
// Das Ergebnis wird ausgegeben
for (j = 0; j < n; j++) {
    System.out.print(sorted[j]);
    System.out.print(' ');
}
System.out.print("\n");
}
}

```

36. Theorem *Der Sortieralgorithmus braucht höchstens $2n^2$ Vergleiche.*

Proof: Wir bestimmen erst eine obere Schranke für die Anzahl der Vergleiche in jedem Durchlauf der `for`-Schleife.

Dort, wo nach dem ersten von Null verschiedenen Wert gesucht wird, braucht man höchstens n Vergleiche, weil man sich im schlechtesten Fall das ganze array ansehen muß. Dasselbe gilt für die Suche nach dem kleinsten Element. Also braucht man insgesamt pro `for`-Schleifendurchlauf höchstens $2n$ Vergleiche. Da die Schleife n -mal durchlaufen wird, braucht man insgesamt höchstens $2n^2$ Vergleiche. ■

Als nächstes analysieren wir den ggT-Algorithmus.

Hier eine präzise Variante.

```

public class GGT {
    public static void main (String args[]) {

```

```

int a;
int b;
int ggT;

// Wir berechnen den ggT von a und b.
// Wir setzen voraus, daß beide Zahlen nicht negativ sind und daß
// a <= b ist.

// Festlegung der Werte

a = 10;
b = 15;

ggT = a;
while (a%ggT != 0 || b% ggT != 0)
    ggT = ggT - 1;
    System.out.println(ggT);
}
}

```

Wir zählen hier die Anzahl der Divisionen mit Rest.

37. Theorem *Der ggT-Algorithmus braucht höchstens $(a - \text{ggT} + 1)^2$ Divisionen mit Rest.*

Wenn a 100 Dezimalstellen hat, braucht man also 10^{100} Divisionen mit Rest. Wenn man pro Sekunde 10^6 Divisionen mit Rest machen kann, braucht man also 10^{97} Sekunden. Ein Jahr hat ungefähr $3 \cdot 10^7$ Sekunden. Man braucht also $3 \cdot 10^{89}$ Jahre. Das ist viel länger als das ganze Erdalter.

Jetzt kommt also die Frage, ob das nicht effizienter geht. Und so ist das auch. Eine effizientere Lösung beruht auf dem folgenden Resultat:

38. Theorem *Sind a, b natürliche Zahlen und ist $b \neq 0$ so ist $\text{gcd}(a, b) = \text{gcd}(b, a \% b)$. Ist $b = 0$ so ist $\text{gcd}(a, b) = a$.*

Proof: Schreibe

$$a = qb + r$$

Dabei ist q der Quotient und r der Rest von a bei der Division durch b . Sei $g = \text{gcd}(a, b)$. Weil $r = a - qb$ ist und g ein Teiler von a und b ist, ist g auch ein Teiler von r . Damit ist r also ein gemeinsamer Teiler von b und r . Es bleibt zu zeigen, daß es keinen größeren gemeinsamen Teiler von b und r gibt. Angenommen, k wäre ein solcher. Dann wäre k auch ein Teiler von $a = qb + r$ und das geht nicht. ■

Daraus ergibt sich der folgende Algorithmus. Er heißt *euklidischer Algorithmus*.

```
public class GGT {
    public static void main (String args[]) {
        int a;
        int b;
        int r;
        int ggT;

        // Wir berechnen den ggT von a und b.
        // Wir setzen voraus, da{\ss} beide Zahlen nicht negativ sind und da{\ss}
        // a <= b ist.

        // Festlegung der Werte

        a = 10;
        b = 15;

        while (b != 0) {
            r = a%b;
            a = b;
            b = r;
        }
        ggT = a;
        System.out.println(ggT);
    }
}
```

39. Theorem *Der euklidische Algorithmus berechnet den ggT von a und b .*

Proof: Nach Theorem 38 Theorem ist der ggT von a und b vor und nach der `while`-Schleife derselbe. Außerdem ist die Folge der b s streng monoton fallend. Es gilt nämlich für den Rest r

$$0 \leq r < b.$$

Daher wird irgendwann mal $b = 0$ erreicht. Dann ist a der ggT von a und b . ■

Hier ist eine kleine Tabelle, die die Berechnung des ggT und die Anzahl der Iterationen zeigt.

a	b	$\text{gcd}(a, b)$	Iterationen
526	614	2	4
5141312	65432123	1	20

Das ist natürlich wesentlich besser als die naive Methode von oben. Wir werden jetzt beweisen, daß das immer so ist. Dazu setzen wir $a = r_0$, $b = r_1$ und

$$r_{i-1} = q_i r_i + r_{i+1}.$$

Hierin ist r_{i+1} der Rest den die Division von r_{i-1} durch r_i läßt.

40. Theorem *Es gilt $r_{i+2} < r_i/2$ für $i \geq 0$.*

Proof: Es ist jedenfalls $r_{i+1} < r_i$ für $i \geq 0$. Wenn $r_{i+1} \leq r_i/2$ ist, dann erst recht $r_{i+2} < r_{i+1} \leq r_i/2$. Wenn aber $r_i > r_{i+1} > r_i/2$ gilt, dann folgt aus

$$r_i = q_{i+1} r_{i+1} + r_{i+2}$$

daß $q_{i+1} = 1$ ist. Damit ist also

$$r_{i+2} = r_i - r_{i+1} < r_i/2.$$

■

Um das nächste Resultat zu formulieren, brauchen wir zwei Bezeichnungen.

Für eine positive reelle Zahl r und eine reelle Zahl $b > 1$ bezeichnet $\log_b r$ den Logarithmus von r zur Basis b . Also ist $\log_b r$ der Exponent mit dem man b potenzieren muß, um r zu erhalten. So ist z.B. $\log_2 2 = 1$, $\log_2 4 = 2$ etc.

Für eine reelle Zahl r ist

$$\lfloor r \rfloor = \max\{x \in \mathbf{Z} : x \leq r\}.$$

Das ist also die größte ganze Zahl, die kleiner oder gleich r ist. So ist z.B.

$$\lfloor 4.5 \rfloor = 4,$$

$$\lfloor -\pi \rfloor = -4,$$

41. Corollary *Die Anzahl der Iterationen im euklidischen Algorithmus ist höchstens $2\lfloor \log_2 b \rfloor + 3$.*

Proof: Nach drei Schritten ist $r_3 < r_1/2$. Nach vier Schritten ist $r_5 < r_1/4$. nach $2i$ Schritten ist $r_{2i+1} < r_1/2^i$. Ist $r_1/2^i < 1$, so ist $r_{2i+1} = 0$, also der Algorithmus fertig. Das ist jedenfalls wahr, wenn $2^i > r_1$ ist oder $i > \log_2 r_1$. Hierzu genügt es, daß $i = \lfloor \log_2 r_1 \rfloor + 1$ ist, wobei $\lfloor \log_2 r_1 \rfloor$ die größte ganze Zahl $\leq \log_2 r_1$ ist. Wenn i so groß ist, dann sind $2i + 1 = 2\lfloor \log_2 r_1 \rfloor + 3$ verbraucht. Mehr Iterationen gibt es nie. ■

Um das vorangegangene Resultat leichter beschreiben zu können, führen wir die O -Notation ein.

42. Definition Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}_{>0}$ Funktionen. Dann schreiben wir $f = O(g)$, wenn es eine positive reelle Zahl K und eine natürliche Zahl n_0 gibt mit der Eigenschaft, daß $f(n) \leq Kg(n)$ für alle $n \geq n_0$.

Es ist z.B. $2\lfloor \log_2 b \rfloor + 3 = O(\log_2 b)$. Das bedeutet: Die Anzahl der Iterationen im euklidischen Algorithmus ist von der Größenordnung $\log_2 b$.

4.3 Übungen

43. Exercise Berechnen Sie

1. $\lfloor \pi^2 \rfloor$.
2. $\log_3(1/27)$.

44. Exercise [Größte Zahl]

1. Entwerfen Sie einen Algorithmus, der den größten Eintrag in einem Array von ints findet.
2. Formulieren Sie diesen Algorithmus in einem Java-Programm.
3. Schätzen Sie die Anzahl der Vergleiche ab, die dieser Algorithmus braucht.

45. Exercise [Erfüllbarkeit] Sei $k \in \mathbb{N}$ und sei x ein array der Länge k vom Typ `boolean`. Sei t ein Term, in dem die Einträge $x[1], \dots, x[k]$ vorkommen, also z.B. $k = 3$ und $t = x[1] \ \&\& \ x[2] \ || \ x[3]$.

1. Entwerfen Sie einen Algorithmus, der entscheidet, ob es Werte für $x[1], \dots, x[k]$ gibt, für die der Wert der Formel `true` ist.
2. Formulieren Sie diesen Algorithmus in Java.
3. Wie oft müssen Sie t höchstens auswerten, um die Antwort zu finden?

46. Exercise [Horner-Schema] Wir wollen einen Ausdruck $\sum_{i=0}^{n-1} a_i$ ausrechnen.

1. Zeigen Sie, daß die direkte Methode $O(n^2)$ Multiplikationen und Additionen erfordert.
2. Verwenden Sie jetzt die Formel $\sum_{i=0}^{n-1} a_i = (\cdots (a_{n-1}x + a_{n-2})x + \cdots + a_1x) + a_0$. Geben Sie ein Auswertungsverfahren an, daß nur $O(n)$ Multiplikationen und Additionen braucht.
3. Implementieren Sie das verbesserte Verfahren.

47. Exercise 1. Beweisen Sie durch vollständige Induktion, für alle $g \in \mathbb{N}$, $g > 1$ gilt $\sum_{i=0}^{n-1} g^i = (g^n - 1)/(g - 1)$.

2. Folgen Sie, daß die Länge der Binärentwicklung einer natürlichen Zahl n genau $\lfloor \log_2 n \rfloor + 1$ ist.

48. Exercise [O -Notation] Entscheiden Sie, ob eine der folgenden Aussagen richtig oder falsch ist. Führen Sie einen entsprechenden Beweis.

1. $n^2 + n + 1 = O(n)$
2. $n^{10} = O(2^n)$.
3. $\log_2(n + 1) = O(\log_2 n)$.
4. $3 + 6 + 9 + \dots + 3n = O(n^2)$.

Kapitel 5

Einführung in die Objektorientierung

Die Technik der Objektorientierung erleichtert die Lösung komplexer Programmieraufgaben (wie z.B. die Erstellung eines Internetbankingprogramms). In diesem Kapitel führen wir in die Objektorientierung ein.

5.1 Einführung

Problem: Wir wollen ein Programm schreiben, das es erlaubt mit der Maus ein Rechteck in ein Fenster zu zeichnen.

Das kann man in Java so machen:

```
import java.awt.Rectangle;
import java.awt.Point;
import java.awt.Graphics;
import java.awt.event.*;

public class SimpleDrawRect extends Applet
implements MouseListener, MouseMotionListener {

    Rectangle dragRect = new Rectangle();
    Point anchorPoint = new Point(0,0);

    public void init() {
        addMouseListener(this);
        addMouseMotionListener(this);
    }
}
```

```

public void mousePressed(MouseEvent event) {
    anchorPoint.x = event.getX();
    anchorPoint.y = event.getY();
}

public void mouseDragged(MouseEvent event) {
    dragRect.setBounds(anchorPoint.x, anchorPoint.y,
        event.getX() - anchorPoint.x,
        event.getY() - anchorPoint.y);

    repaint();
}

public void mouseMoved(MouseEvent event) {}
public void mouseReleased(MouseEvent event) {}
public void mouseEntered(MouseEvent event) {}
public void mouseExited(MouseEvent event) {}
public void mouseClicked(MouseEvent event) {}
// not used by the applet
public void paint(Graphics g) {
    g.drawString("drag mouse to draw a rectangle",30,10);
    g.drawRect( dragRect.x, dragRect.y,
        dragRect.width, dragRect.height);
}

} // end applet

```

Es handelt sich dabei um ein *Applet*, also ein Java-Programm, das in eine sogenannte HTML-Seite eingebunden werden kann und das man dann in einem Internetbrowser wie z.B. Netscape ablaufen lassen kann.

Damit hat man in seinem Browser nicht nur Zugriff zu Daten, die anderwo stehen, sondern auch zu Programmen. Wichtig ist, das Applets nicht alles dürfen, und damit keinen Schaden auf dem eigenen Rechner verursachen können.

Damit man das Applet starten kann, muß man es erst mit javac übersetzen und braucht dann die HTML-Seite

```

<HTML>
<BODY>
<applet code="SimpleDrawRect.class" width=500 height=300>
</applet>
</BODY>
</HTML>

```

im selben Verzeichnis. Die Seite speichert man in einem file, z.B. mit Namen `example.html` und dann führt man das ganze aus mit dem Befehl

```
appletviewer example.html
```

Im folgenden wird beschrieben, wie das funktioniert.

5.2 Klassen und Objekte

5.2.1 Importieren von Packages und Klassen

Bis jetzt können wir in Programmen nur einfache Datentypen verwenden, z.B. `int`, `double`, `boolean` und `arrays` über diesen Typen. Wenn wir also mit Rechtecken arbeiten wollen, müssen wir die mit den elementaren Datentypen konstruieren. Zum Glück gehört aber zum Java Development Kit das *Package* `awt` (Abstract Window Tools) und darin gibt es die Klasse `Rectangle`.

Woher weiß man das?

Aus der Dokumentation von jdk. Wenn man sich die html-Seite

```
/jdk1.1.7/docs/api/packages.html
```

ansieht, erhält man eine Übersicht über die sogenannten *Packages*, die jdk mitliefert.

Will man in einem eigenen Programm dieses Package benutzen, schreibt man:

```
import java.awt.*;
```

Das steht auch in unserem Applet. Damit werden alle Klassen aus dem Package `awt` in unser Programm importiert. Es werden auch noch einige andere Packages importiert.

Wie kann man das Package benutzen?

Man klickt das package `awt` an und erhält eine Seite mit der Überschrift

```
package java.awt
```

Dort sucht man den *class index*. Darin findet man die Klasse `Rectangle`.

Die Klasse klickt man an und kommt auf eine Seite mit der Überschrift

Class `java.awt.Rectangle`

Dort ist beschrieben, wie man die Klasse `Rectangle` benutzen kann. Die Implementierung der Klasse *Rectangle* ist da aber nicht beschrieben.

5.2.2 Die Klasse `Rectangle`

Die Klasse `Rectangle` erlaubt es, in eigenen Programmen mit Objekten vom Typ `Rectangle` zu arbeiten, sofern man die Klasse importiert hat. Solche Objekte besitzen Eigenschaften und Fähigkeiten, die man von Rechtecken erwartet.

Wenn man `Rectangles` verwenden will, muß man sich ansehen, wie Objekte vom Typ `Rectangle` benutzt werden können. Das findet man in der Dokumentation der Klasse.

HTML-Doku zeigen

Grundsätzlich gilt:

Rechtecke leben in Koordinatensystemen. Die Maßeinheit ist *Pixel*. Der Punkt (x, y) ist der linke untere Eckpunkt. Das Rechteck ist achsenparallel. Höhe und Breite haben die offensichtliche Bedeutung.

Bild.

5.2.3 Deklaration

Will man in seinem Programm ein `Rectangle` benutzen, deklariert man eine Variable vom Typ `Rectangle`. Das sieht so aus

```
import java.awt.Rectangle;
```

```
public class Test { public static void main (String args[]) { Rectangle myRect;
```

Wenn man eine öffentliche Klasse importiert, vergrößert sich die Menge der Typen, auch der Variablentypen.

Durch diese Deklaration gibt es die Variable `myRect` vom Typ `Rectangle` aber noch kein `Rectangle`-Objekt. Das ist so ähnlich wie bei den Arrays.

5.2.4 Konstruktion

Will man ein Objekt vom Typ `Rectangle` konstruieren, braucht man dazu eine Konstruktionsanweisung.

Die sieht z.B. so aus:

```
myRect = new Rectangle()
```

Danach zeigt die Variable `myRect` auf ein neues Objekt vom Typ `Rectangle`.

Bild

Auf diese Weise kann man immer ein neues Objekt konstruieren. Man schreibt

```
<Variable vom Typ Klasse> = new <Klassenname>()
```

Das ist der sogenannte *noargs*-Konstruktor. Es kann in Klassen auch noch andere Konstruktoren geben. Das sehen wir unten.

5.2.5 Attribute

Die Variablen oder Attribute von `Rectangles` sind

1. `height`, die Höhe des Rechtecks
2. `width`, die Breite
3. `x`, die *x*-Koordinate
4. `y`, die *y*-Koordinate.

Will man wissen, wie die Datenfelder dieses neuen Objektes gefüllt sind, kann man das mit dem Programm

```
import java.awt.Rectangle;

public class Test {
    public static void main (String args[]) {
        Rectangle myRect;
        myRect = new Rectangle();
    }
}
```

```

    System.out.println(myRect.x);
    System.out.println(myRect.y);
    System.out.println(myRect.height);
    System.out.println(myRect.width);
}
}

```

machen. Die Ausgabe ist

```

0
0
0
0

```

Es wurde also ein Rechteck konstruiert, dessen linke obere Ecke die Koordinaten (0, 0) hat und dessen Höhe und Breite 0 ist.

Auf die öffentlichen Variable `width` kann man also mit

myRect.width

Dies ist dann eine Variable vom Typ `int` und kann auch genauso benutzt werden.

Allgemein kann man auf eine öffentliche Variable

`<Variable vom Typ s>.<Bezeichner der {"o"}ffentlichen Variablen>`

zugreifen

5.2.6 Weitere Konstruktoren

In der Klasse `Rectangle` gibt es aber noch andere Konstruktoren, wie man aus der Übersicht über die Konstruktoren sieht.

Es gibt z.B.

```

Rectangle(int, int)
    Constructs a new rectangle whose top-left corner is at (0, 0) in
    the coordinate space, and whose width and height are specified
    by the arguments of the same name.

```

5.2.7 Methoden

Die nächste Frage ist, was man mit Objekten vom Typ `Rectangle` machen kann. Man kann auf sie die Methoden der Klasse anwenden.

Es gibt z.B.

`contains`

```
public boolean contains(int x,  
                        int y)
```

Checks whether this rectangle contains the point at the specified location (x, y).

Parameters:

x - the x coordinate.

y - the y coordinate.

Returns:

true if the point (x, y) is inside this rectangle; false otherwise.

Das Programm

```
import java.awt.Rectangle;  
  
public class Test {  
    public static void main (String args[]) {  
        Rectangle myRect;  
        myRect = new Rectangle(1,1,10,10);  
        System.out.println(myRect.contains(0,0));  
    }  
}
```

hat die Ausgabe `false`.

Das Programm

```
import java.awt.Rectangle;  
  
public class Test {  
    public static void main (String args[]) {  
        Rectangle myRect;
```

```
        myRect = new Rectangle(1,1,10,10);
        System.out.println(myRect.contains(2,2));
    }
}
```

hat die Ausgabe `false`.

5.2.8 Implementierung

Hier ist die Implementierung der Klasse:

```
public class Rectangle {

    // data fields

    public int x;
    public int y;
    public int width;
    public int height;

    // constructors

    public Rectangle() {
        x = y = width = height = 0;
    }

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    // methods

    public void setBounds(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    public void translate(int dx, int dy) {
```

```
        x += dx;  
        y += dy;  
    }  
}
```

5.2.9 Implementierung der Variablen

Zuerst kommen die Datenfelder oder Variablen. Sie können in folgenden Zugriffs-klassen sein

public: Die Zugriffsklasse erlaubt weltweiten Zugriff.

protected, private `protected` wird noch erklärt

private: Die Zugriffsklasse erlaubt nur Zugriff aus der Klasse.

Man schreibt also z.B.

```
// data fields  
public int x;  
public int y;  
public int width;  
public int height;
```

Man schreibt also eine Folge von Deklarationen mit der entsprechenden Zugriffs-klasse. Dabei können natürlich auch Deklarationen von Variablen von neuen Klassen-typen stehen.

5.2.10 Implementierung von Konstruktoren

Es kann in einer Klasse mehrere Konstruktoren geben. Es gibt jedenfalls immer, auch ohne explizite Implementierung, den noargs-Konstruktor.

Konstruktoren haben immer den gleichen Namen wie die Klasse. Zwei verschiedene Konstruktoren müssen immer eine verschiedene Signatur haben.

In den Konstruktoren werden den Variablen Werte zugewiesen. Es können aber auch irgendwelche anderen Sachen gemacht werden. Im Gegensatz zu Methoden können Konstruktoren aber keinen Rückgabewert haben.

In einem Konstruktor kann man **this** verwenden um auf Variablen des zu konstruierenden Objekts zuzugreifen, wenn diese durch Parameter des Konstruktors verdeckt sind, wenn es also Parameter gibt, die genauso heißen wie die Variablen

Beispiel:

```
public Rectangle(int x, int y, int width, int height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
}
```

Es gibt dieselben Zugriffsklassen wie bei den Variablen.

5.2.11 Implementierung von Methoden

Methoden haben eine Parameterliste und einen Rückgabetyt.

Beispiele für Methoden der Klasse `Rectangle`

```
public int getHeight() {
    return height;
}
```

Diese Methode liefert die Höhe des Rechtecks. Die Anweisung

```
    return height;
```

bestimmt, was der Rückgabewert ist.

Dagegen hat folgende Methode den Rückgabetyt `void`, sie liefert also keine Rückgabe. In einer solchen Methode kann `return` nicht verwendet werden. Es gibt wieder dieselben Zugriffsklassen wie bei den Variablen.

```
public void setBounds(int x, int y, int width, int height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
}
```

In dieser Methode ist der Rückgabetyt `void`, es wird also nichts zurückgegeben.

Es darf verschiedene Methoden desselben Namens geben, sofern sie eine unterschiedliche Signatur haben.

5.2.12 Datenabstraktion

Wir haben bis jetzt ein wichtiges Prinzip der Objektorientierung kennengelernt, die Datenabstraktion.

Klassen stellen neue abstrakte Datentypen zur Verfügung. Objekte eines bestimmten Typs sind Instanzen der entsprechenden Klasse.

Den Benutzer interessiert in erster Linie

1. wie man das Objekt konstruiert,
2. was man das Objekt fragen kann,
3. wie man das Objekt ändern kann,
4. was das Objekt machen kann.

In Java bedeutet das, daß man die Konstruktoren, Datenfelder und Methoden der Klasse kennen muß.

5.3 Die Mouse-Events

Woher weiß das Applet, was die Mouse macht?

Es verwendet die *Interfaces* `MouseListener` und `MouseMotionListener`.

Ein Interface ist eine Sammlung von Methodennamen (abstrakten Methoden), die aber noch keine Implementierung haben. Im Interface `MouseListener` gibt es den Methodennamen `mousePressed(MouseEvent event)`. Wenn das Interface aktiviert ist und die Mouse gedrückt wird, wird ein `MouseEvent` ausgelöst und mit diesem `MouseEvent` wird die Methode `MousePressed` aufgerufen. Da die Methode aber abstrkt ist, also keine Implementierung hat, muß sie der Benutzer selbst implementieren.

in unserem Beispiel sieht die Implementierung so aus:

```
public void mousePressed(MouseEvent event) {
    anchorPoint.x = event.getX();
    anchorPoint.y = event.getY();
}
```

Die x-Variable des Punktes `anchorPoint` wird auf die x-Koordinate der Mouse gesetzt. Für die y-Koordinate passiert das entsprechende.

Die andere Implementierung ist

```
public void mouseDragged(MouseEvent event) {
    dragRect.setBounds(anchorPoint.x, anchorPoint.y,
                       event.getX() - anchorPoint.x,
                       event.getY() - anchorPoint.y);
    repaint();
}
```

Diese Methode wird aufgerufen, wenn die Mouse gezogen wird. Darin wird der linke obere Eckpunkt des Rechtecks auf `anchorPoint` gesetzt und die Länge und Breite des Rechtecks auf die Differenz zwischen der x bzw. y-Koordinate des `anchorPoint` und der entsprechenden Koordinate der Mouse. Man beachte, daß sich dabei auch negative Breite und Höhe ergeben können.

Alle anderen abstrakten Methoden des Interfaces müssen auch implementiert werden. Da sie aber nichts machen sollen, gibt es nur eine leere Implementierung.

5.4 Wir funktioniert das Applet?

Unsere Klasse verwendet das package `applet`. Es enthält Klassen, die es erlauben, Applets zu schreiben.

Zeige das package `applet`

Da unsere Klasse ein Applet werden soll, soll es sich so verhalten, wie ein Applet. Daher wird am Anfang geschrieben

```
public class SimpleDrawRect extends Applet
```

Sie bewirkt, daß `SimpleDrawRect` eine Unterklasse von `Applet` ist, die `Applet` erweitert. In der Klasse ist nun alles verfügbar, was auch in `Applet` verfügbar ist, es sei denn, es ist in der Zugriffsklasse `private`. Es sind also für Objekte vom Typ `SimpleDrwaRect` alle anderen Variablen, Konstruktoren und Methoden der Klasse `Applet` vorhanden.

Die Anweisung hat außerdem die Wirkung, daß beim Laden des Applets in einen Browser oder in den `appletviewer` folgendes passiert:

Die Methode `init()` wird zuerst aufgerufen. Sie teilt dem Applet mit, das es in den Browser geladen wurde und damit initialisiert wurde.

Danach wird die Methode `start()` ausgeführt. Wird der Browser zum icon verkleinert, wird `stop()` ausgeführt. Wird das Applet zerstört, wird zuerst `destroy()` ausgeführt.

Unser Applet braucht nur die Methode `init()`. Ihre Doku sieht so aus:

`init`

```
public void init()
```

Called by the browser or applet viewer to inform this applet that it has been loaded into the system. It is always called before the first time that the start method is called.

A subclass of Applet should override this method if it has initialization to perform. For example, an applet with threads would use the init method to create the threads and the destroy method to kill them.

The implementation of this method provided by the Applet class does nothing.

Man sieht daran, daß diese Methode das Applet informiert, daß es geladen wurde, dann aber nichts macht. Was es macht, muß der Benutzer selbst sagen. Wir sagen, in unserer Implementierung, daß das Applet die Mouse und ihre Bewegungen beachten soll.

```
public void init() {  
    addMouseListener(this);  
    addMouseMotionListener(this);  
}
```

Dadurch wird die Methode `init()` aus der Klasse Applet *überladen*. In diesem Applet wird also diese `init()`-Methode angewendet und nicht die originale.

Die andere Methode, die überschrieben wurde ist die `paint()`-Methode aus der Klasse Component die auch in dem `awt`-Package steht. Sie wird verwendet in der Methode `repaint()`.

5.5 Nach oben links Zeichnen

Wir wollen uns das Prinzip der Vererbung noch etwas genauer ansehen. In unserem Applet können wir Rechtecke nur nach unten rechts, nicht aber nach oben links zeichnen. Das liegt daran, daß beim Zeichnen nach oben links die Höhe oder Breite oder beide negativ werden, weil die Breite als Differenz zwischen der x-Koordinate der Mouse und der x-Koordinate des Aufhängungspunktes bestimmt werden. Für die Höhe gilt analoges. Ist Höhe oder Breite negativ, so ist das Rechteck nach Definition leer ist. Das bedeutet: nach dem Zeichnen sieht man nichts. Wie kann man sich da behelfen?

Man schreibt seine eigene Rechteckklasse, die eine Unterklasse der `Rectangle`-Klasse ist. In dieser Klasse gibt es eine Methode `normalize`. In der werden Rechtecke, die eine negative Breite oder Höhe haben, in Rechtecke verwandelt, die genauso aussehen, aber eine positive Breite und Höhe haben. Das erreicht man dadurch, daß man den Eckpunkt nach oben links verlegt.

Ist die Höhe negativ und die Breite positiv, so liegt der Eckpunkt unten links. Man setzt dann

```
y = y + height;
height = -height;
```

insgesamt sieht der Code der Klasse so aus:

```
class DragRect extends Rectangle{
    void normalize() {
        if (width <0) {
            x += width;
            width = - width;
        }
        if (height <0) {
            y += height;
            height = - height;
        }
    }
    void rotate() {
        this.setBounds(x, y, -height, width);
        this.normalize();
    }
}
```

Welche Veränderungen muß man jetzt in dem Programm machen?

Zeige das gesamte Programm auf Folie.

Man deklariert `dragRect` als `DragRect` und vor dem `repaint()` ruft normalisiert man `dragRect` mit `normalize()`.

Zeige den Effekt.

5.5.1 Vererbung

Wir fassen zusammen, wie man eine Klasse durch Vererbung erweitern kann.

Man deklariert eine Klasse `MyClass` als Unterklasse einer Klasse `Class` durch die Anweisung.

```
MyClass extends Class
```

Sobald man das getan hat, gilt folgendes:

Alle Konstruktoren, Methoden oder Variablen der Zugriffsklasse `public`, `protected` oder `private` `protected` stehen für Objekte der abgeleiteten Klasse zur Verfügung.

`MyClass` kann neue Variablen, Konstruktoren und Methoden bereitstellen.

`MyClass` kann Konstruktoren und Methoden von `Class` überschreiben. Auf Objekte vom Typ `MyClass` wird dann immer die Methode aus der zugehörigen Klasse angewendet.

Man kann Variablen vom Typ `Class` Objekte vom Typ `MyClass` zuweisen. Dann kann man aber nur noch die Methoden anwenden, die es auch in der Oberklasse gibt. Wurde eine solche Methode in `MyClass` überschrieben, so wird die überschreibende Methode angewendet.

5.6 Ein komplexeres Beispiel

Im folgenden Beispiel werden wir sehen, wie man drei verschiedene Formen als Rahmen oder ausgefüllt in drei verschiedenen Farben in einem Applet zeichnet. Dabei werden wir die Objektorientierung noch ein bißchen genauer kennenlernen.

5.6.1 Zeichnen mit der Graphics-Klasse

Zeichnen kann man mit Hilfe der `Graphics`-Klasse aus dem `awt`-Package. Diese stellt Objekte zur Verfügung, die den gegenwärtigen Zeichenkontext repräsentieren. Solche Objekte nennen wir Graphik-Kontext. Ein solches Objekt hat unter anderem eine Vordergrund- und Hintergrundfarbe und stellt verschiedene Zeichenmethoden zur Verfügung.

Wir erläutern diese Zeichenmethoden. Wir wollen Rechtecke, abgerundete Rechtecke und Ovale zeichnen. Die `boundsBox` einer solchen Form ist das kleinste Rechteck, das sie enthält.

Bild.

Hat man eine solche `boundsBox` und ein `Graphics`-Objekt `g`, so wird der Methodenaufruf

```
g.fillOval(boundsBox.x, boundsBox.y, boundsBox.width, boundsBox.height)
```

das größte Oval zeichnen, das in der `boundsBox` liegt und zwar mit der Vordergrundfarbe von `g` als Ausfüllfarbe.

Entsprechend wird der Methodenaufruf

```
g.drawRoundRect(boundsBox.x, boundsBox.y, boundsBox.width, boundsBox.height)
```

das größte abgerundetes Rechteck zeichnen, das in die `boundsBox` paßt. Abgerundet heißt, daß die ecken Viertelkreise mit Durchmesser 20 Pixel sind.

5.6.2 Die Klassenhierarchie Shapes

Wir entwerfen jetzt eine Klassenhierarchie, die die Formen widerspiegelt, die wir zeichnen wollen.

Ganz oben steht die abstrakte Klasse `Shape`. Daß sie abstrakt ist, hat zur Folge, daß keine Instanzen haben kann. Es wird also keine Objekte vom Typ `Shape` geben. Sie vereinheitlicht nur die verschiedenen Formen. Sie sieht so aus:

```
abstract class Shape {
    protected Color color;
    protected DragRect boundsBox;           // the boundsBox is the smallest
    // rectangle enclosing the shape
```

```
    abstract void draw(Graphics g);    // abstract placeholder method that
    // gets instantiated in subclasses
}
```

Jede Unterklasse von `Shape` hat also eine Farbe `color` vom Typ `Color` und eine `boundsBox` vom Typ `DragRect`. Diese Variablen sind `protected`, also nur aus demselben `package` zugänglich.

49. Exercise Ermitteln Sie, wie man ein Java-Package anlegt. (Ausführlicher)

Außerdem gibt es die abstrakte Methode `draw(Graphics g)`, die in einem `Graphics`-Kontext die jeweilige Form zeichnet. Die Methode ist abstrakt, hat also keine Implementierung. Die Implementierung muß für jede Form individuell gemacht werden, weil jede Form anders gezeichnet wird.

Von dieser Klasse leiten wir die Klassen `HollowRectangle`, `HollowOval` und `HollowRoundRect` ab. Von diesen Klassen leiten wir wiederum die Klassen `FilledRectangle`, `FilledOval` und `FilledRoundRect` ab.

Bild

5.6.3 Polymorphie

In jeder dieser Klassen ist die `draw`-Methode implementiert. Mit ihr können Objekte vom jeweiligen Typ gezeichnet werden. Das entscheidende bei der Benutzung der abstrakten Oberklasse ist aber folgendes. Angenommen, man hat eine Variable vom Typ `Shape` deklariert durch die Anweisung

```
Shape myShape;
```

Dann kann man dieser Variablen Objekte vom Typ jeder Unterklasse zuweisen. Man kann z.B. schreiben

```
myShape = new FilledOval();
```

oder

```
myShape = new HollowRectangle();
```

Wenn man jetzt mit

```
myShape.draw();
```

dem Objekt mit Namen `myShape` die Nachricht schickt, daß es sich zeichnen soll, dann wird genau die `draw`-Methode angewendet, die zu dem jeweiligen Objekt gehört. Die Objekte kennen also ihre eigenen `draw`-Methoden, auch wenn sie durch eine Variable einer Oberklasse bezeichnet sind. Man nennt dies **Polymorphie**. Die Nachricht `draw` kann also viele Bedeutungen haben je nachdem, an wen sie gesendet wird.

5.6.4 Die Unterklassen

Von der Klasse `Shape` wird die Unterklasse `HollowOval` abgeleitet. Diese Klasse sieht so aus:

```
class HollowOval extends Shape {
    void draw(Graphics g) {
        g.setColor(Color.black);
        g.drawOval(  boundsBox.x, boundsBox.y,
                    boundsBox.width, boundsBox.height);
    }
}
```

In dieser Klasse ist nur die `draw`-Methode implementiert. Im Graphik-Kontext wird zuerst die Farbe gesetzt. Dies geschieht mit der sogenannten *Klassenkonstanten* `Color.black`. Diese ist in der Klasse `Color` deklariert als

```
public static final Color black
```

Konstante, die `static` ist, also zur Klasse gehört und nicht zu den einzelnen Objekten und als `final`, also also Konstante, die nicht mehr geändert werden kann.

Von `HollowOval` wird `FilledOval` abgeleitet:

```
class FilledOval extends HollowOval {
    void draw(Graphics g) {
        g.setColor(color);
        g.fillOval(boundsBox.x, boundsBox.y,
                 boundsBox.width, boundsBox.height);
        super.draw(g);
    }
}
```

Hier wird ebenfalls die `draw`-Methode überschrieben. Die Anweisung

```
g.setColor(color);
```

setzt die Vordergrundfarbe auf die Farbe, die in `Shape` gespeichert ist. Dann wird das ausgefüllte Oval gezeichnet. Danach soll noch ein schwarzer Rahmen um das ausgefüllte Oval gemahlt werden. Dazu wird die `draw`-Methode aus der Oberklasse mit

```
super.draw(g);
```

aufgerufen. Hier wird die `draw`-Methode der Oberklasse auf das `FilledOval`-Objekt angewendet.

50. Exercise Iteration von `super`.

5.6.5 Variablen des DrawShapes-Applet

`Vector shapeList` gehört zu dem package `java.util`. Es stellt ein Array mit dynamischer Länge zur Verfügung. In diesem Vektor sind hinterher die ganzen Shapes gespeichert, die schon gezeichnet wurden.

`Choice` ist eine Klasse aus dem package `awt`. Es erlaubt die einfache Implementierung der Popup-Menüs. Hier braucht man drei Stück, nämlich `shapeChoice`, die festlegt, welche Form gerade gezeichnet werden soll, `colorChoice`, die festlegt, welches die aktuelle Farbe ist und `fillChoice`, die festlegt, ob gerade nur der Rahmen oder eine ausgefüllte Form gezeichnet wird.

`anchorPoint` speichert die Koordinaten der Mouse, wenn sie gedrückt wird.

Jetzt kommen einige Variablen für das sogenannte `offScreenImage`. Man verwendet diese Technik, um zuerst das Bild zu konstruieren, um es dann als ganzes zu zeichnen. Das vermeidet Flackern.

`offScreenDimension` ist vom Typ `Dimension` aus `awt` und speichert Breite und Höhe des Bildes.

`offScreenImage` ist vom Typ `Image`. Diese abstrakte Klasse ist die Oberklasse für alle Klassen, die beschreiben, wie auf der jeweiligen Plattform Bilder gezeichnet werden.

`offScreenGraphics` ist vom Typ `Graphics`. Diese abstrakte Klasse ist die Oberklasse für alle Klassen, die beschreiben wie der Darstellungskontext ist.

5.6.6 Implementierung von `init`

Als erste Methode wird `init()` überschrieben. Diese Methode haben wir ja schon kennengelernt. Sie wird aufgerufen, wenn das Applet geladen wird. Es werden die Popup-Menüs erzeugt und die Mousebeobachter angeworfen.

5.6.7 Implementierung von `MousePressed`

Es wird der augenblickliche Inhalt der Popup-Menüs abgefragt. Die Farbe des augenblicklichen `Shapes` wird gesetzt. Eine neue `boundsBox` wird erzeugt (das ist wichtig, weil jedes neue Shape seine eigene `boundsbox` braucht. Das neue Shape wird in die Liste eingefügt. Der Startpunkt der Mouse wird gespeichert.

5.6.8 Implementierung von `MouseDragged`

Es wird ein temporärer Name für die `boundsBox` eingeführt, nämlich `dragRect`. Die Koordinaten der `boundsBox` werden gesetzt und normalisiert. Das Bild wird komplett neu gezeichnet.

5.6.9 Implementierung von `MouseReleased`

Ist eigentlich überflüssig und nur zu Demozwecken vorhanden.

5.6.10 Implementierung von `paint()`

Es wird `update` aufgerufen.

5.6.11 Implementierung von `update()`

Diese Methode zeichnet alle Shapes in der Liste im Hintergrund und bringt das ganze Bild am Schluß auf den Bildschirm.

Zuerst wird mit

```
Dimension d = getSize();
```

die Augenblickliche Bildgröße des Applets in `d` gespeichert.

51. Exercise Woher kommt die Methode `getSize`.

Dann wird eine neue Zeichenumgebung kreiert, wenn es noch keine gab, oder der Benutzer die Appletgröße verändert hat. Die Methode `createImage` kommt aus der Klasse `awt.Component`, von `applet.Applet` abgeleitet ist.

Jetzt wird das aktuelle Bild gelöscht. Danach wird die Umrandung des Bildes erzeugt.

Danach werden die Shapes gezeichnet. Die explizite Umwandlung in den Typ `Shape` ist nötig, weil der Typ ansonsten `java.lang.Object` ist.

Schließlich wird das Bild komplett gezeichnet.

Kapitel 6

Verkettung

6.1 Motivation

Am Beispiel der immer größeren Liste von Formen, die in dem `DrawShapes`-Applet gezeichnet wurde, haben wir bereits gesehen, wie wichtig Listen sind, deren Länge dynamisch ist.

Ein anderes Beispiel sind beliebig lange ganze Zahlen. Man kann sie als Liste von `ints` auffassen, wobei dann jedes `int` eine Ziffer in der Darstellung zur Basis 32 ist.

In diesem Kapitel geht es um eine elementare und wichtige Methode, solche Listen mit variabler Länge zu speichern.

6.2 Verkettete Listen

Die Grundidee ist die Verwendung von *verketteten Listen*. Es handelt sich dabei um eine Ansammlung von Speichereinheiten. Es gibt eine erste Speichereinheit. Von jeder Speichereinheit gibt es einen Verweis auf die nächste Speichereinheit. Und es gibt eine letzte Speichereinheit.

Bild.

Man kann solche verketteten Listen erweitern, indem man die Kette an einer Stelle aufbricht und ein neues Glied einhängt oder eins vorn oder hinten anhängt. Man kann die Listen auch verkürzen. Will man ein Glied lokalisieren, so muß man die ganze Liste durchgehen.

6.3 Pointer

Um verkettete Listen zu realisieren, braucht man *Pointer*. Ein Pointer ist ein Datum, daß eine Speichereinheit referenziert; er zeigt auf die Speichereinheit.

Wir haben Pointer schon bei der Java-Programmierung kennengelernt. Ist `Class` eine Klasse und ist `object` eine Variable vom Typ `Class`, dann ist der Wert von `object` ein Pointer. Dieser Pointer zeigt entweder auf ein Objekt vom Typ `Class` oder auf nichts. Im letzteren Fall ist der Pointer `null`.

Das folgende Programm liefert die Ausgabe `null`:

```
public class Simple {
    public static void main (String args[]) {
        Object object = null;
        System.out.println(object);
    }
}
```

Hat man eine Variable `myRect` vom Typ `Rectangle` deklariert und initialisiert, so kann man mit

```
myRect.width
```

die Breite von `myRect` erhalten. Dadurch wird der Pointer *dereferenziert*. Die Referenz wird durch einen Wert, der hinter der Referenz steht, ersetzt.

Dagegen wird in

```
myRect = anotherRect
```

der Variablen `myRect` der Wert der Variablen `anotherRect` zugewiesen. Der Wert von `anotherRect` ist ein Pointer.

Wenn man eine identische Kopie eines Objektes erzeugen will, braucht man die Methode `clone()` aus der Klasse `lang.Object`.

Was passiert mit einem Objekt, auf das kein Pointer mehr zeigt? Wenn Java keinen Platz mehr hat, startet Java einen Garbage-Collector und benutzt den Platz. Das ist bei anderen Programmiersprachen nicht so. Da muß man selbst fürs Aufräumen sorgen.

52. Exercise Welchen Wert hat eine Variable, die ins Nichts zeigt? Wie legt man einen zweiten Pointer auf ein Objekt? Wie legt man eine identische Kopie eines Objektes an?

6.4 Lineare verkettete Listen

Wir beschreiben jetzt etwas präziser lineare verkettete Listen.

Eine solche *lineare verkettete Liste* ist eine Folge von Knoten, die folgende Eigenschaften haben:

1. jeder Knoten hat eine Adresse
2. jeder Knoten trägt eine Information,
3. jeder Knoten außer dem letzten kennt die Adresse seines Nachfolgers,
4. der letzte Knoten hat die Nachfolgeradresse `null`, er weiß also, daß er der letzte ist.

Wenn man das objektorientiert implementieren will, braucht man jedenfalls die Klassen

1. `LinkedList`
2. `ListNode`

Wir wollen in der Klasse `LinkedList` folgende Methoden unterstützen:

1. `insertNewSecodNode(Object info)`: einfügen eines neuen zweiten Knotens mit dem Inhalt `info`
2. `listSearch(Object info)`: wenn `info` als Inhalt in der Liste vorkommt, soll der entsprechende Knoten zurückgegeben werden
3. `deleteLastNode()`: der letzte Knoten soll gelöscht werden
4. `insertNewLastNode()`: der letzte Knoten soll gelöscht werden
5. `print()` die Liste soll ausgedruckt werden

Die Klasse `ListNode` implementieren wir so:

```
public class ListNode {
    public ListNode link;
    public Object info;
}
```

Es gibt eine Variable `link`, die vom Typ `ListNode` ist und auf den nächsten `ListNode` zeigt. Es gibt auch eine Variable `info`, die den Inhalt des Knotens enthält und vom Typ `Object` ist. Diese Variable kann also auf ein beliebiges Objekt zeigen.

Die Klasse `LinkedList` hat die Variablen

```
private int length;
private ListNode firstNode;
```

Die Variable `length` zeigt auf die Länge der Liste. Die Variable `firstNode` zeigt auf den ersten Knoten der Liste. Wenn die Liste leer ist, ist dieser Zeiger `null`.

Wir diskutieren jetzt die Methode `insertSecondNode(Object info)`, die einen neuen Knoten mit dem Inhalt `info` einfügt.

```
public void insertNewSecondNode(Object info) {
    ListNode newNode;
    newNode = new ListNode();
    newNode.info = info;
    newNode.link = firstNode.link;
    firstNode.link = newNode;
}
```

Diese Methode ist extrem einfach. Zuerst wird ein neuer Listenknoten erzeugt. Dessen `info`-Variable wird auf `info` gesetzt. Dessen `link`-Variable zeigt auf denselben Knoten wie die `link`-Variable von `firstNode`. Danach wird der Zeiger `firstNode` auf `newNode` umgelegt.

Diese Methode braucht Zeit $O(1)$. Ein neuer zweiter Knoten kann also sehr schnell eingefügt werden. Wenn die Liste leer war, fügt diese Methode allerdings einen neuen ersten Knoten ein.

Die Methode `listSearch(Object info)` sucht in der Liste nach einem Knoten, dessen `info`-Feld auf ein Objekt zeigt, das mit dem Objekt übereinstimmt, auf das `info` zeigt. Die Frage ist also, ob die Information, auf die `info` zeigt, in der Liste vorkommt. Wenn ein passender Knoten gefunden wird, wird der zurückgegeben. Andernfalls wird `null` zurückgegeben.

Die Idee für das Verfahren ist folgende.

Der Algorithmus geht alle Listenknoten durch und prüft, ob die Information des Knotens mit der vorgegebenen Information übereinstimmt. Dazu verwendet er eine Variable N vom Typ `ListNode`. Die wir zuerst auf `firstNode` gesetzt. Wenn

`firstNode = null` ist, dann ist die Liste leer und es wird `null` zurückgegeben. Andernfalls wird geprüft, ob die Information des Knotens mit der vorgegebenen Information übereinstimmt. Wenn ja, dann wird der Knoten zurückgegeben. Wenn nicht, dann wird N auf den Nachfolgerknoten gesetzt und das Verfahren wird wiederholt.

Bild

Tatsächlich kann das Verfahren exakt wiederholt werden, weil der Nachfolgerknoten des letzten Knotens `null` ist. Hier ist das Programm.

```
public ListNode listSearch(Object info) {
    ListNode N;
    N = firstNode;
    while(N != null) {
        if(info.equals(N.info)) {
            return N;
        } else {
            N = N.link;
        }
    }
    return N;
}
```

Wichtig ist in dem Programm, daß bei der Gleichheitsüberprüfung die Methode `equals` aus der Klasse `Object` benutzt wird. Sie stellt fest, ob die entsprechenden Objekte identisch sind und nicht ob die beiden Zeiger auf dasselbe Objekt zeigen. Letzteres würde mit `==` überprüft.

Die Methode `listSearch` benötigt Zeit $O(n)$. Im schlechtesten Fall, wenn z.B. die Information nicht vorhanden ist, müssen nämlich alle Knoten durchgemustert werden. Das ist sehr langsam. Es gibt Datenstrukturen, die eine effizientere Suche unterstützen.

Die Methode `deleteLastNode` löscht den letzten Knoten.

Um sie zu implementieren, unterscheiden wir drei Fälle.

1. Die Liste ist leer. Dann macht die Methode nichts.
2. Die Liste hat die Länge 1, hat also nur einen Knoten. Dann ist die Liste anschließend leer.
3. Die Liste hat eine Länge, die größer ist als 1. Dann wird der vorletzte Knoten gesucht und sein `link` auf `null` gesetzt.

Wie macht man das konkreter?

Ob die Liste leer ist, erkennt man daran, das `firstNode == 0` wahr ist. Man braucht also nur etwas zu machen, wenn `firstNode != 0` wahr ist.

Ob die Liste die Länge 1 hat, erkennt man daran, daß `firstNode.link == 0` wahr ist. In dem Fall muß man *firstNode* auf `null` setzen.

Ist aber `firstNode.link == 0` falsch, dann muß man den vorletzten Knoten finden. Man erkennt den vorletzten Knoten daran, daß der Nachfolger seines Nachfolgers `null` ist. Man bekommt daher das Programm

```
public void deleteLastNode() {

    ListNode  previousNode;
    ListNode  currentNode;

    if(firstNode != null) {
        if(firstNode.link == null) {

            firstNode = null;
        } else {
            currentNode = firstNode;

            while (currentNode.link.link != null) {
                currentNode = currentNode.link;
            }
            currentNode.link = null;
        }
        length --;
    }
}
```

Dieses Programm ist nicht optimal, weil `currentNode.lnk` zweimal hintereinander bestimmt werden muß. Besser ist folgende Variante:

```
public void deleteLastNode() {

    ListNode  previousNode;
    ListNode  currentNode;
    ListNode  nextNode;

    if(firstNode != null) {
        if(firstNode.link == null) {
```

```

    firstNode = null;
} else {
    previousNode = firstNode;
    currentNode = firstNode.link;
    nextNode = currentNode.link;

    while (nextNode != null) {
        previousNode = currentNode;
        currentNode = nextNode;
        nextNode = currentNode.link;
    }
    previousNode.link = null;
}
length --;
}
}

```

Die Methode benötigt Zeit $O(n)$.

Die Methode `insertNewLastNode` arbeitet ähnlich. Man erzeugt zuerst einen neuen Knoten mit der vorgegebenen Information. Dann unterscheiden wir zwei Fälle:

1. Die Liste ist leer. Dann ist der erste Knoten der neue letzte Knoten.
2. Die Liste ist nicht leer. Dann muß man den letzten Knoten suchen und den neuen letzten Knoten anhängen.

```

public void insertNewLastNode(Object info) {
    ListNode N = new ListNode();
    N.info = info;
    N.link = null;

    if(firstNode == null) {
        firstNode = N;
    } else {
        ListNode P = firstNode;
        while (P.link != null) {
            P = P.link;
        }

        P.link = N;
        length++;
    }
}

```

Die Laufzeit ist $O(n)$.

Schließlich noch die Methode `print`.

```
public void print() {  
  
    ListNode N;  
  
    System.out.print("(");  
  
    N = firstNode;  
  
    while (N != null) {  
        System.out.print(N.info.toString());  
        N = N.link;  
        if (N != null) System.out.print(",");  
    }  
  
    System.out.print(")");  
}
```

Diese Methode hat die Laufzeit $O(n)$. Die Länge der Ausgabe ist proportional zu n und darum ist das effizient.

53. Exercise Implementieren Sie eine Methode `deleteFirstNode`.

Kapitel 7

Rekursion

7.1 Berechnung von Quadratsummen

Wir untersuchen folgendes Berechnungsproblem: Seien m und n natürliche Zahlen, $m \leq n$. Berechne $\sum_{i=m}^n i^2$.

7.1.1 Iterative Lösung

Am einfachsten ist folgende iterative Lösung.

```
static int sumSquaresIt(int m, int n) {
    int i, sum;

    sum = 0;
    for (i = m; i <=n; i++) sum = sum + i*i;
    return sum;
}
```

Sie benötigt $n - m + 1$ Quadrierungen und Additionen.

Wir beschreiben jetzt eine andere Möglichkeit, diese Summe zu berechnen. Setze

$$S(m, n) = \sum_{i=m}^n i^2.$$

Die Funktion kann erfüllt folgende Eigenschaften:

1. $S(n, n) = n^2$.

2. Für $n > m$ ist $S(m, n) = S(m, n - 1) + S(n, n)$.

Dies kann man als eine induktive Definition von $S(m, n)$ betrachten. Gleichzeitig enthält diese Definition eine Methode, die Funktion S zu berechnen. Sie läßt sich so programmieren.

```
static int sumSquaresRe1(int m, int n) {
    if (n == m)
        return n*n;
    else
        return sumSquaresRe1(m,n-1) + sumSquaresRe1(n,n);
}
```

So etwas nennt man ein *rekursives Programm*. Die Idee ist, die Lösung des Berechnungsproblems auf die Lösung von Teilproblemen zurückzuführen. Das muß schließlich auf Anfangsfälle führen, deren Lösung explizit angegeben wird.

Man beachte den starken Bezug zur vollständigen Induktion.

Dies ist nicht die einzige Lösung des Problems. Man kann die Funktion S auch so definieren:

1. $S(m, m) = m^2$.
2. Für $n > m$ ist $S(m, n) = S(m, m) + S(m + 1, n)$.

Daraus wird das Programm

```
static int sumSquaresRe2(int m, int n) {
    if (n == m)
        return m*m;
    else
        return sumSquaresRe2(m,m) + sumSquaresRe2(m+1,n);
}
```

Schließlich gibt es auch noch folgende Alternative:

1. $S(m, m) = m^2$.
2. Für $n > m$ und $k = (m + n)/2$ ist $S(m, n) = S(m, k) + S(k + 1, n)$.

Dies führt zu dem Programm

```
static int sumSquaresRe3(int m, int n) {
    int k;
    if (n == m)
        return m*m;
    else {
        k = m+n/2;
        return sumSquaresRe3(m,k) + sumSquaresRe3(k+1,n);
    }
}
```

7.1.2 Analyse der Quadratsummenprogramme

Wir analysieren die rekursiven Programme.

Das erste rekursive Programm hat folgende Aufrufstruktur: Beispiel mit kleinen Zahlen.

Bild

In jedem Schritt wird genau ein Quadrat ausgerechnet und es wird ein rekursiver Aufruf gemacht. Wenn jede Quadratberechnung konstante Zeit kostet, dann braucht die ganze Berechnung Zeit $O(m - n)$.

Dasselbe gilt für das zweite rekursive Programm.

Im dritten rekursiven Programm ist die Aufrufstruktur komplizierter. Hier erhalten wir einen Baum. Wir berechnen die Anzahl der rekursiven Aufrufe für eine Zweierpotenz. Sei also $m = 1$, $n = 2^k$.

54. Theorem *Ist $m = 1$ und $n = 2^k$ dann ist die Anzahl der rekursiven Aufrufe $2^{k+1} - 1$.*

Proof: Im ersten Schritt ist die Anzahl 1. Dann 2. Dann 4. usw. zum Schluss 2^k . Daher ist die Gesamtzahl

$$\sum_{i=1}^k 2^i = 2^{k+1} - 1.$$

■

Am Schluß muß man 2^k Quadrierungen vornehmen. Im allgemeinen ist die Anzahl der Aufrufe zwischen 2^k und 2^{k+1} wobei $k = \lfloor \log_2 n \rfloor$ ist. Damit ist diese Variante ineffizienter als die beiden anderen.

55. Exercise Ist n eine natürliche Zahl, Dann ist $n! = 2 * 3 * 4 * 5 * \dots * n$.

1. Schreiben Sie zwei verschiedene rekursive Programme, die $n!$ berechnen. Das erste soll die Berechnung von $n!$ auf die Berechnung von $(n-1)!$ zurückführen. Das zweite soll die Berechnung von $n!$ auf die Berechnung von $(n/2)!$ zurückführen.
2. Geben Sie den Aufrufbaum Ihrer Programme für $n = 8$ an und bestimmen Sie die Anzahl der rekursiven Aufrufe des Programms und die Anzahl der Multiplikationen.
3. Beweisen sie durch vollständige Induktion, daß die Anzahl der rekursiven Aufrufe in diesen Programmen eine monoton wachsende Funktion von n ist und daß die Anzahl der Multiplikationen genau $n - 1$ ist.
4. Zeigen Sie durch vollständige Induktion, daß die Anzahl der rekursiven Aufrufe in beiden Programmen von durch eine lineare Funktion in n beschränkt ist.
5. Welche Variante ist die effizientere?
6. Schreiben Sie ein iteratives Programm zur Berechnung von $n!$. Vergleichen Sie seine Effizienz mit der der rekursiven Programmen.

7.2 Umkehr der Knoten in einer verketteten Liste

Dieses Problem kann man auch rekursiv lösen. Die Idee ist folgende:

1. Trenne den ersten Knoten ab.
2. Kehre den Rest um.
3. Hänge den ersten Knoten hinten dran.

Wir brauchen also noch auch noch eine Zusammensetzungsmethode. Die wird nun rekursiv implementiert.

In den Implementierungen gehen wir folgendermaßen vor. Eine Variable vom Typ Listenknoten stellt eine Liste dar. Zeigt die Variable ins nichts, dann wird die leere Liste dargestellt. Sonst wird die Liste dargestellt, die mit dem Listenknoten beginnt, auf den die Variable zeigt.

```
class ListNode {
    ListNode link;
    Object info;
```

```
ListNode concat(ListNode node) {
    if (this == null)
        return node;
    else if (this.link == null)
        this.link = node;
    else
        this.link.concat(node);
    return this;
}

ListNode reverse (ListNode node) {
    if (this == null || this.link != null)
        return this;
    else
        return this.link.reverse.concat(this);
}
}
```

Damit kann man jetzt leicht eine Umkehrfunktion für verkettete Listen implementieren.

7.3 GGT

Rekursion

```
static int gcd(int a, int b) {
    a = abs(a);
    b = abs(b);
    if (b == 0) return a;
    else return gcd(b, a%b);
}
```

7.4 Der Turm von Hanoi

Rekursionsidee:

Der Turm hat die Höhe n .

Wir erklären, wie man einen Turm von n Elementen auf einen Stab umschichtet.

Wenn $n = 0$ ist, dann mache nichts.

Wenn $n > 0$ ist, dann

1. schichte $n - 1$ in die Mitte um,
2. lege den letzten nach rechts,
3. schichte wieder $n - 1$ nach rechts.

Das wird analysiert. Sei $T(n)$ die Zeit für einen Turm der Höhe n . Dann ist

$$T(n) = 2T(n - 1) + c.$$

Also ist $T(n) = 2T(n - 1) + c = 2(2T(n - 2) + c) + c = 2^2T(n - 2) + c(2 + 1) = 2^3T(n - 3) + c(2^2 + 2 + 1) = \dots = 2^nT(0) + c \sum_{i=0}^{n-1} 2^i = 2^n(T(0) + c) - c$. Einen solchen Algorithmus nennt man *exponentiell*, weil seine Laufzeit eine Exponentialfunktion der Eingabelänge ist.

Hier haben wir es also mit einer eleganten aber ineffizienten Lösung zu tun.

7.5 Merge-Sort

Eine rekursive Methode, ein Array von ints zu sortieren ist *Merge-Sort*. Die Idee besteht darin, das Array in zwei Hälften zu teilen, jede Hälfte zu sortieren und die beiden Hälften dann zusammenzufügen.

Wir spezifizieren zwei Methoden.

`mergeSort(int[] A, int k, int n)` sortiert den Abschnitt `A[k:n]`.

`merge(int[] A, int k, int m, int n)` sortiert den Abschnitt `A[k:n]` durch zusammenfügen unter der Voraussetzung, daß `A[k:m]` und `A[m+1:n]` schon sortiert sind.

Wir erklären die Ideen für `merge`. Man verwendet ein Hilfsarray `B`. Man nimmt `A[k]`. Dann überträgt man die Einträge von `A[m+1:k]` solange nach `B`, bis man einen findet, der größer als `A[k]` ist. Dann trägt man `A[k]` ein. Dasselbe macht man mit `A[k+1]`. Man sieht, daß diese Methode für einen Abschnitt der Länge s Zeit $O(s)$ braucht.

Wir erklären an dieser Stelle noch, wie man eine solche Methode implementieren kann. Da ein Array kein Objekt im strengen Sinne ist, kann die Methode auch nicht auf einem Objekt operieren. Wir machen Sie zu einer Methode, die unabhängig von den Objekten der Klasse ist und daher nur von der Klasse abhängt. Solche Methoden deklariert man als `static`. Sie werden mit dem Klassennamen aufgerufen.

Die Methode `mergeSort(A,k,n)` funktioniert so:

1. Wenn $k == n$ wahr ist, dann macht die Methode nichts.
2. Sie berechnet $m = (k + n)/2$.
3. Sie ruft `mergeSort(A,k,m)` und `mergeSort(A,m+1,n)` auf.
4. Sie ruft `merge(A,k,m,n)` auf.

Wir analysieren das Verfahren.

Wir haben bereits gesehen, daß die Laufzeit von `merge` bei einem Array-Abschnitt der Länge s beschränkt durch $c_1 s$ ist mit einer Konstanten c_1 . Wir wollen eine obere Schranke $T(n)$ für die Laufzeit von Merge-Sort bei einem Array-Abschnitt der Länge n beweisen.

Wir betrachten zuerst Array-Abschnitte deren Länge eine Zweierpotenz ist, also $n = 2^t$. Dann gilt

$$T(2^t) \leq 2T(2^{t-1}) + c_1 2^t \leq \dots \leq 2^t T(1) + t c_1 2^t = n(T(1) + c_1 \log_2 n).$$

Um den allgemeinen Fall zu behandeln, nehmen wir an, daß die Funktion $T(n)$ monoton wachsend ist. Setzt man $t = \lceil \log_2 n \rceil$ dann gilt

$$T(n) \leq T(2^{t+1}) \leq 2n(T(1) + c_1(1 + \log_2 n)) = O(n \log n).$$

- 56. Exercise**
1. Schreiben Sie eine rekursive (statische) Methode `mergeSort(int[] A, int m, int n)`, die einen Teil `A[m:n]` eines Arrays von `ints` mittels Merge-Sort sortiert. Diese Methode soll eine Methode `merge(int[] A, int k, int m, int n)` verwenden, die den Teil `A[k:n]` des Arrays `A` sortiert unter der Annahme, das die beiden Teile `A[k:m]` und `A[m+1,n]` bereits sortiert sind. Die Methode `merge` sollen Sie auch implementieren.
 2. Schreiben Sie auch eine Methode `InsertionSort(int[] A, int k, int n)`, die das Array `A` mit Insertion Sort sortiert.
 3. Beweisen Sie, daß Insertion-Sort bei einem Abschnitt der Länge n Laufzeit $O(n^2)$ hat.
 4. Beispiel-Rechnungen für arrays wachsender Länge und Vergleich.

Kapitel 8

Modularität und Datenabstraktion

8.1 Einleitung

In diesem Kapitel werden zwei wichtige Prinzipien der objektorientierten Programmierung vertieft, nämlich die Modularität und die Datenabstraktion.

Das ganze wird am Beispiel von Warteschlangen (Priority Queues) erklärt.

8.2 Warteschlangen

Warteschlangen gibt es an Bushaltestellen. Wenn man einen Stapel von Rechnungen hat, dann müssen die in einer bestimmten Reihenfolge bezahlt werden. Die Rechnungen bilden also auch eine Warteschlange. Berechnungsaufgaben, die ein Computer lösen muß, stehen auch in einer bestimmten Warteschlange.

Was haben diese Warteschlangen gemeinsam? Jedes Element der Warteschlange hat eine bestimmte Priorität. Die Prioritäten kann man miteinander vergleichen. Wer die höchste Priorität hat, kommt als nächster dran. Wenn ein neues Element dazu kommt, bekommt es eine Priorität und muß sich in die Warteschlange einordnen. In den ersten Beispielen muß sich das neue Element immer hinten anstellen. Bei den Jobs, die ein rechner erledigen muß oder bei den Anträgen, die ein Sachbearbeiter bearbeiten muß, gibt es neue Elemente die möglicherweise eine höhere Priorität haben als alte.

Das bedeutet, daß Warteschlangen folgende Operationen unterstützen müssen:

1. Konstruktion einer leeren Warteschlange.

2. Bestimmung der Länge der Warteschlange, also der Anzahl der Elemente in der Warteschlange.
3. Einfügen eines neuen Elementes in die Warteschlange.
4. Wenn die Warteschlange nicht leer ist, Entfernung des Elementes höchster Priorität aus der Warteschlange.

Um Warteschlangen in anderen Programmen benutzen zu können, verwendet man einen abstrakten Datentyp `PriorityQueue`. In Java realisiert man den als Java-Klasse. Um die Klasse in anderen Programmen benutzen zu können, braucht man nur zu wissen, wie die Konstruktoren und Methoden heißen, die obige Funktionen realisieren. Man braucht aber nicht zu wissen, wie die realisiert sind.

Alles was mir jetzt wissen müssen ist, von welchem Typ die Einträge in einer Warteschlange sind und welche Konstruktoren und Methoden für die Warteschlangen realisiert sind.

Die Einträge in der Warteschlange sind vom Typ `Comparable`. Dieser Typ wird später noch genauer erklärt. Es handelt sich dabei um Elemente einer Menge, auf der eine Totalordnung erklärt ist. Von je zwei Objekten vom Typ `Comparable` kann man also sagen, ob sie gleich sind oder welches der beiden größer ist.

Die Konstruktoren und Methoden sind folgende:

1. Konstruktor `PriorityQueue()` konstruiert eine leere Priority Queue.
2. Methode `int size()` liefert die Länge der Warteschlange.
3. Methode `void insert(Comparable X)` fügt X in die Warteschlange ein.
4. Methode `Comparable remove()` liefert das Element mit der höchsten Priorität und entfernt es aus der Warteschlange.

8.3 Anwendung von Warteschlangen

Mit den Informationen aus dem letzten Abschnitt können wir Warteschlangen verwenden, ohne zu wissen, wie sie implementiert sind. Als Beispiel implementieren wir ein Sortierverfahren.

Die Situation ist folgende. Wir haben ein Array A vom Typ `Comparable`. Das wollen wir sortieren. Die Idee ist einfach. Wir fügen Element für Element des Arrays in eine Warteschlange ein. Dann entfernen wir das Element mit höchster Priorität aus der Warteschlange, setzen es an die erste Stelle des Arrays. Danach entfernen wir das zweithöchste Element aus der Warteschlange und setzen es an die zweite Stelle des Arrays usw. Das sieht so aus:

```

| void priorityQueueSort(ComparisonKey[] A) {
|
|     int i;                // let i be an integer array index variable
|
5 |     int n = A.length;    // let n be the length of the array A to be sorted
|
|     PriorityQueue PQ = new PriorityQueue(); // let PQ be initially empty
|
|     for (i = 0; i < n; i++) PQ.insert(A[i]); // put A's items into PQ
10 |
|     for (i = n-1; i >=0; i--) A[i] = PQ.remove(); // remove PQ's items
|                                           // and put them in A
| }

```

Um diese Methode zu implementieren, braucht man nicht zu wissen, wie die Methoden und Konstruktoren der Warteschlange implementiert sind. Man braucht nur zu wissen was sie machen. Die Methode kann also implementiert werden, bevor die Implementierung der Warteschlange fertig ist. Die Implementierung der Warteschlange kann sogar verändert und optimiert werden. Das liegt daran, daß der Benutzer nur durch die drei Methoden Zugriff auf die Warteschlange hat und durch sonst nichts.

8.4 Implementierungen von Warteschlangen

Wir haben jetzt die Anwendung von Warteschlange gesehen. Jetzt ist es Zeit, über ihre Implementierung nachzudenken.

Wir verwenden hier zwei einfache Realisierungen für Warteschlangen.

Zuerst wird die Warteschlange als sortierte verkettete Liste realisiert. Das Element von höchster Priorität zu entfernen, ist dann leicht, weil es vorn steht. Die Methode erfordert Zeit $O(1)$. Ein neues Element einfügen ist aber aufwendiger. Die richtige Stelle muß gefunden werden und in diese Stelle muß das neue Element eingehängt werden. Das erfordert Zeit $O(n)$ wobei n die Länge der Warteschlange ist.

Dann wird die Warteschlange als unsortiertes Array realisiert. Einfügen ist dann einfach, weil es nur hinten anhängen bedeutet. Das erfordert Zeit $O(1)$. Dagegen ist Entfernen schwerer, weil erst mal das Element von höchster Priorität rausgesucht werden muß. Das erfordert Zeit $O(n)$.

8.5 Warteschlangen auf der Basis von verketteten Listen

Wir implementieren die erste Methode. In der Klasse `PriorityQueue` wollen wir intern einen Parameter `count` für die Länge der Liste benutzen und einen Listenknoten `itemList`, der der erste Knoten der verketteten Liste ist und daher die verkettete Liste repräsentiert. Die beiden Parameter werden als `private` deklariert. Sie sind außerhalb der Klasse nicht sichtbar. Wir haben ja schon festgelegt, wie man auf die Klasse zugreifen kann.

Die Methode `size` liefert einfach den Wert von `count`.

Die Methode `remove` fragt erst, ob die Liste leer ist, ob also `count == 0` ist. In diesem Fall gibt sie `null` zurück. Andernfalls

1. merkt sie sich, was der Inhalt des ersten Knotens ist,
2. setzt den ersten Knoten auf den Link des ersten Knotens,
3. reduziert die Länge um 1,
4. gibt den Inhalt des ersten Knotens zurück.

Die Methode `insert` muß ein neues Element in die sortierte verkettete Liste einfügen. Dazu wird eine Methode `sortedInsert` verwendet, die ein Objekt vom Typ `ComparableKey` in die sortierte verkettete Liste einordnet. Diese Methode ist aber nur eine Hilfsmethode. Sie soll nicht von außen verwendbar sein, weil dort ja nur die festgelegten Methoden zur Verfügung stehen sollen. Sie wird daher als `private` deklariert.

Die Methode arbeitet rekursiv. Entweder fügt sie das neue Item an die erste Stelle ein. Das muß sie tun, wenn die Liste leer ist oder wenn der Inhalt des neuen Knotens eine Priorität hat, die wenigstens so groß ist wie die des ersten Knotens. Andernfalls wird das neue Item in die Liste eingefügt, die mit dem zweiten Knoten beginnt.

Wenn man diese Methode hat, ist die Methode `insert` trivial. Man fügt einfach das neue Item in die sortierte verkettete Liste ein und erhöht den Zähler um eins.

8.6 Warteschlangen auf der Basis von Arrays

Hier werden folgende privaten Variablen benutzt:

1. `count` für die Länge der Warteschlange.

2. `capacity` für die Länge des verwendeten Arrays.
3. `capacityIncrement` für die Erweiterungslänge; das Array muß ja erweitert werden, wenn es zu kurz ist, die Warteschlange zu beherbergen.
4. `itemArray` ist das array, das die Warteschlange speichert.

Der Konstruktor `PriorityQueue()` setzt die initialen Werte fest und erzeugt das Array, das die Warteschlange speichert.

Die Methode `insert` ist einfach. Wenn die Kapazität erschöpft ist, braucht man ein längeres Array. Da Arrays eine feste Länge haben, braucht man ein neues Array, kopiert das alte rein und legt den zeiger um. Dann wird die Länge um eins erhöht und das neue Item hinten angehängt.

In der Methode `remove` wird das größte Element und seine Position gesucht. An die Position des größten Elementes wird das letzte Element geschrieben, die Länge wird um eins reduziert und das größte Element wird zurückgegeben.

8.7 ComparisonKey

Wir müssen noch beschreiben, wie die Elemente der Warteschlange realisiert werden. Wir haben dafür den Typ `ComparisonKey` verwendet. Solche `ComparisonKeys` muß man vergleichen können. Genauer gesagt muß man auf sie die folgenden Methoden anwenden können:

1. `int compareTo(ComparisonKey value)`. `X.compareTo(Y)` gibt -1,0,1 zurück je nachdem ob X kleiner, gleich oder größer Y ist.
2. `String toString()` gibt eine Stringdarstellung zurück.

Als konkrete Elemente einer Warteschlange kann man sich ganz viele verschiedene Objekte vorstellen. Wenn wir die Java-Techniken, die wir bisher gelernt haben, verwenden wollen, müssen wir eine Klasse `ComparisonKey` schreiben und davon die Klasse, die die konkreten Elemente implementiert, ableiten. Das schränkt uns aber erheblich ein, weil es in Java keine Mehrfachvererbung gibt. Wenn wir also zum Beispiel `Rectangles` in die Warteschlange einordnen wollten, ginge das nicht, weil wir keine Unterklasse erzeugen könnten, die gleichzeitig von `Rectangle` und von `ComparisonKey` erbt.

Man benutzt statt dessen *Interfaces*. Diese sehen aus wie abstrakte Klassen, haben aber ausschließlich abstrakte Methoden Konstanten.

Eine Klasse kann ein Interface implementieren. Dazu müssen alle abstrakten Methoden des Interfaces implementiert werden. Klassen können mehrere Interfaces implementieren.

Wenn eine Klasse ein Interface implementiert, können Variablen vom Typ des Interfaces Objekte vom Typ der Klasse zugewiesen werden und die Methoden, die im Interface stehen, können auf das Objekt angewendet werden.

In unserem Fall brauchen wir das Interface `ComparisonKey`.

```

| interface ComparisonKey {
|
|     // if k1 and k2 are ComparisonKeys, k1.compareTo(k2) has the
|     // value 0, +1, or -1 according as k1 == k2, k1 > k2, or k1 < k2 in
5 |     // the order of priority defined by the compareTo method
|
|         int compareTo(ComparisonKey value);
|
|
|
10 |     // converts a ComparisonKey object to a printable string
|
|         String toString();
| }

```

8.8 Klasse PQItem

Wir können damit z.B. eine Klasse `PQItem` implementieren.

Die Informationen sind vom Typ `int`. In der zweiten Implementierung sind die Informationen vom Typ `string`.

8.9 Übungen

Kapitel 9

Stacks und Queues

9.1 Stacks

Wir besprechen jetzt eine weitere abstrakte Datenstruktur, einen Stack oder auf Deutsch Keller oder Stapel.

Diese Datenstruktur unterstützt nur folgende Operationen

1. `push` ein Element auf den Speicher legen,
2. `pop`, ein Element aus dem Speicher holen.

Diese Datenstruktur ist elementar bei der Syntaxanalyse und bei der Ausführung von Programmen.

Wenn z.B. Methoden aufgerufen werden, dann wird die Methode auf den Stack gepusht. Wenn die Methode eine andere aufruft, wird die andere auf den Stack gepusht. Wenn die oberste Methode fertig ist, wird sie wieder gepopt. Wir erklären das im nächsten Abschnitt noch genauer.

9.2 Kellerautomaten

Ein einfaches Berechnungsmodell ist das der Kellerautomaten. Sie spielen in der Theorie der formalen Sprachen eine wichtige Rolle und werden in Informatik VI ausführlich benutzt. Wir beschreiben dieses Modell und erklären, wie man es dazu verwenden kann einen Term auf seine syntaktische Korrektheit zu prüfen und auszuwerten.

Kellerautomaten haben ein Eingabeband, einen Keller und endlich viele Zustände. Einer davon ist der Haltezustand. Auf dem Eingabeband steht ein Eingabestring, dessen Zeichen aus einem endlichen Alphabet kommen. Auf dem Band könnte z.B. ein Term stehen. Der Automat arbeitet in Schritten. In einem Schritt ist der Automat in einem bestimmten Zustand und im Keller stehen Zeichen aus dem Alphabet. Er guckt sich die obersten Zeichen auf dem Keller an. Abhängig vom Zustand und von den obersten Zeichen auf dem Keller kann er dann eine oder mehrere der folgenden Aktionen machen

- Er liest ein Zeichen vom Eingabeband und löscht es.
- Er poppt Zeichen aus dem Keller.
- Er pusht Zeichen auf den Keller.
- Er geht in einen neuen Zustand.

Wichtig ist, daß die Anzahl der Zeichen, die der Automat auf den Keller pusht oder aus dem Keller poppt global beschränkt ist. Der Kellerautomat ist fertig, wenn er im ausgezeichneten Haltezustand ist.

Wir erläutern, wie ein Kellerautomat aussieht, einen unvollständig geklammerter Term auswertet.

Der Automat hat vier Zustände: Start, in Arbeit, fertig.

Auf dem Eingabeband steht der Term. Er endet mit #.

Wir illustrieren die Arbeit des Automaten an einem Beispiel. Der Term ist

$$(3 + \max(2, 3)) < 4$$

Auf dem Eingabeband steht

$$3 + \max(2, 3) < 4\#$$

Der Automat liest von links nach rechts.

1. Zuerst wird @ auf den Stack gepusht.
2. 3 wird gelesen. Das ist ein Literal. Der Wert wird bestimmt. Er ist auch 3. Er wird gepusht. Der Stack ist 3@.
3. + wird gelesen und gepusht. Der Stack ist +3@.
4. max wird gelesen und gepusht. Der Stack ist max+3@.

5. (wird gelesen und gepusht. Der Stack ist (max+3@.
6. 2 wird gelesen und gepusht. Der Stack ist 2(max+3@.
7. , wird überlesen.
8. 2 wird gelesen und gepusht. Der Stack ist 32(max+3@.
9.) wird gelesen. $3 = \max(3, 2)$ wird ausgewertet. 32(max wird gepopt. 3 wird gepusht.
10. Die Eingabe ist zuende. # wird gepusht. Der Stack ist #3 + 3@.
11. Oben auf dem Stack steht #3+3. Das wird gepopt. $6 = 3+3$ wird ausgewertet und gepusht.

Am Schluß steht auf dem Stack das Ergebnis.

Wir wollen jetzt den Kellerautomaten formal definieren.

Der Kellerautomat braucht eine endliche Menge K von Zuständen, einen Haltezustand $h \in K$, ein endliches Alphabet Σ , das die Zeichen enthält, die auf dem Eingabeband und auf dem Keller stehen können und eine partielle Funktion

$$\delta : K \times \Sigma \cup \{\epsilon\} \times \Sigma^* \rightarrow K \times \Sigma^*.$$

Diese Funktion legt fest, in welchen Zustand der Automat geht und was er auf den Keller in jeder Situation schreibt. Ist

$$\delta(k, s, u) = (k', u'),$$

so ist der neue Zustand k' und der String u oben auf dem Keller wird durch u' ersetzt.

Die Funktion erfüllt einige Bedingungen:

1. Ihr Definitionsbereich ist endlich. Sie kann also z.B. in einer Tabelle gespeichert werden.
2. Sei $k \in K$ und $(k, x, u), (k, y, v)$ im Definitionsbereich von δ . Sei x ein Präfix von y oder umgekehrt und sei u ein Präfix von v oder umgekehrt. Dann ist $(x, y) = (u, v)$.

Wir definieren jetzt den Kellerautomaten, der einen unvollständig geklammerten Term auswertet.

Es gibt drei Zustände:

1. Zustand s: Start.
2. Zustand a: in Arbeit.
3. Zustand h: fertig.

9.2.1 Start

Wenn der Automat im Start-Zustand ist, geht er ohne ein Zeichen zu lesen in den Zustand a über und pusht $@$. Formal heißt das

$$\delta(s, \epsilon, \epsilon) = (a, @).$$

9.2.2 Zeichen lesen und pushen

Sei $w = @$ oder $w = a@$ oder $w = abc$ mit $abc \neq \omega_2 x \omega_1$, wobei ω_1, ω_2 Infixoperationen sind mit $\pi(\omega_2) \leq \pi(\omega_1)$, dann werden alle gelesenen Zeichen außer Literalen, Kommas und schließenden Klammern auf den Stack gepusht. Ist also s kein Literal, Komma, oder eine schließende Klammer, dann ist

$$\delta(a, s, w) = (a, sw).$$

Kommas werden überlesen, also nichts passiert.

$$\delta(a, ,, w) = (a, w).$$

Literale werden ausgewertet und auf den Stack gepusht.

Ist also s ein Literal, so gilt

$$\delta(a, s, w) = (a, g(s)w).$$

9.2.3 Auswerten ohne Lesen

$w = \omega_2 x_2 \omega_1 x_1$, ω_1, ω_2 beides Infixoperatoren, x_1, x_2 beides Werte, $\pi(\omega_2) \leq \pi(\omega_1)$.

Der Automat berechnet $x = g(\omega_1)(x_1, x_2)$, poppt w und pusht $\omega_2 x_2$. Es ist also

$$\delta(a, \epsilon, w) = (a, \omega_2 g(\omega_1(x_1, x_2))).$$

9.2.4 Auswerten bei schießender Klammer

Das gelesene Zeichen ist (.

$w = x$ (: w wird gepopt, x wird gepusht.

$$\delta(a, (, w) = (a, x).$$

$w = x_2\omega x_1$): w wird gepopt und $g(\omega)(x_1, x_2)$ wird gepusht.

$$\delta(a, (, w) = (a, g(\omega)(x_1, x_2)).$$

$w = x_k \dots x_1$ (ω : w wird gepopt und $g(\omega)(x_1, \dots, x_k)$ wird

$$\delta(a, (, w) = (a, g(\omega)(x_1, \dots, x_k)).$$

9.2.5 Eingabe zuende

wird gepusht.

$$\delta(a, \#, \epsilon) = (a, \#).$$

9.2.6 Endauswertung

$w = \#x@$, x ist ein Wert.

Der Automat geht in den Haltezustand, poppt w und pusht Ergebnis x auf den Stack und geht in den Zustand fertig.

$$\delta(a, \epsilon, \#x@) = (h, x).$$

$w = \#x_2\omega x_1$.

Der Automat geht in den Haltezustand, poppt w und pusht Ergebnis $x = g(\omega)(x_1, x_2)$ auf den Stack und geht in den Zustand fertig.

$$\delta(a, \epsilon, \#x_2\omega x_1) = (a, g(\omega)(x_1, x_2)).$$

9.3 ADT Stack

Der beschriebene Kellerautomat soll nun realisiert werden. Dazu spezifizieren wir einen abstrakten Datentyp Stack. Die Menge X bezeichnet die Menge aller Objekte.

Wir geben die Methoden an, die ein Stack haben soll.

```
S = new Stack() \\ erzeugt ein leeres Stack
S.empty() \\ liefert true, wenn S leer und false andernfalls
S.push(X) \\ pusht X
X = S.pop() \\ X ist das oberste Zeichen, es wird gepopt
X = S.peek() \\ X ist das oberste Zeichen.
```

9.4 Stack-Implementierungen

Mit Arrays oder linked Lists.

9.5 Implementierung von Rekursion mit Stacks

Wir erläutern eine weitere Anwendung von Stacks: die Auswertung rekursiver Funktionsaufrufe durch die Java Virtual Machine.

Wenn die Funktion $f(a_1, \dots, a_n)$ aufgerufen wird, dann wird auf dem Stack zuerst mal ein Rahmen erzeugt. Dieser Rahmen enthält

1. Platz für den Rückgabewert,
2. Einen Verweis auf den vorhergehenden Rahmen im Speicher,
3. Die Adresse des Codes, bei dem es weitergeht, wenn die Funktion fertig ist,
4. Platz für die aktuellen Parameter der Funktion,
5. Platz für die lokalen Variablen der Funktion.

Dieser Rahmen wird auf den Speicher gepusht und erst wieder gepopt, wenn der Funktionsaufruf fertig ist. Wird innerhalb der Funktion eine weitere Funktion aufgerufen, dann wird für die auch so ein Rahmen erzeugt und auf den Keller gepusht.

Das wird am Beispiel der Auswertung von $3!$ erläutert.

Benutzt wird die Funktion

```
double factorial(int n){
    if (n <= 1){
        return 1.0;
    }else{
        return n*factorial(n-1);
    }
}
```

Aufgerufen wird als $x = factorial(3)$

usw. wie im Buch.

9.6 Queues

Ein weiterer wichtiger abstrakter Datentyp ist `Queue`.

Queues unterstützen First in First out.

Hier sind die Methoden:

```
S = new Queue(); // erzeugt eine leere Queue
Q.empty(); // liefer true, wenn die Queue leer ist und false sonst
Q.insert(X); // fügt neues Element hinten in Queue ein
X = Q.remove; // holt das erste Element raus.
```

Hier gibt es auch wieder verschiedene Implementierungsmethoden.

Implementiert man Queues mit verketteten Listen, dann ist es günstig, einen Zeiger vom ersten Element auf das letzte zu haben. Sonst dauert das Einfügen zu lang.

Diese Art der Implementierung eignet sich für Queues, deren Länge sehr variiert, wie z.B. Telefon-Antwort-Queues.

Implementiert man Queues mit Arrays (sequentielle Implementierung), so verwendet man folgende Idee.

Man nimmt ein Array.

Man merkt sich die Position `front` des ersten Elementes, die Position `rear` des letzten Elementes, die Länge `count` der Queue, Die Länge `capacity` des Arrays.

Wird ein neues Element hinten angefügt, dann wird `rear = (rear + 1)%capacity` gesetzt und das neue Element kommt an Position `rear`.

Dies ist besser für Queues fester Länge Job-Queue mit maximaler Anzahl von Jobs).

9.7 Anwendungen von Queues

Drucker Puffer. Queue, in die zu druckende Zeilen geschrieben werden.

Drucker-Spooler. Erst wird der ganze Job in eine Queue geschrieben. Die entsteht im Spooler.

Printer Queue. Enthält die Druckjobs.

9.8 Übungen

Präsenzübung:

Diskutieren und erweitern Sie den Kellerautomat, der unvollständig geklammerte Terme auswertet.

- a) Bestimmen Sie den Definitionsbereich der Funktion δ .
- b) Modifizieren Sie den Kellerautomaten so, daß er nur die Korrektheit der Syntax des Uterms prüft. Am Schluß soll 0 auf dem Keller stehen, wenn die Syntax falsch war und 1 andernfalls.

Hausübung:

Im Buch von Standish (und auf dem Netz) steht ein Applet, das P-Terme auswertet, in denen nur die Operatoren +, *, /, vorkommen.

- a) definieren Sie einen kellerautomaten, der genauso arbeitet, wie das angegebene Programm.
- b) bringen Sie das Programm zum Laufen mit den beiden Stack-Implementierungen von Standish und mit der Stack-Implementierung aus Java.
- c) in dem Programm von Standish wird Division durch Null nicht abgefangen. Ändern Sie das Programm so, daß Division durch Null doch abgefangen wird.

Kapitel 10

Verifikation

In diesem Kapitel geht es darum zu beweisen, das ein Programm das macht, was es machen soll.

10.1 Formaler Rahmen

Beweise sind mathematische Verfahren. Wir brauchen also eine mathematischen Formalismus. Man behauptet die Korrektheit eines Programms, indem man schreibt:

```
Precondition:  
program  
Postcondition:
```

Schreibt man sowas, dann meint man: Angenommen es gilt die Vorbedingung (Precondition). Nachdem das Programm fertig ist, gilt auch die Nachbedingung. Das kann man als mathematischen Satz auffassen.

57. Example Betrachte den folgenden Satz:

```
// Precondition: x == 0, y >= 4  
x = y+3;  
y = y-4;  
// Postcondition: x >= 5, y >= 0
```

Wir beweisen den Satz. Aus $y \geq 4$ folgt $y + 3 \geq 7$, also ist nach der Zuweisung $x = y + 3$ $x \geq 7 > 5$. Ferner ist $y - 4 \geq 0$. Also nach der Zuweisung $y = y - 4$ gilt auch $y \geq 0$.

Es zeigt sich, daß wir sogar noch mehr beweisen können, als in der Postcondition behauptet wurde.

Das Programm heißt *total korrekt*, wenn es terminiert und nach der Terminierung die Nachbedingung gilt. Wird die Terminierung nicht gefordert, so heißt das Programm *partiell korrekt*.

10.2 Korrektheit von Methodenaufrufen

Will man eine Methode verifizieren, dann muß man Vor- und Nachbedingungen formulieren, die sich auf das aktuelle Objekt, die formalen Parameter und den Rückgabewert beziehen.

Wir besprechen ein Beispiel.

Das folgende Programm findet einen Index j zwischen m und n , für den $A[j]$ wenigstens so groß ist wie alle Einträge des Arrays in diesem Bereich.

```
static int findMax(int[] A, int m, int n) {
    int i;
    int j;

    i = m;
    j = m;

    do {
        i++;
        if ( A[i] > A[j] ) j = i;
    } while ( i != n );
    return j;
}
```

Wir formalisieren die Behauptung:

Precondition: $m < n$

$j = \text{findMax}(A, m, n)$

Postcondition: A, m, n werden nicht verändert und

$$\forall i \in \{m, \dots, n\} : A[i] \leq A[j].$$

Um die totale Korrektheit des Programms zu beweisen, müssen wir zeigen, daß das Programm terminiert, also die return-Anweisung erreicht wird und daß dann die Postcondition erfüllt ist.

Für den Beweis ist es nützlich, Zwischenbehauptungen in das Programm einzuführen.

Der Beweis ist wie ein Beweis eines mathematischen Satzes.

Wir beweisen folgende Behauptungen

1. Unmittelbar vor der do-Anweisung gilt $i = m$ und $j = m$.
2. Nach der Anweisung `i++` gilt im k -ten Durchlauf der Schleife $i = m + k$.
3. Nach der `if`-Anweisung in der Schleife gilt $\forall l \in \{m, \dots, i\} : A[j] \geq A[l]$.
4. Nach Abschluß der Schleife gilt $\forall l \in \{m, \dots, n\} : A[j] \geq A[l]$.

Das sieht im Programmtext so aus:

```

|  int findMax(int[] A, int m, int n) {
|
|
|  // precondition: m < n
5 |  // postcondition: returns position of largest item in subarray A[m:n]
|
|  int i;
|  int j;
|
10 |  i = m;
|  j = m;
|
|  {<> ( i == m) and (j == m) and (m < n) <>}
|
15 |  do {
|
|  i++ ;
|
|  if ( A[i] > A[j] ) j = i;
20 |
|  {<> in the kth iteration we have i = m+k}
|  {<> loop invariant: A[j] >= A[m:i] and (i <= n) <>}
|
|  } while ( i != n );
|
25 |  {<> final assertion: A[j] >= A[m:n] <>}

```

```

|
|   return j;           // return j as position of largest item in A[m:n]
|
| }

```

```

/*-----*/

```

Die zweite und dritte Behauptung sind *Schleifeninvarianten*. Sie gelten in jedem Durchlauf der Schleife und auch nach Verlassen der Schleife mit dem entsprechenden Wert für i ($i = n$).

Aus der zweiten Behauptung folgt die Terminierung, weil am Ende des $n - m$ -ten Durchlaufs die Abbruchbedingung $i = n$ erfüllt ist.

Aus der zweiten und dritten Behauptung folgt die vierte Behauptung und daraus folgt die Postcondition.

Wir haben also eine Beweisstrategie aufgebaut. Jetzt beweisen wir die einzelnen Hilfsbehauptungen:

Die erste Behauptung ist offensichtlich korrekt.

Die zweite und dritte Behauptung wird simultan durch Induktion bewiesen. Angenommen, wir sind im k -ten Durchlauf der Schleife. Ist $k = 1$, dann gilt $i = m + 1 = m + k$ und $A[i] = \max\{A[m], A[m + 1]\}$.

Angenommen, die Behauptung gilt für ein $k' < n$. Sei $k = k' + 1$. Weil i um eins erhöht wird, gilt nach Induktionsannahme am Schluss des k -ten Durchlaufs $i = m + k' + 1 = m + k$. Außerdem gilt am Anfang des k -ten Durchlaufs der Schleife nach Induktionsannahme $A[l] \leq A[s]$ für alle $s \in \{m, \dots, k' = k - 1\}$. Nach der Anweisung `i++` ist $i = k$. Die dritte Behauptung wird durch eine Fallunterscheidung bewiesen:

1. Ist $A[i] > A[j]$ so wird $j = i$ gesetzt. Damit gilt nach Induktionsannahme $A[j] \geq A[l]$ für alle $l \in \{m, \dots, i - 1\}$ Außerdem gilt $A[i] = A[i]$.
2. Ist $A[i] \leq A[j]$, so gilt nach Induktionsannahme $A[j] \geq A[l]$ für alle $l \in \{m, \dots, i\}$.

Damit ist die totale Korrektheit des Programms bewiesen.

10.3 Schnelle Exponentiation

Wir verifizieren noch ein weiteres Programm formal.

Wir beschreiben jetzt ein Verfahren zur schnellen Berechnung von Potenzen in einer endlichen Gruppe G . Dieses Verfahren und Varianten davon sind zentral in vielen kryptographischen Algorithmen. Sei $g \in G$ und e eine ganze Zahl. Sei

$$e = \sum_{i=0}^k b_i 2^i.$$

die Binärentwicklung von e . Man Beachte, daß die Koeffizienten b_i entweder 0 oder 1 sind. Dann gilt

$$g^e = g^{\sum_{i=0}^k b_i 2^i} = \prod_{i=0}^k (g^{2^i})^{b_i} = \prod_{0 \leq i \leq k, b_i=1} g^{2^i}.$$

Aus dieser Formel gewinnt man die folgende Idee:

1. Berechne die sukzessiven Quadrate g^{2^i} , $0 \leq i \leq k$.
2. Bestimme g^e als Produkt derjenigen g^{2^i} zu bestimmen, für die $b_i = 1$ ist.

Beachte dabei

$$g^{2^{i+1}} = (g^{2^i})^2.$$

daher kann $g^{2^{i+1}}$ aus g^{2^i} mittels einer Quadrierung berechnet werden. Bevor wir das Verfahren präzise beschreiben und analysieren, erläutern wir es an einem Beispiel. Es zeigt sich, daß der Algorithmus viel effizienter ist als die naive Multiplikationsmethode.

58. Example Wir berechnen g^{73} . Wir schreiben die Binärentwicklung des Exponenten auf:

$$73 = 1 + 2^3 + 2^6.$$

Wir berechnen die Quadrate $g, g^2, g^{2^2} = (g^2)^2, g^{2^3} = (g^{2^2})^2, g^{2^4} = (g^{2^3})^2, g^{2^5} = (g^{2^4})^2, g^{2^6} = (g^{2^5})^2$. Dann berechnen wir die Potenz

$$g^{73} = g * g^{2^3} * g^{2^6}.$$

Wir mussten also sechs mal quadrieren und zweimal multiplizieren und nicht 72 mal multiplizieren.

Um das Beispiel zu konkretisieren, beschreiben wir das modulo-Rechnen. Ist a eine ganze Zahl und m eine natürliche Zahl, dann ist $a \bmod m$ der Rest der Division von a durch m . Es ist also $10 \bmod 3 = 1$, weil $10 = 3 * 3 + 1$ ist, $-10 \bmod 3 = 2$, weil $-10 = -4 * 3 + 2$ ist. Für den Rest R gilt immer $0 \leq r < m$. Der Rest ist eindeutig bestimmt. Wenn $a \bmod m = b \bmod m$ gilt, dann schreibt man $a \equiv b \bmod m$. Aus $a \equiv b \bmod m$ und $c \equiv d \bmod m$ folgt $a \pm c \equiv b \pm d \bmod m$ und $a * c \equiv b * d \bmod m$.

59. Example Wir wollen $3^{73} \bmod 11$ berechnen. Wir schreiben die Binärentwicklung des Exponenten auf:

$$73 = 1 + 2^3 + 2^6.$$

Dann bestimmen wir die sukzessiven Quadrate $3, 3^2 \equiv 9 \bmod 11, 3^{2^2} \equiv 9^2 \equiv 4 \bmod 11, 3^{2^3} \equiv 4^2 \equiv 5 \bmod 11, 3^{2^4} \equiv 5^2 \equiv 3 \bmod 11, 3^{2^5} \equiv 3^2 \equiv 9 \bmod 11, 3^{2^6} \equiv 4 \bmod 11$. Also ist $3^{73} \equiv 3 * 3^{2^3} * 3^{2^6} \equiv 3 * 5 * 4 \equiv 5 \bmod 11$.

Hier kommt eine Implementierung des Algorithmus.

```
public GroupElement power(int e) {
    GroupElement result = returnOne();
    GroupElement multiplier = copy();

    while (e > 0) {
        if(e%2 == 1){
            result = result.multiply(multiplier);
            e = e-1;
        }
        e = e/2;
        multiplier = multiplier.square();
    }
    return result;
}
```

Sie berechnet g^e für ein Gruppenelement g . Die Methode ist aber eine Variante des ursprünglichen Algorithmus, weil man sich die Speicherung der sukzessiven Quadrate sparen will. Das aktuelle Quadrat steht immer in der Variablen `multiplier`.

Precondition: g ist ein Gruppenelement. $e \geq 0$.

`g.power(e)`

Postcondition: $result = g^e$

Vor der `while`-Schleife ist `result == 1, multiplier == g`.

Jetzt kommt die Schleifeninvariante. Sie zu finden, ist das wichtigste und oft das schwierigste. Sei

$$e = \sum_{i=0}^{n-1} b_i * 2^i.$$

und für $k \in \{0, \dots, n-1\}$ sei

$$f_k = \sum_{i=0}^{k-1} b_i 2^{i-k}, \quad e_k = \sum_{i=k}^{n-1} b_i * 2^{i-k}$$

Am Ende der while-Schleife ist nach dem k -ten Durchgang

$$e = e_k$$

(für $k = n$ ist das 0) und

$$result = g^{f_k}$$

und

$$multiplier = g^{2^k}$$

$k = 1$. Ist *eungerade* so ist $b_0 = 1$. Also wird in diesem Fall $result = result * multiplier = 1 * g = g = g^{b_0}$ ausgerechnet. Außerdem ist das neue e gleich $(e-1)/2 = (\sum_{i=1}^n b_i 2^i)/2 = \sum_{i=1}^n b_i 2^{i-1}$, wie behauptet. Ist aber e gerade, dann ist $b_0 = 0$. Also wird in diesem Fall $result$ nicht verändert. Tatsächlich ist $result = g^{b_0} = 1$. Außerdem ist das neue e gleich $e/2 = (\sum_{i=1}^n b_i 2^i)/2 = \sum_{i=1}^n b_i 2^{i-1}$, wie behauptet. Schließlich ist $multiplier = g^2$.

$k \rightarrow k + 1$. Sei $k < n$.

Es ist zu Anfang der Schleife $e = \sum_{i=k}^n b_i 2^i$ und $result = g^{\sum_{i=0}^{k-1} b_i 2^i}$. Das folgt aus der Induktionsannahme. Ist *eungerade* so ist $b_k = 1$. Also wird in diesem Fall

$$result = result * multiplier = g^{\sum_{i=0}^{k-1} b_i 2^i} * g^{2^k} = g^{\sum_{i=0}^k b_i 2^i}$$

ausgerechnet. Außerdem ist das neue e gleich

$$(e-1)/2 = (\sum_{i=k+1}^n b_i 2^{i-k})/2 = \sum_{i=k+1}^n b_i 2^{i-k-1},$$

wie behauptet. Ist aber e gerade, dann ist $b_k = 0$. Also wird in diesem Fall $result$ nicht verändert. Tatsächlich ist dann

$$result = g^{\sum_{i=0}^{k-1} b_i 2^i} = g^{\sum_{i=0}^k b_i 2^i}.$$

Außerdem ist das neue e gleich

$$e/2 = (\sum_{i=k+1}^n b_i 2^{i-k})/2 = \sum_{i=k+1}^n b_i 2^{i-k-1},$$

wie behauptet. Schließlich ist der neue multiplier das Quadrat des alten und damit

$$(g^{2^k})^2 = g^{2*2^k} = g^{2^{k+1}}.$$

Die Terminierung des Programms folgt daraus, daß schließlich $e = 0$ ist.

10.4 Selbstüberprüfung von Methoden

Anstatt formal zu verifizieren, daß eine Methode das richtige macht, kann man oft an den Schluß einer Routine einen Test schreiben, der bestätigt, daß das Ergebnis richtig ist.

Wir besprechen ein paar Beispiele.

Der Algorithmus zur Lösung der Pell-Gleichung muß

10.5 Übungen

1. Verifizieren Sie einen (vorgegebenen) ggt-Algorithmus. Schreiben Sie dazu geeignete Preconditions und Postconditions auf und finden Sie geeignete Schleifeninvarianten.
2. Verifizieren Sie Merge-Sort. Schreiben Sie dazu geeignete Preconditions und Postconditions auf und finden Sie geeignete Schleifeninvarianten.
3. Sei n eine natürliche Zahl, die ein Quadrat einer anderen natürlichen Zahl ist. Angenommen, n hat Bitlänge k .
 - a) Zeigen Sie, daß $m = \lceil k/2 \rceil$ die binäre Länge der Quadratwurzel von n ist.
 - b) Entwerfen Sie einen Algorithmus, der die Quadratwurzel von n Bit für Bit berechnet. Dazu soll eine Variable *result* verwendet werden. Am Anfang ist $result = 2^{m-1}$. Dann wird eine Schleife für $i = m - 2, m - 3, \dots, 0$ durchlaufen. In der Schleife wird immer geprüft, ob $(result + 2^i)^2$ größer als n ist. Wenn nicht, wird *result* auf $result + 2^i$ gesetzt.
 - c) Implementieren Sie den Algorithmus in Java.
 - d) Formulieren Sie Pre- and Postconditions für den Algorithmus, die die Berechnungsaufgabe widerspiegeln.
 - e) Formulieren Sie geeignete Schleifeninvarianten und beweisen Sie die totale Korrektheit des Algorithmus.
 - f) Geben Sie ein einfaches Verifikationsprogramm für den Algorithmus an.
 - g) Zeigen Sie, daß der Algorithmus Zeit $O(\log n)$ braucht.

Kapitel 11

Objektorientiertes Design

11.1 Die Idee

Verwendet man objektorientierte Programmierung, hat man nicht zuerst die Probleme im Auge, die man lösen will, sondern man modelliert den Weltausschnitt, in dem die Probleme entstehen und gelöst werden sollen.

Die zentrale Frage ist also nicht, was das System machen soll, sondern womit es etwas machen soll oder genauer, mit welchen Objekten etwas gemacht werden soll.

Wenn man also einen unvollständig geklammerten Term auswerten will, dann stellt man fest, daß man Terme braucht und Strings und bei der Verwendung von Kellerautomaten auch Stacks usw. (Natürlich hat man auch eine Lösungsmethode im Kopf, aber vorerst interessiert man sich nur für die Objekte, die in der Lösungsmethode vorkommen).

Der Vorteil liegt auf der Hand. Treten in demselben oder einem ähnlichen Bereich andere Probleme auf, dann kann man Objekte wiederverwenden. Man kann das System auch leichter erweitern, indem man einfach neue Objekte hinzufügt.

Das Verfahren ist Bottom-Up und nicht Top-Down. Man überlegt sich nicht erst, was hinterher mal gemacht werden soll, sondern man überlegt sich, was die Ingredienzien sind.

11.2 Objektorientiertes Design

Mit den Erklärungen des vorigen Abschnitts kann man jetzt das objektorientierte Design genauer beschreiben.

Es handelt sich um die Konstruktion von Software-Systemen als eine strukturierte Ansammlung von Implementierungen abstrakter Datentypen.

Die Implementierungen der abstrakten Datentypen heißen Klassen. Es

handelt sich um eine Ansammlung. Die einzelnen Teile können in anderen Zusammenhängen wiederverwendet werden. Die Struktur kommt aus dem Verhältnis der Klassen zueinander. Klassen können einander benutzen. So kann die Stack-Klasse die Klasse LinkedList benutzen. Oder Klassen können voneinander abgeleitet sein (Vererbung). Das wird später noch genauer besprochen.

Wir können jetzt noch folgende Leitsätze für objektorientierte Glückseligkeit formulieren:

1. (Objektbasierte Modularisierung) Systeme werden auf der Basis ihrer Datenstrukturen (und nicht ihrer Funktionen) modularisiert.
2. (Datenabstraktion) Objekte sollen als Implementierungen abstrakter Datentypen beschrieben werden.
3. (Automatische Speicherverwaltung) Nicht benutzte Objekte werden automatisch vernichtet.
4. (Klassen) Moduln sind Typen und nicht einfache Typen sind Moduln. Die Moduln heißen Klassen.
5. (Vererbung) Klassen können als Erweiterung bzw. Spezialisierung anderer Klassen definiert werden.
6. (Polymorphie und dynamische Bindung) Programmeinheiten (Variablen) können sich auf Objekte von mehreren Klassen beziehen. Operationen können unterschiedliche Bedeutungen in unterschiedlichen Klassen haben.
7. (Mehrfachvererbung) Klassen können von mehreren verschiedenen Klassen erben.

11.3 Abstrakte Datentypen und Klassen

Wie charakterisiert man die Objekte, die im System vorkommen sollen? Man verwendet dazu die Beschreibung abstrakter Datentypen (da gibt es eine umfangreiche Theorie). Für einen abstrakten Datentyp muß folgendes festgelegt werden:

1. Wie heißt der Typ?
2. Welche partiellen Funktionen für diesen Typ werden zur Verfügung gestellt?

3. Was sind die Definitionsbereiche dieser Funktionen?
4. Was machen diese Funktionen?

In der Theorie der abstrakten Datentypen gibt es eine formal-logische Methode, diese Fragen zu beantworten. Das hat den Vorteil, daß man dann formal Aussagen über die Datentypen beweisen kann.

Wir geben ein Beispiel.

Der Typ heißt $Stack[X]$, wobei die Menge der Elemente, die in dem Stack sein dürfen, mit X bezeichnet wird. (Es handelt sich also in Wirklichkeit um einen durch X parametrisierten Datentyp).

Die Funktionen sind

$$\begin{array}{l}
 \text{empty} \quad : \quad STACK[X] \rightarrow \text{boolean} \\
 \text{new} : \rightarrow STACK[X] \\
 \text{push} : X \times STACK[X] \rightarrow STACK[X] \\
 \text{pop} : STACK[X] \rightarrow STACK[X] \\
 \text{top} : STACK[X] \rightarrow X
 \end{array}$$

Die Definitionsbereiche sind:

für pop und top darf der Stack nicht leer sein.

Die Wirkung der Funktionen kann axiomatisch beschrieben werden.

Für alle $x \in X$ und alle Stacks s über X gilt folgendes

1. Ein neuer Stack ist leer, d.h. $\text{empty}(\text{new}()) = \text{true}$.
2. Ein leerer Stack, auf den etwas gepusht wird, ist nicht mehr leer, d.h. $\text{empty}(\text{push}(x, s)) = \text{false}$.
3. Wird x gepusht, so steht x oben, d.h. $\text{top}(\text{push}(x, s)) = x$.
4. Wird x gepusht, so steht x oben, d.h. $\text{pop}(\text{push}(x, s)) = x$.

Es ist aber nicht so einfach zu sehen, daß die axiomatische Charakterisierung wirklich das beschreibt, was man will.

Die Funktionen werden die öffentlichen Methoden der entsprechenden Klasse. Die Vorbedingungen und Axiome werden zu Vor- und Nachbedingungen, die in der Verifikation benutzt werden können.

Die Implementierung eines abstrakten Datentyps ist ein Programm, das die spezifizierten Dienste (Funktionen) zur Verfügung stellt.

11.4 Vererbung

Ist eine Klasse eine Erweiterung (Nachfahre) einer anderen Klasse, so werden alle Variablen und Methoden und (falls vorhanden) der NoArgs-Konstruktor vererbt. Die Methoden können also jetzt auch auf Objekte vom abgeleiteten Typ angewendet werden. Die Methoden können aber auch (mit derselben Signatur) überschrieben (neuimplementiert) werden.

Variablen von einem Typ können Objekte von jedem abgeleiteten Typ zugewiesen werden. Wird eine Methode auf das durch die Variable dargestellte Objekt angewendet, dann wird immer die Version der Methode genommen, die dem Objekt im Ableitungsbaum am nächsten ist.

Wir diskutieren die Vorteile, die die Vererbung mit sich bringt.

Wir wissen ja bereits, daß in der objektorientierten Softwareentwicklung Typen und Moduln identifiziert werden. Wir betrachten also zuerst die Auswirkung auf Moduln. Moduln sind abgeschlossene Programmeinheiten, die mit einem Benutzer Dienste mittels definierter Schnittstellen zur Verfügung stellen. Moduln müssen abgeschlossen sein, damit der Benutzer sich auf die Services verlassen kann. Aber andererseits ist im Systementwurf nicht von vornherein klar, welche Dienste der Benutzer braucht. Das kann sich dynamisch entwickeln. Daher müssen Moduln andererseits auch erweiterbar sein. Weitere Services müssen hinzugefügt werden können, ohne daß die alten alle neu implementiert werden müssen aber auch ohne daß die Services der bestehenden Moduln geändert werden können. Trotzdem müssen vielleicht auch einige Dienste in Erweiterungsmoduln geändert werden. Dies alles leistet die OO. Sie erlaubt die Erweiterung von Moduln unter Beibehaltung der schon vorhandenen Dienste und die Neuimplementierung einiger Dienste.

Wir betrachten Klassen als nächstes als Implementierungen abstrakter Datentypen. Will man einen neuen Typ implementieren, dessen Instanzenmenge Teilmenge der Instanzenmenge eines anderen Typs ist (Rechtecke sind Polygone), dann kann man einen solchen Typ durch Vererbung realisieren. Einzelne Methoden im Obertyp können in den abgeleiteten Typen andere Bedeutung haben. Es ist sogar möglich, die Methode im Obertyp abstrakt zu lassen, weil erst in den abgeleiteten Typen klar wird, wie sie benutzt werden sollen.

11.5 Mehrfachvererbung

Wir geben eine objektorientierte Modellierung einer einfachen mathematischen Situation an. Wir interessieren uns für die Begriffe Halbgruppe, Monoid, Gruppe, Ring, Körper.

Die Objekte, mit denen die Programme arbeiten, sind im allgemeinen Elemente dieser Mengen. Da wir Mehrfachvererbung erlauben wollen, benutzen wir Interfaces. Wir nennen sie `SemiGroupElement`, `GroupElement`, `MonoidElement`, `RingElement`, `FieldElement`.

Literaturverzeichnis

- [MSS96] S. Middendorf, R. Singer, and S. Strobel. *JAVA Programmierhandbuch und Referenz*. dpunkt.verlag, 1996.
- [Sta98] T. Standish. *Data structures in JAVA*. Addison-Wesley, 1998.