

UNIVERSITÄT AUGSBURG

FAKULTÄT FÜR ANGEWANDTE INFORMATIK
INSTITUTE FOR SOFTWARE & SYSTEMS ENGINEERING



MASTERARBEIT
VON
ANTONIO GRIECO

**Programmiersprachliche Konzepte von COBOL im
Vergleich mit Java – Eine praxisorientierte
Einführung**

Erstgutachter: Prof. Dr. Alexander Knapp

Zweitgutachter: Prof. Dr. Bernhard Bauer

Betreuer: Prof. Dr. Alexander Knapp & Jonathan Streit

Abgabedatum: 6. Mai 2018

Abstract

Zusammenfassung

Inhaltsverzeichnis

Abstract	I
Zusammenfassung	II
Inhaltsverzeichnis	III
Abbildungsverzeichnis	V
Listings	VI
1 Aus alt mach neu – COBOL im gefährlichen Alter	1
1.1 Problemstellung	1
1.2 Ziel der Arbeit	4
1.3 Aufbau der Arbeit	5
2 Methodik der Arbeit	6
2.1 Vorhandene Literatur	6
2.2 Experten-Interviews	7
2.3 Entwicklungsumgebungen	8
3 Herausforderungen für COBOL und Java in betrieblichen Informationssystemen	11
3.1 Datenmengen und Dimensionierung	11
3.2 Langlebigkeit, Wartbarkeit und Verlässlichkeit	11
3.3 Modularisierung, Wiederverwendbarkeit und Variabilität	11
3.4 Darstellungsgenauigkeit – Fließ- und Festkommaarithmetik	14
3.5 Schnittstellen und Datenquellen	18
3.6 Reporting	19
4 Vergleich wichtiger Sprachkonzepte	20
4.1 Programmstruktur	20
4.2 Variablen und Datentypen	26
4.3 Arrays	31
4.4 Programmablauf und Kontrollfluß	33

4.4.1	Ablauf	33
4.4.2	Verzweigungen	37
4.4.3	Schleifen	39
4.4.4	Weitere Schlüsselwörter	41
4.4.5	Ausnahmebehandlung	44
4.4.6	Nebenläufigkeit	45
4.5	Funktionen, Unterprogramme und Rückgabewerte	46
4.6	Dateien	51
4.7	Generische Programmierung	54
4.8	Konventionen	55
4.8.1	Groß- und Kleinschreibung	56
4.8.2	Affixe	56
4.8.3	Schlüsselwörter	58
4.8.4	Formatierung des Quelltextes	58
4.9	Weitere Sprachkonzepte	58
4.9.1	Benannte Bedingungen	58
4.9.2	Mehrfachverzweigungen	62
4.9.3	Speicherausrichtung	66
4.9.4	Reorganisation von Daten	67
4.9.5	Implizierte Variablennamen	68
4.9.6	Modifier	71
5	Typische Pattern in COBOL und Java	72
5.1	Komplexe Datenstrukturen	72
5.1.1	Listen	72
5.1.2	Sets	75
5.1.3	Maps	78
5.1.4	Verbunddatenstrukturen	78
5.2	Entwurfsmuster	81
5.2.1	Callback-Muster	82
5.2.2	Singleton-Muster	86
5.2.3	Dependency Injection	87
5.3	Redundanzen durch Wertekopien	89
5.4	Externe Deklaration von Daten	90
6	Fazit (Sprechender Name je nach Ergebnissen)	94
Literatur		VIII

Abbildungsverzeichnis

4.1	Strukturelle Bestandteile eines Java-Programms	20
4.2	Strukturelle Bestandteile eines COBOL-Programms	23
4.3	UML-Diagramm einer Aggregation	68

Listings

2.1	Erstellen eines neuen COBOL Programms	9
2.2	Erstes COBOL-Programm in der Kommandozeile	10
3.1	Ungenauigkeit am Beispiel einer float-Variable	15
3.2	Dezimalzahlen in COBOL	17
4.1	Initializer in Java	22
4.2	Anonyme Klassen und Funktion in Java	23
4.3	Variablendeklarationen in Java	27
4.4	Variablendeklarationen mit verschiedenen Scopes	28
4.5	Variablendeklarationen in COBOL	29
4.6	Felder in Java	32
4.7	Felder in COBOL	32
4.8	Java main-Methode	34
4.9	Programmablauf in COBOL	35
4.10	Programmablaufunterschiede in COBOL mit Sections und Paragraphs	36
4.11	Verzweigung in Java	38
4.12	Verzweigung in COBOL	38
4.13	Schleifen in Java	40
4.14	Schleifen in COBOL	41
4.15	Beispiele für die Verwendung von break und continue in Java	42
4.16	EXIT PERFORM in COBOL	43
4.17	Rudimentäre Fehlerbehandlung in COBOL	45
4.18	Methoden in Java	46
4.19	Rekursion in Java	48
4.20	Rekursion in COBOL	49
4.21	Unterprogramme in COBOL	50
4.22	Datei-Ein- und Ausgabe in Java [18]	52
4.23	Eingabedatei recordFile.txt	53
4.24	x	53
4.25	Datei-Ausgabe in COBOL [18]	54
4.26	Generics in Java	54

4.27	Beispiel für COBOL Stufennummer 88	59
4.28	Setzen von Werten mithilfe benannter Bedingungen	60
4.29	Bedingte Werte in Java	61
4.30	Setzen eines konstanten Wertes mit einem Enum in Java	62
4.31	Mehrfachverzweigungen in Java	63
4.32	Mehrfachverzweigungen in COBOL mit ALSO	65
4.33	Mehrfachverzweigungen in COBOL als EVALUATE TRUE	66
4.34	Stufennummer 66 und RENAMES -Befehl	67
4.35	Keine implizierten Variablennamen in logischen Ausdrücken in Java	69
4.36	Verwendung von Variablennamen in logischen Ausdrücken in Java	70
4.37	Implizierte Variablennamen in COBOL	70
5.1	ArrayList Beispiel in Java	73
5.2	Einfache Listen Implementierung in COBOL	74
5.3	HashSet Beispiel in Java	76
5.4	Einfache Set Implementierung in COBOL	78
5.5	Structs und Unions in COBOL	80
5.6	Java Bean	81
5.7	Observer und Observable in Java	83
5.8	Listener in Java	85
5.9	Singleton in Java	86
5.10	Dependency Injection in Java	88
5.11	Import in Java	90
5.12	COBOL-Copybook Datei (COPYBOOK.cpy)	91
5.13	Nutzung eines COBOL-Copybook	91
5.14	COBOL-Copybook Datei (COPYBOOK-EVALUATE.cpy)	92
5.15	Nutzung von COPYBOOK-EVALUATE.cpy	92

1 Aus alt mach neu – COBOL im gefährlichen Alter

1.1 Problemstellung

Der folgende Abschnitt soll die Problemstellung verdeutlichen, welche der Arbeit zugrunde liegt. Dazu wird erläutert, welche Wichtigkeit COBOL innehat und anschließend mit der Bedeutung, die der Sprache heutzutage in der Lehre tatsächlich beigemessen wird gegenübergestellt. Anhand dessen werden Schwierigkeiten für den Arbeitsmarkt analysiert und beschrieben.

Wichtigkeit von COBOL

»Viele Millionen Cobol-Programme existieren weltweit und müssen laufend gepflegt werden. Es ist bei dieser Situation undenkbar und unter wirtschaftlichen Gesichtspunkten unvertretbar in den nächsten Jahren eine Umstellung dieser Programme auf eine andere Sprache durchzuführen.« [13]

Was Herr Dr. Strunz neben vielen anderen Experten bereits 1979 prophezeite, hat auch heute noch Gültigkeit. Obwohl COBOL zum Ende der 50er Jahre entstand, 1959 veröffentlicht wurde und damit fast 60 Jahre alt ist, trifft man es auch heute in betrieblichen Informationssystemen noch häufig an. In der britischen Tageszeitung *The Guardian*, zitiert der Autor Scott Colvey in seinem Artikel [5] anlässlich des 50. Geburtstages von COBOL den Micro Focus Manager David Stephenson: »‘some 70% to 80% of UK plc business transactions are still based on Cobol’«. Weiter führt er darin Aussagen von IBM Software-Leiter Charles Chu an, welcher die Aussagen von Stephenson bestätigt: »[...] there are 250bn lines of Cobol code working well worldwide. Why would companies replace systems that are working well?«. Stephen Kelly, Geschäftsführer von Micro Focus, betont zudem, dass sich Stand 2009 über 220 Milliarden COBOL-Codezeilen im

produktiven Einsatz befanden, welche vermutlich 80% der insgesamt weltweit aktiven Codezeilen ausmachten. Außerdem wurden zum damaligen Zeitpunkt, Schätzungen zufolge 200-mal mehr COBOL-Transaktionen ausgeführt als Google Suchanfragen verzeichnen konnte. [15] Diese Aussagen decken sich mit den Angaben in *COBOL Programmers Swing with Java* [8]. Auch darin betonen Doke u. a., dass – Stand 2005 – mit 225 Milliarden Codezeilen, etwa 70% des weltweiten Codes in COBOL geschrieben sind.

Daran wird nicht nur deutlich, dass COBOL in den vergangenen Jahren einen enormen Marktanteil ausmachte, sondern auch die weitere Bedeutung der Sprache für die Zukunft: Wieso sollte funktionierender Code mit Hilfe von teuren und riskanten Prozessen ersetzt werden? Da sich viele Unternehmen dieser Frage eines Umstiegs von COBOL auf eine modernere Lösung ausgesetzt sehen, auf die sich nur schwer eine Antwort finden lässt, welche die Risiken und Kosten aufwiegt, stieg die Zahl des weltweit betriebenen COBOL-Codes über die vergangenen Jahre sogar noch weiter an. Dieses Risiko ergibt sich vorrangig durch Transaktionen immenser Geldsummen, die mit COBOL-Systemen durchgeführt werden. Denn »Täglich werden Transaktionen mit einem Volumen von schätzungsweise drei Billionen Dollar über Cobol-Systeme abgewickelt. Dabei geht es um Girokonten, Kartennetze, Geldautomaten und die Abwicklung von Immobilienkrediten. Weil die Banken aggressiv auf eine Digitalisierung ihres Geschäftes setzen, wird Cobol sogar noch wichtiger. Denn Apps für Smartphones etwa sind in modernen Sprachen geschrieben, müssen aber mit den alten Systemen harmonieren.« [2]

Im TIOBE-Index [26] für April 2018 rangiert COBOL auf Platz 25 mit einem Rating von 0.541%. Dieser Index wird auf Basis von Suchanfragen nach den entsprechenden Programmiersprachen, auf den meist frequentiertesten Internetseiten, erstellt. COBOL ist somit zwar nur Teil jeder 200. Suchanfrage, rangiert jedoch damit trotzdem vor anderen etablierten oder aufstrebenden Sprachen wie *Kotlin*, *Scala* oder *Haskell*. Außerdem gilt es hier zu beachten, dass COBOL zu einer Zeit entstand, in der das Internet noch lange nicht existierte und Informationen über die Sprache mittels Büchern verbreitet und vermittelt wurden. Daher ist auch heute noch das Internet nicht die vorrangige Quelle, um Wissen über COBOL zu akquirieren. Unter diesen Gesichtspunkten ist das TIOBE-Rating von COBOL als noch höher einzuschätzen.

Bedeutung in der Lehre

Da COBOL bereits 60 Jahre alt ist, haben heutzutage bereits viele einstige COBOL-Entwickler das Rentenalter erreicht. Im Artikel *Cobol-Programmierer gesucht* [2] beschreibt der Autor exemplarisch den Fall eines 75-Jährigen Entwickler, der wegen seiner Erfahrung trotz seines Alters immer noch in der Branche tätig ist.

Junge COBOL-Entwickler sind rar, da COBOL nur noch selten Teil der Ausbildung ist. Doke u. a. führen in *COBOL Programmers Swing with Java* [8] an, dass im Jahr 2002 lediglich für 36.2% der Studenten COBOL Teil des Grundstudiums war, obwohl im Jahr 1995 noch 89.7% der befragten Bildungseinrichtungen angaben COBOL-Kurse als festen Bestandteil der Ausbildung zu haben. Sieht man sich dagegen die Zahlen zu Java als Vertreter moderner Programmiersprachen an, lässt sich ein klarer Trend erkennen. Erst 1995 entstanden, stieg die Zahl der Universitäten, die Java lehrten von 42.5% im Jahr 1998 auf 90.0% im Jahr 2002. Spinnt man diesen Wandel, zu dem sich in der Zwischenzeit noch eine Fülle neuerer Sprachen hinzugesellt hat, ins heutige Jahr weiter, lässt sich erahnen, wie selten Lehrveranstaltung zum Thema COBOL inzwischen geworden sind.

Man sieht also, dass sich die Lehre, obwohl der Bedarf an COBOL-Programmierern weiterhin immens ist, der Fokus der Ausbildung bei anderen Programmiersprachen liegt, was die Wirtschaft zusammen mit dem zunehmenden Alter erfahrener COBOL-Entwickler vor Probleme beim Stillen der Nachfrage an Arbeitskräften stellt.

Kontroverse Beurteilungen von COBOL

Die in Abschnitt 1.1 angeführten Aussagen und Meinungen stammen oftmals von Personen aus dem Umfeld von Unternehmen, die teils stark vom Weiterbestehen COBOLs profitieren. Diese Aussagen sind daher, wenn auch sicherlich nicht falsch, vorsichtig und vor allem sehr differenziert zu betrachten.

Der mehrfach prämierte Informatiker Edsger Wybe Dijkstra z.B. findet sehr klare, andere Worte zu COBOL: »The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.« [9]

Florian Hamann nennt in seinem Artikel *In Banken Leben Dinosaurier* [10] die bereits erwähnte zunehmende Knappheit von Arbeitskräften auch als einen wichtigen Faktor

dafür, weshalb COBOL über kurz oder lang von moderneren Systemen und Sprachen verdrängt und abgelöst wird.

Trotz dieser Kontroversen kann festgehalten werden, dass es nach wie vor einen gleichbleibend hohen Bedarf an Entwicklern gibt, den es zu decken gilt. Allerdings entstand die Sprache weit vor wichtigen Entwicklungen und Innovationen in der Informatik und bildet so keine zeitgemäße Grundlage für umfangreiche und noch weniger für neue Systeme.

1.2 Ziel der Arbeit

Die vorliegende Arbeit soll einen Beitrag zur Lösung der in Abschnitt 1.1 beschriebenen Probleme leisten. Dies soll mit Hilfe eines Leitfadens geschehen, der fachkundigen Java-Entwicklern den Einstieg in COBOL erleichtert, indem gängige Sprachkonzepte gegenübergestellt und verglichen werden.

So soll es möglich sein, vorhandenes Wissen über die Softwareentwicklung, im speziellen mit Java, in einen COBOL-Kontext zu bringen und passende Sprachkonzepte nutzen zu lernen. Des Weiteren soll aufgezeigt werden, welche konzeptuellen Herausforderungen sich bei der COBOL-Entwicklung und -Migration ergeben.

Im Fokus steht hierbei neben der Einführung in relevante Sprachkonstrukte stets auch die Experteneinschätzung zur Nutzung der verschiedenen Konzepte. Daher wird, wenn möglich, zusätzlich zu den erklärten Paradigmen erläutert, wie die Verwendung in der Praxis aussehen bzw. nicht aussehen sollte und je nach Sprachmittel gegebenenfalls in der Praxis zu verwendende Alternativen aufgezeigt.

Ziel der Arbeit ist jedoch nicht, vorhandene Java-Entwickler zu Neuentwicklungen mit COBOL zu animieren oder diese gar zu COBOL-Entwicklern umzuschulen. Wichtig ist in diesem Zusammenhang vielmehr ihr Wissensspektrum so zu erweitern, dass es ihnen möglich wird, komplexe fachliche Zusammenhänge – vor allem die »business logic« – bestehender COBOL-Architekturen zu erkennen und zu verstehen. Dadurch sind diese Entwickler fortan flexibler einsetzbar und geschult, um mit Migrations-, Renovierungs- und Wartungsaufgaben von COBOL-Systemen betraut werden zu können.

1.3 Aufbau der Arbeit

Um leichter Verständnis über die vorliegende Arbeit zu erlangen, wird an dieser Stelle vorweg kurz auf den Aufbau eingegangen.

In Kapitel 1 wird die Motivation zum Bearbeiten des Problems und das damit verfolgte Ziel erklärt.

Die vorhandene Literatur und das Vorgehen bei der Erstellung der Arbeit werden in Kapitel 2 erläutert.

Kapitel 3 behandelt die grundlegenden Herausforderungen bei der Entwicklung von betrieblichen Informationssystemen und zeigt wie sich diese Problemstellungen in COBOL und Java adressieren lassen.

Die wichtigsten Sprachmittel und Konzepte werden in Kapitel 4 aufgezeigt und gegenübergestellt.

Kapitel 5 veranschaulicht wichtige und häufig auftretende Muster der Sprachen und zeigt wie und ob diese in der jeweils anderen abgebildet werden können.

Das Kapitel 6 beinhaltet eine Zusammenfassung und Interpretation der Thematik und gibt ein Resümee der Arbeit.

2 Methodik der Arbeit

2.1 Vorhandene Literatur

Die aufgeführte Literatur gibt oftmals einen sehr detaillierten Einblick in COBOL und bietet Hilfestellungen mit »Nachschlage-Charakter«. So wie beispielsweise in *Teach Yourself Cobol in 21 Days* [3], *COBOL Programming - Tutorials, Lectures, Exercises, Examples* [27] oder *COBOL Programming Fundamental* [14] werden häufig möglichst viele der vorhandenen COBOL-Konstrukte vorgestellt, mit Beispielen beschrieben und so ihre Verwendung gezeigt.

Neuere Literatur wie *COBOL for the 21st Century* [23] betrachten dabei häufig zusätzlich Neuerungen wie die objektorientierte Verwendung von COBOL. *Cobol 2002 ge-packt* [21] hingegen stellt mehr ein Syntax-Wörterbuch dar, als eine wirkliche Beschreibung oder Einführung in COBOL.

Beginning COBOL for Programmers [6] bietet den wohl umfassendsten Überblick, sowie ausführliche Beispiele und Erklärungen zur Verwendung und wirkt dabei nicht wie ein klassisches Nachschlagewerk sondern wie ein klar strukturiertes Fachbuch, das jedoch mit einem klaren roten Faden durch die Bestandteile von COBOL führt. Dabei zieht es vor allem in der Einführung auch an einigen, wenigen Stellen Parallelen zu Java.

Alle diese Werke setzen ein gewisses generelles Vorwissen im Bereich der Programmierung und Informatik voraus, was auch in dieser Arbeit der Fall sein soll. Jedoch ist an nur wenigen Stellen ein vergleichender Charakter zu anderen Sprachen zu erkennen und sehr selten die Erwähnung der jeweiligen Praxisrelevanz oder der besten Einsatzmöglichkeiten entsprechender Konstrukte zu finden.

Diese Arbeit soll nicht die vielfältig bestehende Literatur um ein weiteres ähnliches Werk ergänzen, sondern wichtige Bestandteile der Sprachen gegenüberstellen und Vorgehensweisen bei der Entwicklung aufzeigen. Dabei wird bewusst nur selten in die Tiefe der einzelnen Bestandteile eingegangen und alle möglichen Verwendungsarten

beschrieben, sondern versucht praktisch relevante Aspekte zu beleuchten. Für einen tieferen Einblick in die gesamten Sprachfeinheiten bietet sich die bereits erwähnte Literatur an, welche auch bei der Erstellung der Inhalte als Informationsquelle genutzt wurde.

COBOL Programmers Swing with Java [8] basiert auf den gleichen Ideen. Jedoch versucht dieses Buch Personen mit fundierten COBOL-Kenntnissen die Entwicklung in Java beizubringen indem Konzepte gegenübergestellt werden. Allerdings werden teilweise wichtige Details nicht erwähnt oder an manchen Stellen sogar falsch beschrieben, weshalb dieses Buch aus fachlicher Sicht zwar als eine gute Brücke von COBOL zu Java einzuschätzen ist, jedoch nicht als einzige Quelle dienen sollte. Den Brückenschlag in die andere Richtung – von Java zu COBOL – lässt es allerdings nicht ohne weiteres zu, was nicht zuletzt daran liegt, dass – wie der Name bereits andeutet – ein großer Teil des Buches grafischen Oberflächen mit *Swing* gewidmet ist.

Im Gegensatz zu vorherigem steigt *Java for COBOL Programmers* [4] tiefer in die syntaktischen Konstrukte von Java ein und behandelt außerdem Java EE (Enterprise Edition). Außerdem zielt es weniger auf grafische Systeme ab, als auf Systeme die Daten verarbeiten. So werden verschiedene Ein- und Ausgabe-Mechanismen erklärt, der Umgang mit XML-Formaten beschrieben und – als Teil der Java EE – die Verwendung von Datenbanken erläutert.

COBOL Programmers Swing with Java [8] und *Java for COBOL Programmers* [4] ermöglichen also COBOL-Entwicklern den schnellen Einstieg in Java und bieten durch die jeweils unterschiedlichen Schwerpunkte einen guten Überblick. Nicht nur, dass in diesen beiden Quellen die Sicht – Java lernen als COBOL-Entwickler – eine andere als in dieser Arbeit ist, auch wird nur selten bzw. gar nicht auf die praktische Relevanz der beschriebenen Konstrukte eingegangen. Dadurch fehlt der Charakter eines Leitfadens, welcher diese Arbeit prägen soll.

2.2 Experten-Interviews

Die vorliegende Arbeit soll vorhandene Expertise von Experten nutzen, um statt einem Nachschlagewerk für syntaktische Zwecke einen Leitfaden zu erarbeiten – der von praktischer Relevanz getrieben – die wichtigsten Eigenschaften von COBOL und Java beleuchtet und gegenüberstellt. Daher stellen neben den angesprochenen literarischen Quellen, vor allem Experteninterviews einen Kernpunkt dieser Arbeit dar. Um den

Praxisbezug zu gewährleisten, wurden diese geführt, transkribiert und darauf aufbauend relevante Themenbereiche und Praktiken ausgemacht und analysiert.

Interviewt wurden Experten aus dem Hause der **itestra GmbH**. Diese »Mitarbeiter kombinieren eine exzellente Informatik-Ausbildung mit Branchen-Know-how«, »kennen sowohl Legacy-Technologien wie Assembler, RPG und COBOL als auch Java, JS, C# und iOS« und »verstehen alte Systeme und setzen moderne Technologien ein«.¹ Die Mitglieder der Entwicklerteams können dabei also auf mehrjährige Erfahrungen im Bereich der Renovierung und dem Reengineering von COBOL-Systemen blicken, was die befragten Personen zur wohl wichtigsten Quelle dieser Arbeit macht.

Befragt wurden die drei kundigen COBOL-Entwickler *Ivaylo Bonev*, *Jonathan Streit* und *Thomas Lamperstorfer*. Dabei ging es nicht darum eine repräsentative Stichprobe nach statistischem Vorgehen zu erheben, sondern darum eine individuelle Bewertung der Schwierigkeiten und Stolpersteine bei der Entwicklung, Wartung und dem Verständnis von bestehenden und neuen COBOL-Systemen, sowie eine Einschätzung zu Parallelen und Diskrepanzen mit Java zu erhalten. Daher wurde kein Fragenkatalog ausgearbeitet, sondern offener Input der erwähnten Personen gefordert, um die gewünschten subjektiven Meinungen zu bekommen.

Der Umfang der Interviews beläuft sich auf 3 Stunden Audiomaterial bzw. 30 A4-Seiten Transkription in der ersten Phase und zusätzlichem Feedback in der Zwischenphase, bei dem die Experten diese Arbeit beurteilt und weitere Anregungen gegeben haben.

2.3 Entwicklungsumgebungen

Um Codebeispiele für diese Arbeit zu erstellen, zu kompilieren und auszuführen wurden jeweils für Java und COBOL IDEs verwendet.

Für Java-Code wurde die bekannte Eclipse² Umgebung verwendet. Dabei handelt es sich um eine etablierte IDE, welche eine Vielzahl von Funktionen zur Entwicklung und zum Debugging liefert.

¹<https://itestra.com/leistungen/software-renovation/>

²<http://www.eclipse.org/>

Der COBOL-Code dieser Arbeit wurde in der OpenCobolIDE³ entwickelt. Dabei handelt es sich um eine minimalistische IDE, welche zum Beispiel Syntax-Highlighting oder eine übersichtliche Darstellung von Fehlern bietet. Der darunterliegende Compiler GnuCOBOL⁴ wurde jedoch auch teilweise direkt als Kommandozeilenwerkzeug ausgeführt. Im Gegensatz zur sonst üblichen COBOL-Entwicklung auf einem Hostsystem ermöglicht dieser Compiler das Erzeugen von ausführbaren Dateien für gängige Linux Betriebssysteme. Dies war in dieser Arbeit sehr wichtig, um nicht auf ein System angewiesen zu sein, welches meist nur in Produktivumgebungen betrieben wird und zu dem der Zugriff oft mühsam und – durch die verschiedenen Abrechnungsmodelle dieser Hostrechner – teuer oder schlichtweg nicht möglich ist.

Das erste COBOL-Programm

Um bereits an dieser Stelle einen kleinen Einblick in COBOL, die Programmierung und die Ausführung mit der OpenCobolIDE zu bekommen, wird ein kurzes COBOL-Programm implementiert. Die einzelnen Bestandteile davon werden im Laufe der Arbeit genauer beschrieben.

```
1      IDENTIFICATION DIVISION.  
2      PROGRAM-ID. HELLO_USER.  
3      DATA DIVISION.  
4      FILE SECTION.  
5      WORKING-STORAGE SECTION.  
6          01 USERNAME PIC X(20) VALUE SPACES.  
7  
8      PROCEDURE DIVISION.  
9      MAIN-PROCEDURE.  
10         DISPLAY "Your name: " WITH NO ADVANCING.  
11         ACCEPT USERNAME.  
12         IF USERNAME EQUALS SPACES  
13             MOVE "world" TO USERNAME  
14         END-IF.  
15         DISPLAY "Hello " USERNAME.  
16         STOP RUN.  
17     END PROGRAM HELLO_USER.
```

Listing 2.1: Erstellen eines neuen COBOL Programms

³<https://github.com/OpenCobolIDE/OpenCobolIDE>

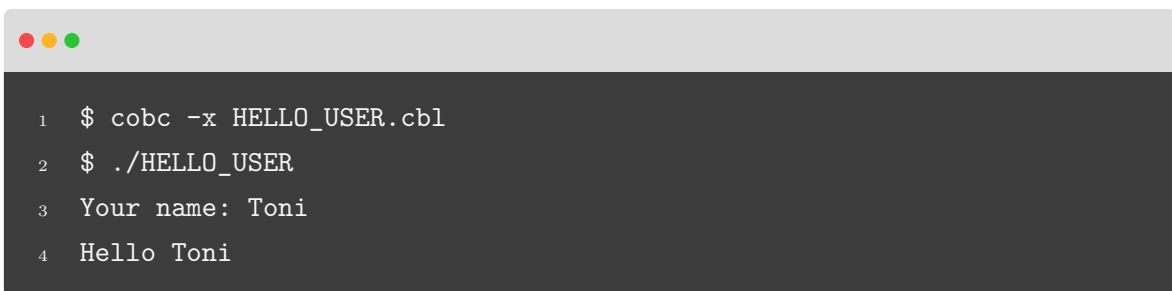
⁴<https://sourceforge.net/projects/open-cobol/>

Legt man in der OpenCobolIDE ein neues Programm an, so enthält die Datei das bekannte »Hello world« als Beispielprogramm. Wir werden dieses Programm nun so erweitern, dass es die Eingabe eines Benutzernamens erwartet und so eine persönliche Begrüßung ausgibt. Listing 2.1 zeigt das fertige Programm.

Als erstes wurde die **PROGRAM-ID** festgelegt. Dies ist der Programmname, wie er auch nach außen – für eventuelle andere Programm – sichtbar wird, und sollte daher eindeutig sein. Wichtig hierbei ist auch das Setzen des richtigen Namens in der letzten Zeile, die das **END PROGRAM** enthält.

Anschließend wird eine Variable mit dem Namen **USERNAME** angelegt, die aus 20 alphanumerischen Zeichen (**PIC X(20)**) besteht und mit Leerzeichen (**VALUE SPACES**) initialisiert wird.

Mittels **DISPLAY** wird der Nutzer aufgefordert seinen Namen einzugeben, den das **ACCEPT**-Schlüsselwort dann in die angesprochene Variable schreibt.

A terminal window with a dark background and light text. It shows the execution of a COBOL program. The first two lines are commands: '\$ cobc -x HELLO_USER.cbl' and '\$./HELLO_USER'. The next two lines are the program's output: 'Your name: Toni' and 'Hello Toni'.

```
1 $ cobc -x HELLO_USER.cbl
2 $ ./HELLO_USER
3 Your name: Toni
4 Hello Toni
```

Listing 2.2: Erstes COBOL-Programm in der Kommandozeile

Anschließend wird geprüft ob der Nutzer eine Eingabe gemacht hat. Ist dies der Fall, wird eine persönliche Begrüßung ausgegeben. Andernfalls erscheint die generische Meldung »Hello world«. Diese Ausgaben werden wie bereits die Eingabeaufforderung mit **DISPLAY** ausgegeben.

Kompiliert wird das Programm nun mit den Tasten *F8* (kompilieren) bzw. *F5* (kompilieren und ausführen). Soll der GnuCOBOL-Compiler direkt ausgeführt werden, reicht ein einfaches Kommando um eine ausführbare Datei zu erstellen. Dies wird in Listing 2.2 dargestellt.

3 Herausforderungen für COBOL und Java in betrieblichen Informationssystemen

Bei der Entwicklung von betrieblichen Informationssystemen sehen sich Entwickler mit grundlegenden Fragen und Anforderungen an die einzusetzenden Technologien und Programmiersprachen konfrontiert.

Dieses Kapitel soll einen Überblick über die wichtigsten Entscheidungskriterien für die Herangehensweise geben und aufzeigen, welchen Herausforderungen sich – im Speziellen COBOL und Java – in diesen Informationssystemen stellen müssen.

3.1 Datenmengen und Dimensionierung

3.2 Langlebigkeit, Wartbarkeit und Verlässlichkeit

3. -Langlebigkeit&Wartbarkeit -> Infrastruktur, nicht Sprache

3. -Verlässlichkeit -> Infrastruktur, nicht Sprache

3.3 Modularisierung, Wiederverwendbarkeit und Variabilität

3.3 Storyline

Sehr wichtige Punkte bei der Entwicklung von betrieblichen Informationssystemen sind die Modularisierung und Wiederverwendbarkeit. Um ein System für die Zukunft wart-

und erweiterbar zu machen ist eine gewisse Modularisierung anzustreben. Code muss somit nicht mehrmals geschrieben werden, was auch das spätere Einarbeiten in ein Projekt erleichtert, da der Projektumfang deutlich verringert werden kann.

Zudem ist, sei es um um Beispiel verschiedene Mandanten, Tarife oder Sparten abzubilden, die im Grunde die selbe Logik beinhalten, in betrieblichen Informationssystemen häufig eine gewisse Variabilität gefordert. Auch diese kann durch Wiederverwendbarkeit und Modularisierung stark begünstigt werden.

Java

Java ist eine hoch modulare Sprache. Alleine objektorientierte Paradigmen wie Kapselung, Polymorphie oder Aggregation/Komposition sorgen dafür, dass Code in hohem Maße wiederverwendet werden kann. Dabei ist vor allem die Gliederung in Klassen und Funktionen (siehe Abschnitt 4.5) ausschlaggebend. Des weiteren können Bibliotheken als Java Archive (kurz `jar` genannt) distribuiert werden und in anderen Projekten wiederverwendet werden. Dieses Konzept nutzt auch die Programmiersprache an sich bereits in hohem Maße aus und so werden viele Funktionalitäten über Packages (siehe ??) bereitgestellt. Die am häufigsten gebrauchten Bibliotheken sind dabei `java.util`, welche grundlegende Datenstrukturen wie zum Beispiel Listen (siehe Unterabschnitt 5.1.1) bereitstellt, `java.io` welche Datenein- und Ausgabe ermöglicht und allen voran `java.lang` welche – wie der Name bereits andeutet – Ergänzungen zu Programmiersprachlichen Mitteln liefert.

Durch diese praktischen Modularisierungsmöglichkeiten ist es in Java auch gut möglich Variabilität zu erreichen. So kann bestehende Logik wiederverwendet oder beispielsweise durch Vererbung minimal angepasst und nachträglich erweitert werden und sorgt dafür, dass Wartungen am System die sich auf Erweiterungen des Umfangs beziehen – z.B. das Einführen eines neuen Tarifs – mit verhältnismäßig geringem Aufwand umgesetzt werden können.

Ein weiterer Punkt der Java zu einer Sprache macht, die dafür sorgt, dass Programme wiederverwendet werden können ist die Tatsache, dass Java in plattformunabhängigen Byte-Code übersetzt wird. Die Java-Virtual-Machine (*JVM*) führt dann diesen Byte-Code aus und sorgt so dafür, dass bereits kompilierte Programme auf allen Systemen mit JVM ausführbar sind und so weiterverteilt werden können ohne neu kompiliert

werden zu müssen. Diese JVM wiederum ist ein plattformabhängiges System, welches jedoch für – nahezu – alle gängigen Systeme und Plattformen verfügbar ist.

COBOL

Im Gegensatz zu Java lässt COBOL ein Modularisierungskonzept vermissen. Wie in Abschnitt 4.5 nachzulesen ist, fehlen grundlegende Spracheigenschaften um die Wiederverwendbarkeit von Code sicherzustellen.

Wie ein Herr Lamperstorfer betonte, sieht man daher in der Praxis oftmals Code-Blöcke die ein und die selbe Logik abbilden, aber durch die Verwendung von anderen Daten nochmals im Copy-Paste-Stil in den Code integriert wurden. Das sorgt für ein hohes Maß an Redundanz. Um diese Redundanz zu vermeiden werden aber auch gängigerweise Datenstrukturen für mehr als nur einen Zweck im Programm »missbraucht«. Darunter leidet natürlich die Les- und Wartbarkeit von COBOL-Code sehr, da häufig nicht klar ist welche Daten, in welchem Kontext, wie verwendet werden. Zu diesem Thema sei auf Abschnitt 4.8.2 verwiesen.

In COBOL kann Variabilität im Vergleich zu Java nur sehr schwer erreicht werden. Code muss oftmals in hohem Maße kopiert werden um ähnliche Funktionalität abzubilden und so fallen Anpassungen in diesem Bereich unverhältnismäßig groß aus.

Auch ein Bibliothekskonzept ist in COBOL nicht vorhanden. So werden Programme und aufgerufene Unterprogramme beim Kompilieren statisch zu einer ausführbaren Einheit gelinkt. Um dieses Verhalten zumindest soweit zu beeinflussen, dass dynamisch geladenen Unterprogramme entstehen kann als »Trick« eine Variable eingeführt werden, welche den Namen des Unterprogramms enthält. Wird nun das Programm aufgerufen, welches in dieser Variable definiert ist und nicht in einer festen Zeichkette definiert ist, nimmt der Compiler an, dass das geladene Unterprogramm variieren kann – auch wenn der Inhalt der Variablen nicht verändert wird – und vermeidet so ein statisches Linken.

Zwar unterstützt COBOL in neueren Standards und Compilern eine objektorientierte Entwicklung, jedoch ist diese Spracherweiterung in der Praxis irrelevant. Die meisten gängigen Systeme auf denen COBOL Programme betrieben werden verfügen nicht über derartig neue Compiler und auch bei der Verwendung merkt man, dass diese Konzepte nachträglich hinzugefügt wurden und eigentlich nicht Bestandteil der Sprache sind.

Hat man das Glück ein System mit einem kompatiblen Compiler zu haben, so bleibt als weiterer Stolperstein der Fakt, dass die ohnehin raren COBOL-Entwickler nicht mit der Verwendung von objektorientierter Entwicklung firm sind. Daher wird diese Spracherweiterung in der vorliegenden Arbeit nicht behandelt.

3.4 Darstellungsgenauigkeit – Fließ- und Festkommaarithmetik

Vor allem in betrieblichen Informationssystemen – die oftmals Geldbeträge durch eine gewisse Anzahl von Rechenschritten errechnen sollen – ist es unerlässlich einen Blick auf die Rechengenauigkeit des Systems und der verwendeten Sprachen zu werfen. Diese ist oftmals eine Folge der Speicherrepräsentation – irrationale Zahlen können durch endlichen Speicherbedarf nicht abgebildet werden – rationaler Zahlen, die erheblichen Einfluss auf den Darstellungsbereich hat. Man unterscheidet grundsätzlich zwischen Speicherungen in Fließ- und Festkomma-Darstellung.

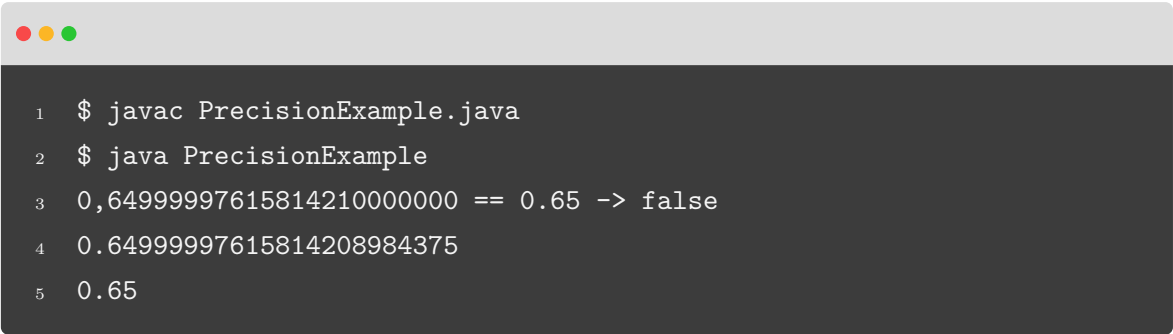
Fließkommaarithmetik

In modernen Programmiersprachen wie Java werden Datentypen für rationale Zahlen in der Fließkommarepräsentation gespeichert. Daher auch der Name **float** für engl. »floating point«. Diese Darstellung hat den großen Vorteil, dass sowohl kleine Zahlen die gegen Null gehen als auch sehr große Zahlen mit dem gleichen Speicherbedarf dargestellt werden können, da quasi das Dezimaltrennzeichen verschoben werden kann. Java verwendet zur Darstellung standardmäßig den Datentypen **double**, also ein **float** mit doppelter Darstellungsgenauigkeit bzw. doppeltem Speicherbedarf. Somit wird es möglich als kleinsten Absolutwert 2^{-1074} und als größten $(2 - 2^{-52}) \cdot 2^{1023}$ darzustellen.

Dabei werden Zahlen nach *IEEE 754*-Standard in Vorzeichen, Exponent und Mantisse umgerechnet und gespeichert. Ohne näher auf diesen eingehen zu wollen sei kur erwähnt, dass dieser einen Algorithmus festlegt, mit dessen Hilfe Variablen in einem Speicherbereich repräsentiert werden. Dieser Speicherbereich kann sich je nach Datentyp und Programmiersprache zwar unterscheiden, ist jedoch an sich stets fester Größe. Dadurch und durch den Umstand, dass Zahlen vor dem Speichern umgerechnet werden ergibt sich die Problematik, dass bestimmte Zahlen nicht exakt repräsentiert werden können

und lediglich die »näheste« Repräsentation gespeichert werden kann. Dieser Effekt ist schwer absehbar und kann in der Praxis zu ungenauen (Zwischen-)Ergebnissen führen.

```
1  import java.math.BigDecimal;
2
3  public class FloatingPoint {
4      public static void main(String[] args) {
5          float notPrecisely = 0.65f;
6          System.out.printf("%.23f == 0.65 -> %s\n",
7                          notPrecisely, (notPrecisely == 0.65));
8
9          BigDecimal notPreciselyBigDecimal = new BigDecimal(0.65f);
10         System.out.println(notPreciselyBigDecimal);
11
12         BigDecimal preciselyBigDecimal = new BigDecimal("0.65");
13         System.out.println(preciselyBigDecimal);
14     }
15 }
```



```
1  $ javac PrecisionExample.java
2  $ java PrecisionExample
3  0,64999997615814210000000 == 0.65 -> false
4  0.64999997615814208984375
5  0.65
```

Listing 3.1: Ungenauigkeit am Beispiel einer float-Variable

Listing 3.1 zeigt beispielhaft wie die Repräsentation eines Wertes vom tatsächlichen abweichen kann. Durch die Weiterverwendung eines solchen, nicht-exakt repräsentierten, Wert würden sich unter Umständen Folgefehler in Berechnungen ergeben. Außerdem können wie gezeigt Gleichheitsvergleiche von Zahlen, insbesondere von Berechnungsergebnissen, dadurch fehlerbehaftet sein, weshalb Fließkommatypen stets auf ein Werteintervall statt auf Gleichheit geprüft werden sollten.

Diese beschriebenen Fließkommatypen werden stets zur Basis 2 berechnet und heißen daher auch binäre Fließkommatypen. In der `java.math`-Bibliothek findet sich jedoch auch ein Objekttyp `BigDecimal` welcher ein Fließkommawert zur Basis 10, also ein dezimales Fließkomma darstellt. Die Speicherung beruht indessen auf zwei `Integer`-Werten, die

ein unskalierten Faktor und einen Exponenten zur Skalierung darstellen. Außerdem ist dieser Typ steuerbar was die Rundung, die Exaktheit von Ergebnissen und das Verhalten bei nichtdarstellbaren Werten angeht. Mit den in Listing 3.1 aufgezeigten Effekten lässt sich außerdem festhalten, dass `BigDecimal`s nur über andere `BigDecimal`-Objekte oder `Strings` zuverlässig instantiiert werden können. Andere Konstruktoren speichern die übergebenen Werte in primitiven Datentypen zwischen, wodurch eben diese ungewünschten Fehler in der Repräsentation wieder auftreten. `BigDecimal` bietet somit eine Möglichkeit Werte exakt abzuspeichern bzw. Kenntnis über unexakte Speicherung – in der Regel durch Exceptions – zu erhalten und diese zu steuern. Mit diesem Objekttypen gehen jedoch Speicher- und Laufzeit-Overheads einher die nicht vernachlässigt werden dürfen.

Festkommaarithmetik

Um die angesprochenen Probleme zu umgehen verwenden manche Sprachen eine Festkommaarithmetik, um rationale Zahlen zu Speichern oder bieten zumindest Datentypen um eine derartige Speicherrepräsentation zu erreichen.


Dabei wird im Gegensatz zu Fließkommazahlen festgelegt wieviele Stellen einer Zahl vor- bzw. nach dem Komma gespeichert werden sollen. Jede Ziffer wird dabei für sich – je nach Implementierung durch eine bestimmte Codierung – gespeichert und erlaubt somit absolute Genauigkeit im Werte- bzw. Darstellungsbereich. Auch ist der Umgang mit Überläufen fest definiert und führt zu konsistentem und abschätzbarem Verhalten. Ergebnisse werden stets zur Speicherung »abgeschnitten« außer man definiert explizit, dass gerundet werden soll. Listing 3.2 enthält Beispiele zu beiden Varianten. `PIC 9V9(2)` deklariert eine Variable mit genau einer Vor- und zwei Nachkommastellen. Damit wäre beispielsweise sichergestellt, dass alle Geldbeträge < 10 – auch nach Berechnungen – korrekt dargestellt werden können.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. PRECISION-EXAMPLE.
3      DATA DIVISION.
4      FILE SECTION.
5      WORKING-STORAGE SECTION.
6          01 TWO-DECIMALS-VALUE PIC 9V9(2) VALUE 0.50.
7          01 THREE-DECIMALS-VALUE PIC 9V9(3) VALUE 0.499.
8          01 RESULT PIC 9V9(2) VALUE ZERO.
9
10     PROCEDURE DIVISION.
11     MAIN-PROCEDURE.
12         COMPUTE RESULT =
13             TWO-DECIMALS-VALUE + THREE-DECIMALS-VALUE.
```



```
14      DISPLAY RESULT.  
15      COMPUTE RESULT ROUNDED =  
16          TWO-DECIMALS-VALUE + THREE-DECIMALS-VALUE.  
17      DISPLAY RESULT.  
18      STOP RUN.  
19  
20  END PROGRAM PRECISION-EXAMPLE.
```



```
1 0.99  
2 1.00
```

Listing 3.2: Dezimalzahlen in COBOL

Ein weiterer Vorteil der Abbildung mit Festkomma ist die Tatsache, dass beliebig große Zahlen gespeichert werden können. Dies sorgt neben höherem Speicherbedarf, jedoch auch dafür, dass sofern Grundrechenarten für eine Ziffer implementiert sind, beliebig lange Ziffernfolgen nach dem gleichen Schema verarbeitet werden können. Die Ergebnisse werden dabei auch zeichenweise gespeichert und lassen so keine Rundungsfehler oder Fehler aufgrund von unzureichendem Speicherplatz zur Abbildung zu. Der Speicherbereich kann in COBOL jedoch zum Beispiel durch das Nutzen von **PACKED DECIMALS** mit dem Schlüsselwort **COMP-3** hinter der **PICTURE**-Anweisung reduziert werden. Hierbei wird lediglich ein Nibble ($\frac{1}{2}$ Byte) pro Ziffer benötigt.

❗ In betrieblichen Informationssystemen und speziell bei der Verarbeitung von Geldbeträgen, ist es also unerlässlich die Sicherheit einer exakten Darstellung von Zahlen zu haben. Während die binäre Fließkommadarstellung Speicherplatz-Vorteile und Flexibilität des Wertebereichs einer Zahl bietet, jedoch Werte unter Umständen nicht exakt repräsentieren kann, stellt Festkommaarithmetik sicher, dass Zahlen exakt und vorhersehbar repräsentiert werden. Dies wird durch erhöhten Speicherbereich und fehlende Flexibilität erkauft, ist jedoch in der Praxis oftmals unerlässlich. Eine Möglichkeit diese Sicherheit in Java zu erreichen ist das Nutzen des **BigDecimal**-Typen, der viele Nachteile und vor allem Unsicherheiten gegenüber binären Fließkommatypen aus dem Weg räumt. Jedoch führt dieser unter Umständen zu Performanz- bzw. Speichereinbußen. COBOL bietet mit Verwendung der Festkommaarithmetik bereits standardmäßig eine Darstellungssicherheit und Vorhersagbarkeit die vielen modernen Sprachen fehlt.

3.5 Schnittstellen und Datenquellen

In betrieblichen Informationssystemen stellen außerdem Schnittstellen ein wichtiges Thema dar. Sowohl das Bereitstellen von standardisierten und dokumentierten Interfaces als auch das Nutzen von anderen Systemen über ihre Schnittstellen ist stets Teil aller Anwendungsfälle.

Vor allem in der heutigen Zeit, in der Informationssysteme nicht mehr als alleinige Verarbeitungs-, Reporting- und Darstellungsschicht fungieren, sondern eingebettet in einen größeren Kontext aus verschiedensten Modulen, mobilen Applikationen und Websites funktionieren sollen und mit diesen kommunizieren müssen ist ein ausgereiftes Schnittstellenkonzept und die standardisierte Bereitstellung und Nutzung von Daten und Diensten unerlässlich.

Für Java sind an dieser Stelle eine Fülle an Bibliotheken erhältlich, welche Netzwerkkommunikation über verschiedenste Protokolle auf unterschiedlichen Ebenen ermöglichen. Neben diversen Fremdbibliotheken, bietet bereits das JDK bereits unterschiedliche Methoden zur Kommunikation mit Fremdsystemen.

Zudem ist es, durch die Definition von Interfaces, auch auf Klassenebene möglich, Schnittstellen zu bieten, die eine einfache Erweiterung von und Verbindungen zu Neusystemen möglich machen.

COBOL hingegen lässt an dieser Stelle einige Funktionalität vermissen. Durch das fehlende Bibliothekskonzept – wie in Abschnitt 3.3 erläutert – und das gänzliche Fehlen von Netzwerkkommunikationsmechanismen ist es, in reinem COBOL, nicht möglich Netzwerkschnittstellen festzulegen die von außen erreichbar sind oder solche zu nutzen. Auch intern kann ein COBOL-System nur bedingt Standards definieren die zwischen unterschiedlichen Programmteilen für einheitliche Kommunikationskanäle sorgen. COBOL-Systeme basieren, wie Abschnitt 3.6 beschreibt, auf einfachen EVA-Prinzipien.

Durch den Wandel der letzten Jahre bzw. Jahrzehnte in hochdimensionalen und komplexen Softwaresystemen wird es immer schwerer reine COBOL-Systeme zu nutzen und sinnvoll in eine heterogene IT-Landschaft einzubetten. Kunden sind von Applikationen für verschiedenste Systeme abhängig und müssen sich darauf verlassen können, dass die

datenquelle
datenbanken

3. Batch-
betrieb ->
Schnitt-
stellen
(Host
nicht CO-
BOL)

Erweiterung um neue Back- und Frontends, ohne Einfluss auf bestehende Komponenten vonstatten gehen kann. Auch ist es nötig die Basis für interne Systemerweiterungen zu schaffen, indem festgelegte Interfaces bedient und verwendet werden. Diese Anforderungen können mit modernen Sprachen wie Java mühelos erreicht werden, wohingegen Altsysteme nur spärliche Möglichkeiten in dieser Richtung bieten.

3.6 Reporting

EVA-Prinzip mit Zitat!

4 Vergleich wichtiger Sprachkonzepte

4.1 Programmstruktur

Dieser Abschnitt behandelt die strukturellen Unterschiede von Java- und COBOL-Programmen. Dazu wird erläutert in welche Einheiten sich die Programme der jeweiligen Sprache aufteilen lassen.

Struktur eines Javaprogramms

Abbildung 4.1 gibt einen zusammenfassenden Überblick über die Teile eines Java-Programms und bildet graphisch ab, wie sich die jeweiligen Komponenten zusammensetzen können.

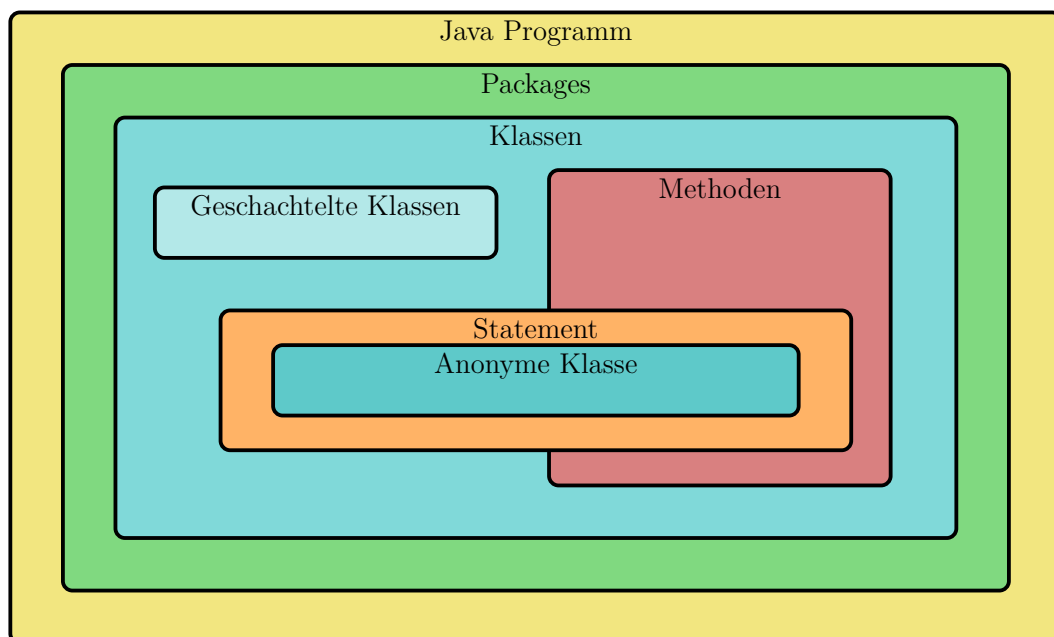


Abbildung 4.1: Strukturelle Bestandteile eines Java-Programms

Anhand dieses Diagrams werden die wichtigsten Konzepte der Strukturierung von Java-Code aufgezeigt. Zeile 1 beinhaltet die Package-Deklaration, d.h. hiermit wird die Klasse dem hier genannten Package zugeordnet. Diese Deklaration **muss** gleich der Ordnerhierarchie sein, in denen die Java-Dateien verwaltet werden.

Die nächstkleinere Einheit eines Java-Programms stellen Klassen dar. Hierbei handelt es sich um das Kernkonzept der objektorientierten Programmierung. Von dieser Klasse können fortan Objekte instanziiert werden. Um einen tieferen Einblick in die Thematik der Objektorientierung zu erhalten sei an dieser Stelle einschlägige Fachliteratur erwähnt. Diese erwähnte Klasse muss dabei in einer Datei gespeichert sein, die den selben Namen trägt wie die Klasse selbst. Aus dem Klassennamen `MasterThesis` folgt also der Dateiname `MasterThesis.java`.

Teil dieser Klassen können wiederum Methoden, Variablendeklarationen und weitere Klassen sein. Diese können jeweils statisch oder auch einer Instanz zugeordnet sein. Auch dabei handelt es sich um ein gängiges Konzept der objektorientierten Softwareentwicklung. Hierzu sei an dieser Stelle lediglich erwähnt, dass statische Methoden, Variablen und Klassen Teil der Klasse sind und kein konkret instanziiertes Objekt benötigen während nicht-statische Komponenten stets ein konkretes Objekt einer Klasse benötigen.

Diese weiteren Klassen haben strukturell die selben Eigenschaften wie die umgebende Klasse, außer, dass sie nicht in einer Datei gespeichert sein müssen bzw. können deren Name dem Klassennamen entspricht.

Methoden wiederum bestehen aus einzelnen Statements. Zu erwähnen ist, dass Variablendeklarationen auch ein Statement darstellen. Während Variablendeklarationen an jeder Stelle innerhalb einer Klasse möglich sind, sind andere Statements als Teil einer Klasse nur dann gültig, wenn diese in geschweiften Klammern stehen. Diese Blöcke werden – der Reihe nach – vor jedem Konstruktoraufruf ausgeführt und heißen deshalb auch *Initializer*. Auch ist die Definition von statischen *Initializer* möglich, die einmalig nach dem Laden einer Klasse ausgeführt werden. Listing 4.1 führt Beispiele dafür an.

```
1  public class Initializer {  
2  
3      static { // static initializer  
4          System.out.println("Executed once for class on load");  
5      }  
6  
7      { // initializer
```

```
8      System.out.println("Called once for every instance");
9  }
10
11  public Initializer() {
12      System.out.println("Executed after all initializers");
13  }
14
15  public static void main(String[] args) {
16      System.out.println("main: start");
17      Initializer initializer = new Initializer();
18      System.out.println("main: termination");
19  }
20 }
```

Listing 4.1: Initializer in Java

Statements die aus Variablendeklarationen, Zuweisungen oder Methodenaufrufen bestehen, müssen im Gegensatz zu Block-Statements, wie z.B. Schleifen oder Verzweigungen, stets mit einem Semikolon beendet werden.

Die letzten strukturellen Elemente sind anonyme Klassen und Funktionen, auch Lambda-Funktionen genannt. Wobei anonyme Funktionen in Java genaugenommen nur eine syntaktische Schreibweise einer speziellen anonymen Klasse sind. Die Verwendung wird in Listing 4.2 illustriert. Die Zeilen 9 – 14 beinhalten eine anonyme Klasse, die das `IntConsumer`-Interface implementiert. Die völlig identische anonyme Klasse wird implizit durch die Lambda-Funktion in den Zeilen 16 – 18 implementiert.

```
1  package de.masterthesis;
2
3  import java.util.function.IntConsumer;
4  import java.util.stream.IntStream;
5
6  public class AnonymousClassAndMethodExample {
7
8      public static void main(String[] args) {
9          IntStream.range(0, 10).forEach(new IntConsumer() {
10              @Override
11              public void accept(int value) {
12                  System.out.print(value + " ");
13              }
14          });
15      }
```

```

15     System.out.println();
16     IntStream.range(0, 10).forEach(value -> {
17         System.out.print(value + " ");
18     });
19 }
20 }

```

Listing 4.2: Anonyme Klassen und Funktion in Java

Neben der inhaltlichen Struktur bleibt noch zu erwähnen, dass Java Programme keinen festen Formatierungsregeln folgen müssen. Neben einigen wenigen festgelegten Eigenschaften – die Packagedeklaration muss vor Imports stehen, welche wiederum vor allem anderen stehen müssen – können Java-Programme beliebig formatiert werden.

Struktur eines COBOL-Programms

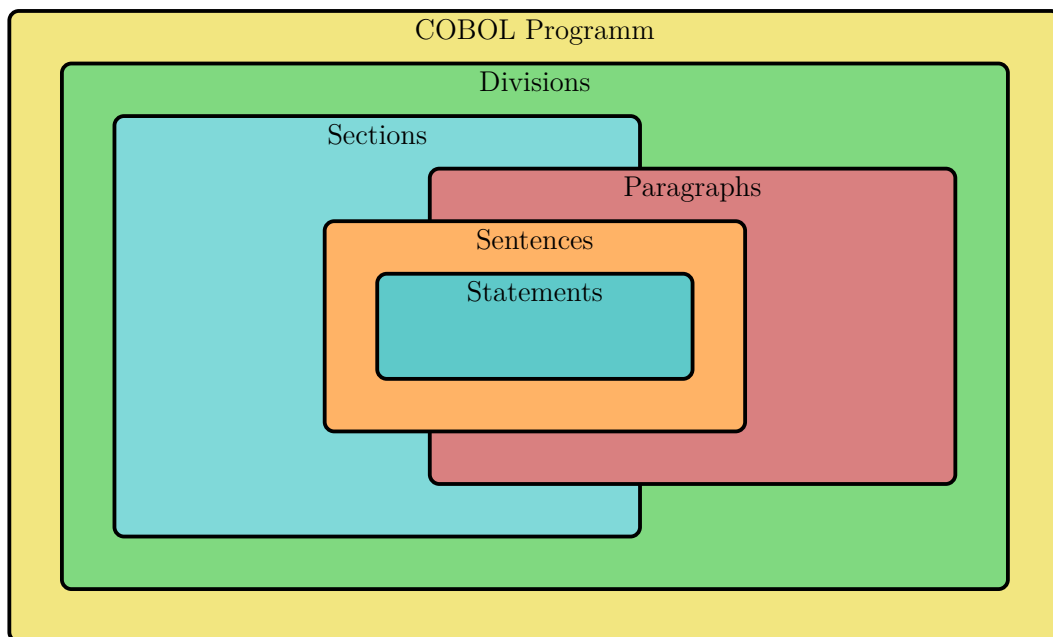


Abbildung 4.2: Strukturelle Bestandteile eines COBOL-Programms

Abbildung 4.2 zeigt die strukturellen Bestandteile eines COBOL-Programms. Ein Programm kann bestehen aus vier fest definierten Divisions:

- **IDENTIFICATION DIVISION** – Hier werden grundlegende Daten zum Programm, wie der Name oder der Autor festgelegt.
- **ENVIRONMENT DIVISION** – Definiert die Ein- und Ausgabe sowie Konfigurationen der Systemumgebung.
- **DATA DIVISION** – Diese Division beinhaltet die Definitionen von Daten. Dazu zählen Variablen oder auch Datei-Record-Definitionen.
- **PROCEDURE DIVISION** – Innerhalb dieser Division befindet sich der ausführbare Code.

Eine Division – außer der **IDENTIFICATION DIVISION** – kann wiederum aus verschiedenen Sections bestehen, wobei diese nur innerhalb der **PROCEDURE DIVISION** frei definiert werden können.

Die **ENVIRONMENT DIVISION** kann eine **CONFIGURATION SECTION** und eine **INPUT-OUTPUT SECTION** enthalten. Erstere erlaubt Definitionen zum Zielsystem. Letztere beinhaltet Definitionen zum Dateizugriff sowie zu Ein- und Ausgabeoperationen.

Teil der **DATA DIVISION** sind folgende Sections:

- **FILE SECTION** – Definiert Dateien bzw. Dateischemata auf die im Programm zugegriffen werden soll.
- **WORKING-STORAGE SECTION** – Enthält Variablendeklarationen, welche über mehrere Programmaufrufe hinweg bestehen bleiben.
- **LOCAL-STORAGE SECTION** – Enthält Variablendeklarationen, welche bei jedem Programmaufruf neu alloziiert werden.
- **LINKAGE SECTION** – Enthält Definitionen von Variablen, welche bei einem Programmaufruf von außen übergeben werden können.

In der **PROCEDURE DIVISION** finden sich schließlich vom Entwickler definierte Sections, welche ein COBOL-Pendant zu Funktionen in Java darstellen.

Die nächstkleinere Einheit eines COBOL-Programms stellen Paragraphs dar. Diese lassen sich – mit kleinen Unterschieden – im Allgemeinen wie Sections verwenden.

In bestehenden COBOL-Programmen lassen sich daher zwei unterschiedliche Stile beobachten. Auf der einen Seite gibt es Programme die lediglich aus Paragraphs bestehen und auf der anderen existieren Systeme in denen Sections verwendet wurden und durch Paragraphs untergliedert sind. Generell ist zweite Variante vorzuziehen wie auch Richards in „Enhancing Cobol Program Structure“ [20] beschreibt, da dadurch sowohl die Programmstruktur lesbarer wird als auch die Fehleranfälligkeit verringert wird. Auf beide Eigenschaften wird im weiteren Verlauf der Arbeit eingegangen. An dieser Stelle soll lediglich festgehalten werden, dass es verschiedene Varianten gibt.

Sections und Paragraphs können wiederum aus Sentences bestehen. Dabei handelt es sich um ein oder mehr Statements. Ein Sentence wird stets von einem Punkt abgeschlossen. Während Sections Paragraphs also Analogien zu Methoden in Java sind kann man Sentences am ehesten mit Block-Statements – sobald diese geschachtelt werden stimmt diese Analogie nicht mehr – und Statements mit Semikolon-terminierten Statements in Java vergleichen. Herr Streit merkte dazu im Interview an, dass diese Punkte stets ein Statement terminieren sollten. Alles andere sei schlechter Stil und könne zu unabsehbaren semantischen Fehlern führen.

COBOL-Programme lassen sich nicht beliebig formatieren. So folgt ein COBOL-Programm einem festgelegten spaltenweisen Aufbau:

- **Spalte 1 – 6**

In diesen Spalten befindet sich die sog. Sequenznummer. Damit können Programmzeilen nummeriert werden. Der Zeichensatz dafür entspricht dem zugrundeliegenden System. So könnten Zeilen auch beispielsweise mit Buchstaben versehen werden.

- **Spalte 7**

In dieser Spalte kann ein Zeichen gesetzt werden, um dem Compiler die Bedeutung der Zeile kenntlich zu machen. Ein * leitet z.B. eine Kommentarzeile ein und mit - kann ein nicht-numerisches Literal aus der vorherigen Zeile fortgeführt werden.

- **Spalte 8 – 11 und Spalte 12 – 72**

Diese Spalten enthalten Definitionen und ausführbaren Programmcode. Je nach COBOL-Dialekt sind diese beiden Bereiche jedoch im Hinblick auf Variablendeklarationen unterschiedlich. Während in ersterem nur die Stufennummern 01 und 77 deklariert werden dürfen, müssen alle anderen in dem Bereich ab Spalte 12 stehen. Dies gilt jedoch wie erwähnt nicht auf allen Systemen.

- **Spalte 73 – 80**

In klassischem COBOL dienen diese Spalten dazu Kommentare zur aktuellen Zeile einzufügen. Wie Herr Streit betonte sind diese in Altsystemen exzessiv genutzt um Versionsinformationen – wie Änderungsdatum oder Ticketnummern – festzuhalten, sollten jedoch zum Wohl der Übersichtlichkeit entfernt werden und im Zuge einer Renovierung oder Migration durch eine modernen Versionsverwaltung wie z.B. SVN oder Git ersetzt werden. Auch ein Änderungsvergleich zwischen Programmversionen in einem entsprechenden Werkzeug wird durch diese Kommentare erheblich erschwert. Wichtig sei es jedoch zu verstehen, dass diese Kommentare, zu Zeiten in denen es keine Versionsverwaltungssoftware gab, sinnvoll waren.

Im sogenannten *Free-Format*, welches von einigen COBOL-Dialekten unterstützt wird, gelten diese Beschränkungen nicht. Dabei gilt lediglich, dass Spalte 1 wie Spalte 7 zur Kennzeichnung von Kommentaren fungiert. Auch die Breite eine Zeile kann hierbei, im Gegensatz zum klassischen COBOL, 80 Zeichen überschreiten.

4.2 Variablen und Datentypen

Eine wichtiges Sprachmittel von Programmiersprachen ist die Verwendungsmöglichkeit von Variablen. Je nach Programmiersprache haben diese Variablen unterschiedliche Eigenschaften und werden verschieden deklariert, initialisiert bzw. definiert. Dieser Abschnitt soll die Unterschiede dabei zwischen COBOL und Java herausarbeiten.

Variablen in Java

Eine Variable in Java hat stets einen bestimmten Datentypen. Dies können primitive Datentypen – **float**, **double**, **byte**, **char**, **short**, **int**, **long**, **boolean** – aber auch komplexe Objekttypen sein. Variablen primitiver Zahltypen haben dabei stets ein Vorzeichen.

```
1 public class VariableExample {  
2  
3     final int CONSTANT_VARIABLE = 1907;  
4     int primitiveClassVariable = 0;  
5     VariableExample complexClassVariable = null;  
6 }
```

```
7     public static void main(String[] args) {  
8         // CONSTANT_VARIABLE = 1860 -> Fehler  
9         int primitiveLocalVariable = 1;  
10        VariableExample complexLocalVariable = new VariableExample();  
11    }  
12  
13 }
```

Listing 4.3: Variablendeklarationen in Java

Listing 4.3 soll einige Konzepte der Variablendeklaration und -definition verdeutlichen:

- Variablen können sowohl als Teil einer Klasse als auch lokal innerhalb einer Methode deklariert werden.
- Variablen mit dem *Modifier* **final** sind Konstanten und können nicht mehr geändert werden.
- Die Deklaration erfolgt nach dem Muster »<Datentyp> <Variablenname>«.
- Die Initialisierung einer Variable geschieht durch das zuweisen eines Wertes.
- Komplexe Objekttypen können den Wert **null** haben. Das bedeutet, die Variable, die in diesem Fall eine Referenz auf einen Speicherbereich darstellt, ist leer. Hier gilt es zu beachten, dass primitive Datentypen nicht **null** sein können.
- Instanzen eines Objekttypen werden durch das Schlüsselwort **new** und den Aufruf eines Konstruktors erzeugt. Primitive Datentypen haben keine Konstruktoren.

Die Deklaration von Variablen bestimmter Datentypen sorgt dafür, dass ausreichend Speicherplatz für diese reserviert wird. Die Stelle der Deklaration im Code ist dabei frei wählbar und muss lediglich vor der ersten Verwendung stehen.

Der sog. Scope, zu deutsch Gültigkeitsbereich, gibt in der Programmierung an, in welchem Bereich eine Variable gültig ist. In Java ist der Scope einer Variablen meist einfach zu erkennen. Eine Variable ist innerhalb der geschweiften Klammern gültig, die die Variablendeklaration beinhalten. Dies verdeutlicht Listing 4.4.

Die Variable `memberVariable` ist innerhalb der gesamten Klasse `ScopeExample`, also in jeder enthaltenen Methode, verschachtelten Klassen und wiederum deren Methoden, gültig. Eine Variable mit selbem Namen kann auch innerhalb einer Methode deklariert werden. Auf die Instanzvariable kann mit dem `this`-Schlüsselwort zugegriffen werden. In geschachtelten Klassen muss zusätzlich der Klassenname vorangestellt werden wie Zeile 27 zeigt. Ist keine lokale Variable mit selbem Namen vorhanden, so kann dieses Schlüsselwort auch weggelassen werden.

Wie Zeile 19 zeigt ist es auch nicht möglich auf lokale Variablen einer anderen Funktion zuzugreifen. Gleiches gilt für Instanzvariablen verschachtelter Klassen.

```

1  package de.masterthesis;
2
3  public class ScopeExample {
4
5      int memberAndLocalVariable;
6
7      void memberFunction(int parameter) {
8          // innerMemberVariable = 0; -> Ungültig
9          int memberAndLocalVariable = 0;
10         {
11             // int memberAndLocalVariable = 1; -> Ungültig
12             int localVariable;
13         }
14         int localVariable;
15         this.memberAndLocalVariable = 0;
16     }
17
18     void otherMemberFunction() {
19         // parameter = 0; -> Ungültig
20         this.memberAndLocalVariable = 0;
21     }
22
23     class InnerClass {
24         int innerMemberVariable;
25
26         void innerMemberFunction(int innerParameter) {
27             ScopeExample.this.memberAndLocalVariable = 0;
28             innerParameter = 0;
29         }
30     }
31 }

```

Listing 4.4: Variablendeklarationen mit verschiedenen Scopes

Variablen in COBOL

Die Deklaration von Variablen unterscheidet sich in COBOL stark von der in Java. Neben der Eigenschaft, dass Variablen nur innerhalb der **DATA DIVISION** – als Teil der **WORKING-STORAGE SECTION** oder der **LOCAL-STORAGE SECTION** – deklariert werden können, ist in COBOL die Definition eines Datentyps gleichzeitig auch die Festlegung der Ausgabe-Repräsentation dieser Variable.

Dies sorgt dafür, dass bereits an der Stelle der Variablendeklaration festgelegt werden muss, wie diese Daten im folgenden Programm dargestellt werden. Das Schlüsselwort dafür ist die **PICTURE**- oder kurz **PIC**-Anweisung.

```

1  IDENTIFICATION DIVISION.
2  PROGRAM-ID. VARIABLE-EXAMPLE RECURSIVE.
3  DATA DIVISION.
4  WORKING-STORAGE SECTION.
5  01 PERSON-DATA.
6      05 PERSON-NAME PIC X(10) VALUE "Mustermann".
7      05 FILLER PIC X VALUE SPACE.
8      05 PERSON-HEIGHT-CM PIC 9(3) VALUE 178.
9      05 PERSON-HEIGHT-M REDEFINES PERSON-HEIGHT-CM PIC 9V99.
10
11  PROCEDURE DIVISION.
12      DISPLAY PERSON-DATA.
13      DISPLAY PERSON-NAME.
14      DISPLAY PERSON-HEIGHT-CM.
15      DISPLAY PERSON-HEIGHT-M.
16
17  END PROGRAM VARIABLE-EXAMPLE.
```



```

1  $ ./variableExample
2  Mustermann 178
3  Mustermann
4  178
5  1.78
```

Listing 4.5: Variablendeklarationen in COBOL

Listing 4.5 demonstriert die Deklaration von vier Variablen. Variablen Anhand dieses Beispiels sollen wiederum verschiedene Konzepte der Variablendeklaration in COBOL illustriert werden

Jede Variablendeklaration beginnt mit einer Stufennummer. Diese Stufennummer sorgt für Gruppierung von Variablen. Zulässig sind dabei Zahlen zwischen 01 und 49. Die Stufennummern sollten mit ausreichendem Abstand gewählt werden – in der Praxis werden dazu 5er-Schritte gewählt – um ein nachträgliches Einfügen zwischen zwei Stufennummern zu erleichtern. Die speziellen Stufennummern 66, 77 und 88 werden später separat behandelt. Wie das Beispiel zeigt, lassen sich auf einzelne Variablen auch über den Gruppennamen zugreifen. Konstanten sind hierbei in COBOL nicht möglich und so gilt es, wie Herr Bonev und Herr Lamperstorfer betonten, sicherzustellen, dass konstante Werte an keiner Stelle des Programms verändert werden.

Der Stufennummer folgt ein eindeutiger Name für die Variable. An dieser Stelle kann jedoch auch das Schlüsselwort **FILLER** verwendet werden. Dies sorgt dafür, dass eine Platzhaltervariable angelegt wird, auf die jedoch später nicht direkt zugegriffen werden kann.

Die Festlegung der Repräsentation geschieht wie bereits erwähnt durch ein **PIC**. Nach diesem **PIC** wird festgelegt wie diese Variable dargestellt werden soll. »X« steht dabei für ein alphanumerisches, »9« für ein numerisches Zeichen und »A« für einen Buchstaben. Die Angabe der Stellen einer Variable wird durch die Wiederholung des jeweiligen Zeichens oder die verkürzte Notation mit der nachgestellten Anzahl der Wiederholungen in runden Klammern – z.B. XXXX $\hat{=}$ X(4) – erreicht. Als Dezimaltrennzeichen wird wie gezeigt ein »V« verwendet und ein vorangestelltes »S« sorgt dafür, dass eine numerische Variable ein Vorzeichen führt. Es wird also genau festgelegt wie viele Vor- und Nachkommastellen eine Variable hat.

Die Initialisierung einer Variable erfolgt durch die **VALUE**-Anweisung, gefolgt von dem Wert, welcher der Variablen zugewiesen werden soll. Dabei gibt es die Schlüsselwörter **SPACE** bzw. **SPACES** und **ZERO** bzw. **ZEROS**, die anstelle eines Wertes verwendet werden können, um eine Variable mit Leerzeichen bzw. Nullen zu initialisieren.

Durch die Definition der Repräsentation findet man in der Praxis oft Variablen, die auf den selben Speicherbereich wie eine andere Verweisen, jedoch die dort enthaltenen Daten anders darstellen bzw. interpretieren. Dies geschieht wie im Beispiel gezeigt mithilfe des Schlüsselworts **REDEFINES**.

Eine Typsicherheit ist in COBOL nicht ausreichend gewährleistet. So kann eine Variable mit **REDEFINES** oder eine Variable, welche die eigentliche gruppiert, den Speicherbereich einer anderen mit, unter Umständen ungültigen, Werten befüllen. Auch sind uninitialisierte Variablen teilweise mit falschen Datentypen vorbelegt.

Auch der Scope von Variablen in COBOL unterscheidet sich sehr stark von Java. Allgemein kann festgehalten werden, dass auf eine Variable von jeder Stelle innerhalb eines Programms aus zugegriffen werden kann. Das sorgt dafür, dass schnell Fehler auftreten können, die sich durch unbeabsichtigten Zugriff auf falsche Variablen ergeben. In der Praxis sind diese, so alle befragten Experten übereinstimmend, häufiger beobachtbar und bergen hohes und vor allem schwer auszumachendes Fehlerpotential. An dieser Stelle sei lediglich der Unterschied der **WORKING-STORAGE SECTION** und der **LOCAL-STORAGE SECTION** erwähnt. Während Variablenwerte in ersterer über mehrere Programmaufrufe hinweg erhalten bleiben, werden Variablen der **LOCAL-STORAGE SECTION** bei jedem Aufruf neu instanziiert. Diesen Unterschiedes sind sich COBOL-Entwickler in der Praxis nicht immer bewusst, obwohl so zumindest teilweise eine Reduzierung von Seiteneffekten durch falsch belegte Variablen erreicht werden könnte. Daher war Herrn Streits Vorschlag, Variablen wenn möglich in der **LOCAL-STORAGE SECTION** anzulegen.

Zum Abschluss dieses Abschnitts sei erwähnt, dass der Speicherplatz von Variablen weder in COBOL noch in Java händisch freigegeben werden. In Java sorgt der *garbage collector* dafür, dass Speicherbereich, der nicht mehr verwendet wird wieder freigegeben wird. In COBOL geschieht dies mit dem Ende eines Programms.

4.3 Arrays

Eine zentrale Datenstruktur in der Programmierung stellen Felder bzw. Arrays – in COBOL auch als *table* bezeichnet – dar. Dabei handelt es sich um eine geordnete Sammlung von Werten des selben Typs auf die, im Gegensatz zu z.B. verketteten Listen, direkt zugegriffen werden kann.

```

1  public class Arrays {
2
3      public static void main(String[] args) {
4          int[] intArray = new int[10];
5
6          for(int counter = 0; counter < intArray.length; counter++)
7              {

```

```

8         intArray[counter] = counter;
9     }
10 }
11 }

```

Listing 4.6: Felder in Java

Zeile 4 in Listing 4.6 beschreibt das Anlegen eines Arrays in Java mittels **new**-Schlüsselwort, wohingegen Zeile 8 den Zugriff auf ein Element zeigt. Die Indizierung der Elemente beginnt dabei mit dem Element 0. Ein Feld der Größe 10 hat also die Indizes 0 – 9. Sowohl beim Anlegen als auch beim Zugreifen auf ein Element des Arrays wird der `[]`-Operator verwendet.

In COBOL können Felder durch **OCCURS**, gefolgt von der Anzahl der zu speichernen Werte und **TIMES** angelegt werden. Dies illustriert Listing 4.7. Das **INDEXED BY**-Schlüsselwort kann dazu genutzt werden, eine Variable zu definieren, mit der das Array indiziert werden kann. Dies ist jedoch nicht zwangsläufig notwendig. Hierbei sei zu erwähnen, dass Indizes in COBOL auch *subscript* genannt werden.

```

1  IDENTIFICATION DIVISION.
2  PROGRAM-ID.  ARRAYS.
3
4  DATA DIVISION.
5  WORKING-STORAGE SECTION.
6      01 ARRAY-ELEMENT PIC 9(2) OCCURS 10 TIMES
7         INDEXED BY ELEMENT-INDEX.
8
9  PROCEDURE DIVISION.
10 MAIN-PROCEDURE.
11     ACCEPT ELEMENT-INDEX.
12     ACCEPT ARRAY-ELEMENT(ELEMENT-INDEX) .
13     DISPLAY ARRAY-ELEMENT(ELEMENT-INDEX) .
14     STOP RUN.
15
16 END PROGRAM ARRAYS.

```

Listing 4.7: Felder in COBOL

In Zeile 12 ist der Zugriff auf ein einzelnes Feld-Element zu sehen. Dies geschieht in COBOL mittels runder Klammern. Zu beachten ist hierbei, dass COBOL die einzelnen

Elemente beginnend mit 1 indiziert. Im Gegensatz zu Java hat ein Array der Größe 10 in COBOL also die Indizes 1 – 10.

Eine erwähnenswerte Besonderheit von Feldern in COBOL und in Java ist, dass diese auch mehrdimensional sein können. In COBOL spricht man statt von Dimensionen von Level. Jedes Element der ersten Dimension bzw. des ersten Levels besteht also aus einem weiteren Feld. Ein Zugriff auf ein beispielhaftes zweidimensionales Array ist dann mittels [] [] in Java bzw. (X,Y) in COBOL möglich. Java und COBOL unterscheiden sich jedoch dahingehend, dass COBOL auch einen Zugriff auf eine ganze Dimension ermöglicht wohingegen in Java stets ein einzelnes Element referenziert werden muss. Dies gilt sowohl für schreibenden als auch lesenden Zugriff.

Eine weitere Gemeinsamkeit ist die Tatsache, dass Felder in beiden Sprachen eine feste Größe haben. Nach dem Anlegen des Feldes kann diese Größe nicht mehr geändert werden. Jedoch muss in COBOL bereits zur Zeit der Kompilierung festgelegt werden, wie viele Elemente ein Array beinhalten soll. In Java kann diese Größe auch variabel zur Laufzeit des Programms festgelegt werden.

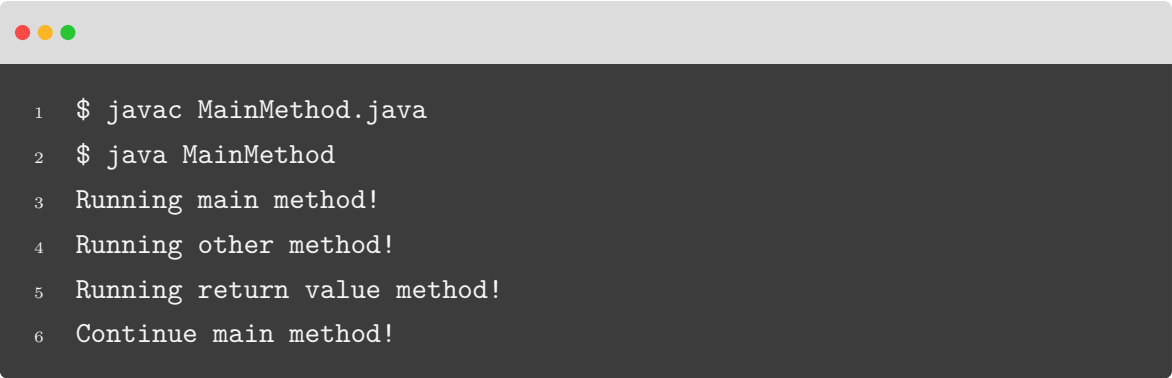
4.4 Programmablauf und Kontrollfluß

Durch die in Abschnitt 4.5 erläuterten Unterschiede ergeben sich auch im Programmablauf Diskrepanzen. Allerdings bleibt hier zu erwähnen, dass diese Unterschiede nicht zu einem gänzlich anderen Ablauf führen, sondern eher dafür sorgen, dass Gemeinsamkeiten nicht auf den ersten Blick erkennbar sind, obwohl der Ablauf im Grunde sehr ähnlich ist. Sowohl Java- als auch COBOL-Programme werden im Allgemeinen von oben nach unten durchlaufen. Beiden Programmiersprachen ist gemein, dass sie einen definierten Einstiegspunkt in ein Programm haben. Während jedes Java-Programm in der `main`-Methode startet wird ein COBOL-Programm stets sequenziell von oben nach unten abgearbeitet und durchlaufen und beginnt daher stets mit der ersten Zeile der **PROCEDURE DIVISION**.

4.4.1 Ablauf

```
1  public class MainMethod{  
2      public static void main(String[] args) {
```

```
3      System.out.println("Running main method!");
4      otherMethod();
5      System.out.println("Continue main method!");
6  }
7
8  public static void otherMethod() {
9      System.out.println("Running other method!");
10 }
11 }
```



```
1 $ javac MainMethod.java
2 $ java MainMethod
3 Running main method!
4 Running other method!
5 Running return value method!
6 Continue main method!
```

Listing 4.8: Java main-Methode

Listing 4.8 demonstriert einen sehr simplen Programmablauf in Java. Wie bereits erwähnt ist der Startpunkt eines jeden Java-Programms die **main**-Methode. Von dieser aus können weitere Methoden aufgerufen werden und sobald das Ende dieser Methode erreicht ist terminiert das Programm. Im vorliegenden Beispiel wird also nach einer Ausgabe in **main**, die Funktion **otherMethod** aufgerufen, bevor der Ablauf wieder in der **main**-Methode fortgesetzt wird. Daran soll folgendes Verhalten deutlich werden: Endet eine aufgerufene Funktion wie geplant – d.h. ohne eine **Exception** – wird stets mit der nächsten Anweisung nach dem Funktionsaufruf fortgefahren.

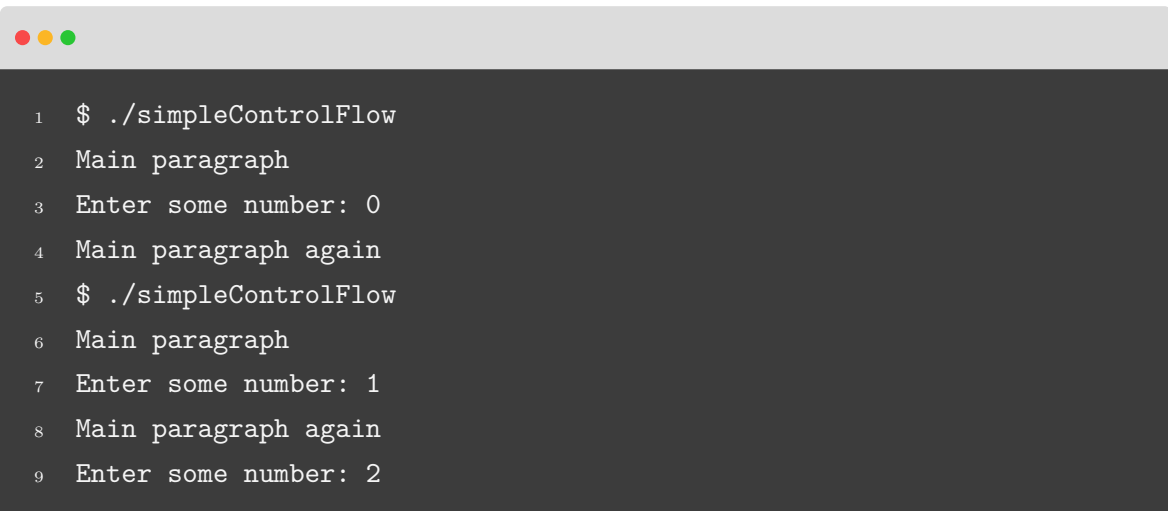
In COBOL gestaltet sich der Programmablauf ähnlich. Das Programm wird stets von oben nach unten durchlaufen. Wobei dieser lineare Ablauf z.B. durch die Verwendung von **PERFORM**-, **CALL**-, **GO TO**- oder **NEXT SENTENCE**-Anweisungen verändert werden kann.

Ein Unterprogramm wird mit **CALL** aufgerufen und gibt hingegen mit **GOBACK** die Kontrolle zurück an das aufrufende Programm. Die Ausführung eines COBOL-Programms endet beim Erreichen einer **STOP RUN**-Anweisung oder mit dem Ende des Programms (**END PROGRAM**).

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. SIMPLE-CONTROL-FLOW.
3
4      DATA DIVISION.
5      WORKING-STORAGE SECTION.
6          01 INPUT-NUMBER PIC 9.
7          88 IS-ZERO VALUE 0.
8
9      PROCEDURE DIVISION.
10     MAIN-PARAGRAPH.
11         DISPLAY "Main paragraph".
12         PERFORM SECOND-PARAGRAPH.
13         DISPLAY "Main paragraph again".
14         IF IS-ZERO THEN
15             STOP RUN
16         END-IF.
17
18     SECOND-PARAGRAPH.
19         DISPLAY "Enter some number: " WITH NO ADVANCING.
20         ACCEPT INPUT-NUMBER.
21
22     END PROGRAM SIMPLE-CONTROL-FLOW.

```



```

1  $ ./simpleControlFlow
2  Main paragraph
3  Enter some number: 0
4  Main paragraph again
5  $ ./simpleControlFlow
6  Main paragraph
7  Enter some number: 1
8  Main paragraph again
9  Enter some number: 2

```

Listing 4.9: Programmablauf in COBOL

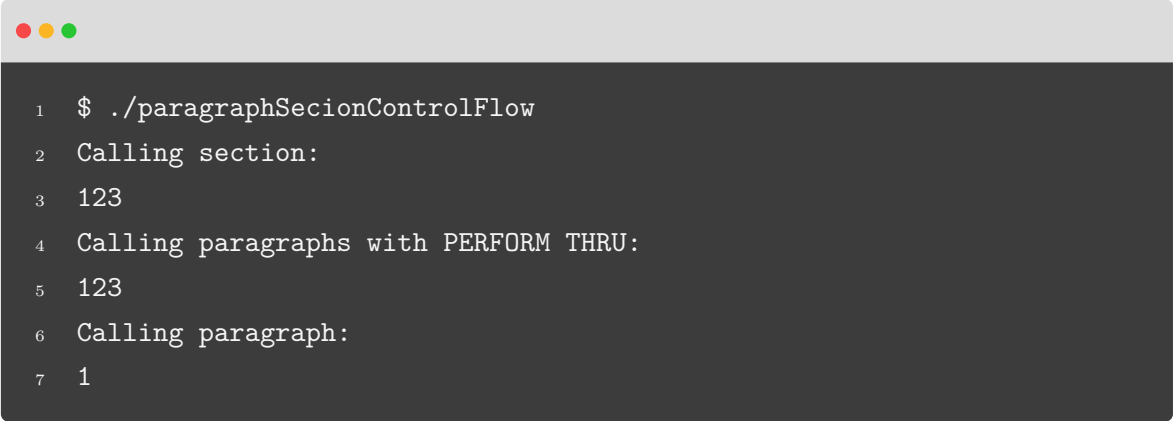
Die beiden Ausführungen von Listing 4.9 zeigen das angesprochene Verhalten eines COBOL-Programms. Beim ersten Durchlauf wird für die Variable `INPUT-NUMBER` der Wert 0 eingegeben, was durch das Ausführen der `STOP RUN`-Anweisung in Zeile 15, das Beenden des Programmes bewirkt. Beim zweiten Mal wird hingegen der Wert 1 eingegeben. Dieser Wert verhindert das Abschließen des Programms in Zeile 15, wodurch der Programmablauf in Zeile 17 fortgesetzt wird und somit erneut die Eingabeaufforderung erscheint.

Wie in Abschnitt 4.1 beschrieben besteht ein COBOL-Programm aus verschiedenen strukturellen Komponenten. Diese haben auch einen gewissen Einfluss auf den Programmablauf. Dies soll das Beispiel in Listing 4.10 veranschaulichen.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. PARAGRAPH-SECTION-CONTROL-FLOW.
3
4      PROCEDURE DIVISION.
5      MAIN-PROCEDURE.
6          DISPLAY "Calling section:".
7          PERFORM TEST-SECTION.
8          DISPLAY SPACE.
9          DISPLAY "Calling paragraphs with PERFORM THRU:".
10         PERFORM FIRST-TEST-PARAGRAPH THRU THIRD-TEST-PARAGRAPH.
11         DISPLAY SPACE.
12         DISPLAY "Calling paragraph:".
13         PERFORM FIRST-TEST-PARAGRAPH.
14         STOP RUN.
15
16     TEST-SECTION SECTION.
17     FIRST-TEST-PARAGRAPH.
18         DISPLAY "1" WITH NO ADVANCING.
19
20     SECOND-TEST-PARAGRAPH.
21         DISPLAY "2" WITH NO ADVANCING.
22
23     THIRD-TEST-PARAGRAPH.
24         DISPLAY "3" WITH NO ADVANCING.
25
26
27     END PROGRAM PARAGRAPH-SECTION-CONTROL-FLOW.

```



```

1  $ ./paragraphSecionControlFlow
2  Calling section:
3  123
4  Calling paragraphs with PERFORM THRU:
5  123
6  Calling paragraph:
7  1

```

Listing 4.10: Programmablaufunterschiede in COBOL mit Sections und Paragraphs

Wird mittels **PERFORM** eine Section aufgerufen, so werden alle Paragraphs innerhalb dieser Section der Reihe nach ausgeführt. Ruft man jedoch einen Paragraph auf, so wird nur dieser Paragraph ausgeführt. Eine weitere Möglichkeit ist die Kombination des

PERFORM mit dem **THRU**-Schlüsselwort. Hierbei werden alle Paragraphs zwischen zwei festgelegten Paragraphs ausgeführt. Der Kontrollfluss geht bei jeder Variante stets an das Statement nach dem **PERFORM**.

Um Verwirrungen vorzubeugen und lesbaren Code zu erhalten sollten alle Paragraphs stets Teil einer Section sein und auch nur diese Ziel einer **PERFORM**-Anweisung sein. Der letzte Paragraph einer Section sollte dabei immer ein **EXIT**-Paragraph sein, also nur das Schlüsselwort **EXIT** beinhalten. So ist das Ende einer Section beim lesen des Codes klar erkennbar. Außerdem stellt dieser **EXIT**-Paragraph oftmals eine Ausnahme zur Verwendung des **GO TO**-Befehls dar. Dies ist nötig, da COBOL keinen Befehl wie das **return** in Java enthält, um die Ausführungskontrolle an die aufrufende Stelle zurückzugeben. Dieses Vorgehen wurde auch von Richards bereits 1984 als best-practice beschrieben [20]. Die meisten Code-Beispiele dieser Arbeit enthalten bewusst keinen separaten **EXIT**-Paragraph, um den Umfang und die Übersichtlichkeit der Listings so gering wie möglich zu halten.

4.4.2 Verzweigungen

Eine wichtige Eigenschaft von Programmiersprachen ist konditionelle Verzweigung, also die Ausführung von Programmteilen nur unter bestimmten Voraussetzungen. Sowohl Java als auch COBOL bieten hierfür die Schlüsselwörter **if-else** (Java) bzw. **IF-ELSE-END-IF** (COBOL). Auch die Verwendung ist sehr ähnlich wie folgende Beispiele zeigen sollen.

```

1  import java.util.Scanner;
2
3  public class IfExample {
4
5      public static void main(String... args) {
6          int number = new Scanner(System.in).nextInt();
7          if (number == 0) {
8              System.out.println("Number is 0");
9              System.out.println("Number is still 0");
10         } else
11             System.out.println("Number is not 0");
12
13         System.out.println(
14             number == 0 ? "Number is 0" : "Number is not 0"
15         );

```

```

16     }
17 }

```

Listing 4.11: Verzweigung in Java

In Listing 4.11 wird anhand einer Nutzereingabe eine Fallunterscheidung bzw. Verzweigung gemacht. Dabei soll gezeigt werden, dass es möglich ist sowohl mehrere Zeilen als auch nur eine Zeile konditionell auszuführen. Soll mehr als eine Zeile untergeordnet werden, ist eine Gruppierung als Block – mit geschweiften Klammern – nötig. Der **else**-Zweig zeigt eine einzelne Anweisung als bedingt auszuführendes Statement. Das letzte Statement zeigt die Verwendung des konditionalen Operators »?«. Dabei handelt es sich lediglich um eine kurzschreibweise für ein **if-else** mit jeweils einer Anweisung.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. IF-EXAMPLE.
3      DATA DIVISION.
4      WORKING-STORAGE SECTION.
5      01 VAR PIC 9.
6
7      PROCEDURE DIVISION.
8      MAIN-PROCEDURE SECTION.
9          ACCEPT VAR.
10         PERFORM END-IF-EXAMPLE.
11         ACCEPT VAR.
12         PERFORM PERIOD-IF-EXAMPLE.
13         STOP RUN.
14
15     END-IF-EXAMPLE SECTION.
16         IF VAR = 0
17             DISPLAY "VAR = 0"
18         ELSE
19             DISPLAY "VAR != 0"
20         END-IF.
21
22     PERIOD-IF-EXAMPLE SECTION.
23         IF VAR = 0
24             DISPLAY "VAR = 0"
25         ELSE
26             DISPLAY "VAR != 0".
27
28     END PROGRAM IF-EXAMPLE.

```

Listing 4.12: Verzweigung in COBOL

Listing 4.12 bildet selbige Logik in COBOL ab. Die beiden Sections **END-IF-EXAMPLE** und **PERIOD-IF-EXAMPLE** zeigen dabei zwei unterschiedliche Wege diese zu konstruieren. Während erstere eine **ELSE**- und eine **END-IF** Anweisung nutzt, um das Konstrukt aufzubauen und zu terminieren, verwendet letztere die Eigenschaft, dass ein **IF** auch durch ein Sentenceende – siehe Abschnitt 4.1 – abgeschlossen werden kann. Dies erlaubt jedoch keine verschachtelten Verzweigungen und kann – wie die befragten Experten anmerkten – in der Praxis schnell zu Fehlern oder zumindest zu schwer durchschaubarem Verhalten führen. Herr Streit betonte, dass bestehende Programme teilweise solche Konstrukte beinhalten, ein **IF** jedoch stets mit einem **END-IF** terminiert werden sollte. Dies sorgt dafür, dass es dem Compiler möglich ist Fehler in der Verzweigung zu erkennen und eine bessere Lesbarkeit zu erreichen.

4.4.3 Schleifen

Wie in vielen anderen Sprachen unterstützen Java und COBOL auch Schleifenkonstrukte. Während Java dafür dedizierte Schlüsselwörter bereitstellt fungiert in COBOL auch dafür das **PERFORM**-Statement. Dies kann für Unklarheiten sorgen weil dieses Schlüsselwort wie später in der Arbeit beschrieben weitere Funktionen erfüllt. Jedoch soll an dieser Stelle lediglich auf die Verwendung als Schleifenkonstrukt eingegangen werden.

Java bietet mit **while**-, **do-while**- und **for**-Schleifen drei unterschiedliche Arten von Schleifen. Listing 4.13 enthält alle drei Konstrukte. Während die ersten beiden kopf- und fußgesteuert eine Bedingung überprüfen wird eine **for**-Schleife i.d.R. dazu genutzt um Werte einer bestimmten (Zahlen-)Menge zu durchlaufen.

```

1  import java.util.stream.IntStream;
2
3  public class Loops {
4
5      public static void main(String... args) {
6          int number = 0;
7          while (number < 10) {
8              number++;
9          }
10
11         number = 0;
12         do {
13             number++;
14         } while (number < 10);
15     }

```

```

16     for (number = 0; number < 10; number++) {
17         System.out.println(number);
18     }
19
20     for (Integer num : IntStream.range(0, 10).toArray()) {
21         System.out.println(num);
22     }
23 }
24 }

```

Listing 4.13: Schleifen in Java

COBOL nutzt für alle Schleifen das **PERFORM**-Schlüsselwort. In Verbindung mit weiteren Statements entstehen so unterschiedliche Schleifentypen. Listing 4.14 beschreibt die wichtigsten davon. Eine bedingte Schleifenausführung lässt sich mithilfe des **UNTIL**-Schlüsselworts und einer nachfolgenden Bedingung erreichen. Eine Zählschleife, entsprechend eines **for** in Java, kann durch **VARYING**, **FROM** und **BY** konstruiert werden. Jede Schleife kann zusätzlich durch die Angabe von **WITH TEST AFTER** von einer kopfgesteuerten zu einer fußgesteuerten Schleife gemacht werden, d.h. die Bedingung wird nach einem Schleifendurchlauf geprüft und nicht davor.

```

1     IDENTIFICATION DIVISION.
2     PROGRAM-ID. LOOP-EXAMPLE.
3     DATA DIVISION.
4     WORKING-STORAGE SECTION.
5     01 NUM PIC 9(2).
6
7     PROCEDURE DIVISION.
8     MAIN-PROCEDURE SECTION.
9         PERFORM PERFORM-UNTIL.
10        PERFORM PERFORM-VARYING.
11        PERFORM PERFORM-VARYING-TEST-AFTER.
12        STOP RUN.
13
14    PERFORM-UNTIL SECTION.
15        MOVE 0 TO NUM.
16        PERFORM UNTIL NUM = 10
17            COMPUTE NUM = NUM + 1
18        END-PERFORM.
19        DISPLAY NUM.
20
21    PERFORM-VARYING SECTION.
22        MOVE 0 TO NUM.
23        PERFORM VARYING NUM FROM 0 BY 1 UNTIL NUM = 10
24        DISPLAY NUM
25        END-PERFORM.

```



```

26
27     PERFORM-VARYING-TEST-AFTER SECTION.
28         MOVE 0 TO NUM.
29         PERFORM WITH TEST AFTER VARYING NUM FROM 0 BY 1 UNTIL NUM = 10
30             DISPLAY NUM
31         END-PERFORM.
32
33     END PROGRAM LOOP-EXAMPLE.

```

Listing 4.14: Schleifen in COBOL

4.4.4 Weitere Schlüsselwörter

Weitere Schlüsselwörter die den Kontrollfluß – vor allem im Zusammenhang mit Verzweigungen und Schleifen – in Java steuern können sind außerdem **break**, **continue** und **goto**. Zu beachten ist dabei, dass das **goto**-Schlüsselwort zwar im Sprachstandard noch definiert ist, jedoch in keiner gängigen JVM implementiert ist. Die Verwendung führt zu Fehlern beim Kompilieren. In Listing 4.15 finden sich beispielhafte Verwendungen der beiden anderen Schlüsselwörter.

```

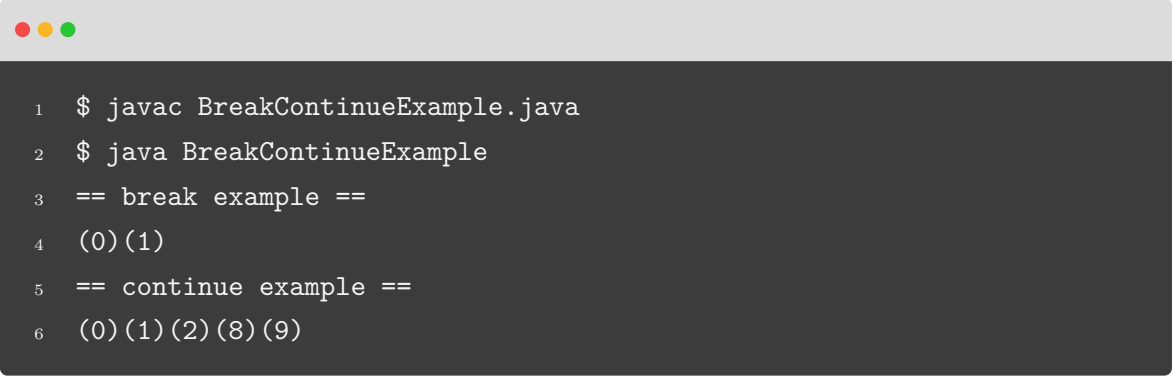
1  public class BreakContinueExample {
2
3      public static void main(String[] args) {
4          breakExample();
5          continueExample();
6      }
7
8      private static void breakExample() {
9          System.out.println("\n== break example == ");
10         for (int counter = 0; counter < 10; counter++) {
11             if (counter > 1 && counter < 8)
12                 break;
13             System.out.print("(" + counter + " ");
14         }
15     }
16
17     private static void continueExample() {
18         System.out.println("\n== continue example == ");
19         for (int counter = 0; counter < 10; counter++) {
20             if (counter > 2 && counter < 8)
21                 continue;

```

```

22         System.out.print("(" + counter + ")");
23     }
24 }
25 }

```



```

1 $ javac BreakContinueExample.java
2 $ java BreakContinueExample
3 == break example ==
4 (0)(1)
5 == continue example ==
6 (0)(1)(2)(8)(9)

```

Listing 4.15: Beispiele für die Verwendung von break und continue in Java

Ein einfaches **break** sorgt wie gezeigt dafür, dass die direkt umfassende Schleife verlassen wird. Auch ein simples **continue** hat Auswirkungen auf die direkt beinhaltende Schleife. So sorgt es dafür, dass der aktuelle Schleifendurchlauf abgebrochen und mit dem nächsten fortgefahren wird. Unterabschnitt 4.9.2 zeigt eine weitere Verwendung des **break**-Statements. Deutlich unüblicher – jedoch nicht weniger relevant – ist der Gebrauch eines Labels in Java. Dieses Label kann in der Verbindung mit einer **break**- oder **continue**-Anweisung genutzt werden, um mehrere umfassende Schleifen verlassen bzw. um mit dem nächsten Schleifendurchlauf einer weiter außen befindlichen Schleife fortgefahren zu werden. Die Anweisung betrifft dabei die Schleife, welche das Label trägt.

In COBOL ist ebenfalls das Schlüsselwort **CONTINUE** vorhanden. Allerdings ist hierbei Vorsicht geboten, da dieses abweichende Bedeutung vom gleichnamigen Java-Schlüsselwort hat. Während in Java, wie erwähnt, zum nächsten Schleifendurchlauf gesprungen werden kann, entspricht dieses Schlüsselwort in COBOL lediglich einer Anweisung bei der nichts ausgeführt wird. Dies ist in der Praxis häufig zu beobachten, um z.B. Verzeigungsteile leer zu lassen ohne die Bedingung negieren zu müssen.

Neben diesem ist **NEXT SENTENCE** ein Schlüsselwort das häufig in älterem Code zu finden sei, wie Herr Lamperstorfer bestätigte. Dieses kann dazu genutzt werden, um den aktuellen Sentence zu verlassen und mit der Anweisung die darauf folgt fortzufahren. Zu beobachten sei die Verwendung auch häufig zur Negation einer Bedingung,

indem der **IF**-Zweig lediglich dieses Statement enthält und der **ELSE**-Zweig die Logik bei nichtzutreffen der Bedingung enthält. Diese Konstrukte sollten jedoch vermieden werden und durch ein einfach Negieren mit **NOT** geschrieben bzw. ersetzt werden. Aus Gründen der Unübersichtlichkeit ist dieses Schlüsselwort in GnuCOBOL standardmäßig verboten.

Seltener zu finden ist dagegen die **EXIT PERFORM**-Anweisung. Diese kann innerhalb von Schleifen dazu genutzt werden, um, wie mit einem **break** in Java, die umgebende Schleife zu verlassen oder durch **EXIT PERFORM CYCLE**, wie mit einem **continue** in Java, mit dem nächsten Schleifendurchlauf fortzufahren. Dies soll Listing 4.16 verdeutlichen.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. EXIT-PERFORM.
3      DATA DIVISION.
4      WORKING-STORAGE SECTION.
5      01 NUM PIC 9(2).
6
7      PROCEDURE DIVISION.
8      MAIN-PROCEDURE SECTION.
9          PERFORM EXIT-PERFORM.
10         PERFORM EXIT-PERFORM-CYCLE.
11         STOP RUN.
12
13     EXIT-PERFORM SECTION.
14         MOVE 0 TO NUM.
15         PERFORM VARYING NUM FROM 0 BY 1 UNTIL NUM = 10
16             EXIT PERFORM
17             DISPLAY "This is omitted!"
18         END-PERFORM.
19         DISPLAY NUM.
20
21     EXIT-PERFORM-CYCLE SECTION.
22         MOVE 0 TO NUM.
23         PERFORM VARYING NUM FROM 0 BY 1 UNTIL NUM = 10
24             EXIT PERFORM CYCLE
25             DISPLAY "This is omitted!"
26         END-PERFORM.
27         DISPLAY NUM.
28
29     END PROGRAM EXIT-PERFORM.
```



```

1  00
2  10
```

Listing 4.16: **EXIT PERFORM** in COBOL

An dieser Stelle sei ausdrücklich erwähnt, dass die Verwendung des **GO TO**-Befehls – abgesehen von oben genanntem Einsatz als *return*-Ersatz innerhalb einer Section – in COBOL unterlassen werden sollte, oftmals sogar durch projekt- oder unternehmensspezifische Vorgaben verboten ist, da ansonsten sehr schwer verständlicher und wartbarer Code entstehen kann. Leider findet man sich in der Praxis oftmals mit Code konfrontiert, der **GO TO**-Befehle zur Steuerung des Ablaufs verwendet. Sogar Schleifenkonstrukte sind in älteren Programmen oft damit realisiert, worauf Herr Streit hinwies.

4.4.5 Ausnahmebehandlung

In modernen Sprachen sind Ausnahmebehandlungsmechanismen vorhanden, um die Steuerung des Kontrollflusses klar von der Fehlerbehandlung zu trennen. So wird zum einen eine übersichtlichere Implementierung erlaubt, aber auch erreicht, dass bereits der Compiler auf Fehler hinweisen kann die bei der Ausführung auftreten können bzw. gänzlich das kompilieren bei ungenügender Fehlerbehandlung verweigert.

Java bietet dabei das Konzept der **Exceptions**. Diese lassen sich in sogenannte *checked* und *unchecked-Exceptions* unterteilen. Während *checked-Exception* stets einer ausreichenden Fehlerbehandlung im Code bedürfen und ansonsten zu Fehlern des Kompilervorgangs führen, können *unchecked-Exceptions* unbehandelt gelassen werden. Von der genauen Verwendungserklärung sei an dieser Stelle abgesehen und lediglich auf weiterführende Literatur wie *Java for COBOL Programmers* [4] von Byrne und Cross verwiesen.

COBOL bietet zur generellen Ausnahmebehandlung keine Methodik. Fehlerfälle müssen über Variablenwerte signalisiert, geprüft und entsprechend behandelt werden. Herr Streit wies darauf hin, dass eine ungenügende Prüfung hierbei zum kompletten Absturz des Programms führen kann. Jedoch ist es möglich vordefinierte Fehler bei Berechnungen oder String-Zuweisungen abzufangen und darauf zu reagieren wie Listing 4.17 zeigt. Dazu können **ON SIZE ERROR** und **ON OVERFLOW** genutzt werden.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. IF-EXAMPLE.
3      DATA DIVISION.
4      WORKING-STORAGE SECTION.
5      01 NUM PIC 9(2).
6      01 TARGET-STRING PIC X(5).
7
8      PROCEDURE DIVISION.
9      MAIN-PROCEDURE SECTION.
10     ACCEPT NUM.
```

```
11      COMPUTE NUM = NUM * 2
12      ON SIZE ERROR DISPLAY "The value could not be doubled."
13      STOP RUN
14  END-COMPUTE.
15  DISPLAY "Doubled number: " NUM.
16
17  STRING 'This is a little too long' DELIMITED BY SIZE
18  INTO TARGET-STRING
19  ON OVERFLOW DISPLAY "The string is too long!".
20
21  STOP RUN.
22
23  END PROGRAM IF-EXAMPLE.
```

Listing 4.17: Rudimentäre Fehlerbehandlung in COBOL

4.4.6 Nebenläufigkeit

Als letzter wichtiger Punkt muss an dieser Stelle erwähnt werden, dass es in Java möglich und auch üblich ist, Programme nebenläufig zu entwickeln. Das heißt mehrere Threads arbeiten parallel und führen Verarbeitungen – je nach Hardware nur scheinbar – gleichzeitig aus. Dabei muss der Entwickler auf die Synchronisation von gemeinsam genutzten Speicherbereichen achten, um gültige Daten zu gewährleisten. Diese Nebenläufige Programmierung birgt zwar ein gewisses Fehlerpotential bei der Implementierung, sorgt jedoch dafür, dass Logik tendenziell effizienter ausgeführt wird.

In COBOL ist diese nebenläufige Ausführung nicht möglich. Ein COBOL-Programm führt Verarbeitungsschritte stets sequenziell aus und erlaubt keine parallelen Ausführungen. Über einen Transaktionsmonitor ist es jedoch teilweise möglich, dass verschiedene Programme gleichzeitig ausgeführt werden. Jedoch haben diese keine Kenntnis von anderen ausgeführten Programmen. Teilweise können verschiedene Programme auch die gleichen Speicherbereiche reservieren und so quasi miteinander arbeiten. Jedoch gibt es in COBOL keine Möglichkeit zur Synchronisation, weshalb dieses Vorgehen nur sehr selten beobachtet werden kann. Bei einem Transactionsmonitor handelt es sich um eine Art Middleware, vergleichbar zu Application-Servern in Java, welche Anfragen entgegennimmt, dafür sorgt, dass Ressourcen geöffnet und aufgeräumt werden, auf Hostsystemen Terminal-Masken zur Verfügung stellt und wie erwähnt entscheidet wie viele und welche Programme parallel ausgeführt werden. Ein Beispiel hierfür ist das

Customer Information Control System kurz *CICS*. Um aus einem COBOL-Programm Teile des Transaktionsmonitors aufzurufen gibt es Befehle wie `EXEC CICS`.

4.5 Funktionen, Unterprogramme und Rückgabewerte

Ein wichtiger Bestandteil von vielen Programmiersprachen sind Prozeduren und Funktionen. Dabei handelt es sich um Codeabschnitte, die von einer anderen Stelle aus aufgerufen werden können. Im Gegensatz zu Prozeduren, die bestimmte Verarbeitungsschritte durchlaufen, liefern Funktionen dabei noch zusätzlich einen Rückgabewert.

Java

In Java muss jede Anweisung Teil einer Funktion sein. Wie in Abschnitt 4.4 beschrieben startet ein Java-Programm auch innerhalb einer Funktion. Die Unterscheidung zwischen Funktion und Prozedur wird in Java gängigerweise nicht getroffen. Oft findet sich auch die Bezeichnung Methode. Java unterscheidet sich von manchen anderen modernen Sprachen dadurch, dass es keine Methodenschachtelungen erlaubt.

```
1  public class MethodExample {
2
3      public void printGreeting(String greeting) {
4          System.out.println(greeting);
5      }
6
7      public String getGreeting(String firstName, String surname) {
8          return String.format("Hello %s %s!", firstName, surname);
9      }
10
11     public void process() {
12         String greeting = getGreeting("Max", "Mustermann");
13         printGreeting(greeting);
14     }
15 }
```

Listing 4.18: Methoden in Java

Listing 4.18 zeigt verschiedene Methoden. An diesem Beispiel sollen zwei wichtige Konzepte dargestellt werden:

- **Übergabe von Parametern** Wie gezeigt erhalten die Funktionen unterschiedliche Parameter. Die Methode `process()` zum Beispiel erhält keinen Parameter, wohingegen `getGreeting(String, String)` zwei Parameter vom Typ `String` erwartet.
- **Rückgabe eines Wertes** Eine Funktion muss stets den Typ ihres Rückgabewertes definieren. Soll kein Rückgabewert geliefert werden so ist der Typ als `void` zu definieren. Im Gegensatz zu anderen Sprachen kann eine Methode in Java lediglich einen Wert zurückliefern.

Der Aufruf einer Funktion erfolgt wie in Zeile 12 gezeigt mittels Methodenname und den zu übergebenen Parametern. Außerdem zeigt sich, dass Funktionen mithilfe des `return`-Statements **einen** Rückgabewert an die aufrufende Funktion zurückgeben können. Ein `return` kann auch dazu benutzt werden um Funktionen an anderen Stellen als der letzten Zeile zu verlassen. Eine Funktion ohne Rückgabewert – `void` – terminiert implizit in der letzten Zeile und benötigt kein explizites `return`, kann dieses jedoch auch ohne Rückgabewert nutzen um vorher den Funktionsablauf zu beenden. Eine Methode mit Rückgabewert muss stets einen Wert mit passendem Datentyp durch `return` zurückgeben.

Eine eher selten genutzte, wenn doch elegante Möglichkeit die sich mit Funktionen in Java bietet, ist die Rekursion. Dabei handelt es sich um eine Funktion die sich selbst aufruft.

```
1  public class RecursionExample {
2
3      public static int facultyRecursive(int number) {
4          if (number <= 1)
5              return 1;
6          return number * facultyRecursive(number - 1);
7      }
8
9      public static int facultyIterative(int number) {
10         int product = 1;
11         while (number > 1) {
12             product = product * number;
13             number--;
14         }
```

```
15         return product;  
16     }  
17 }
```

Listing 4.19: Rekursion in Java

Die Funktionen `facultyRecursive` und `facultyIterative` in Listing 4.19 berechnen die Fakultät einer übergebenen Zahl. Klar ersichtlich ist, dass auch schon ein sehr kleines Beispiel durch eine rekursive Implementierung eleganter und durch weniger Code ausgedrückt werden kann. Auf der anderen Seite ist die iterative Implementierung sicherer, da die rekursive Variante unter Umständen an die nicht fest definierte Grenze der maximalen Rekursionstiefe gelangt.

COBOL

Das Konzept einer Funktion existiert in COBOL nicht. Lediglich das Aufrufen einer Section oder eines Paragraphs mithilfe eines **PERFORM** geben ansatzweise ähnliche Möglichkeiten und können daher als Vergleich herangezogen werden. Jedoch können dabei weder Parameter übergeben noch ein Wert zurückgeliefert werden. Darum ist es nötig Werte, die innerhalb einer SECTION verwendet werden sollen in Variablen zu kopieren. Auch ein Wert, der zurückgeliefert werden soll muss in eine solche Variable kopiert werden. Wie in ?? beschrieben sind diese Variablen jedoch immer global innerhalb eines Programms definiert.

Oft bringt das allerdings Probleme mit sich. Zum Beispiel werden in der Praxis oft Variablen, die für etwas anderes gedacht sind, an einer anderen Stelle wiederverwendet. So ist nicht ganz klar, welchen Zweck Variablen erfüllen. Ein weiterer großer Nachteil ist, dass Logik oftmals kopiert und sehr ähnlich nochmals geschrieben werden muss, um auf anderen Daten zu operieren.

Rekursive **PERFORM**-Aufrufe sind zwar syntaktisch möglich, jedoch führt die Ausführung zu einem undefinierten Verhalten des Programms und ist deshalb in jedem Fall zu unterlassen. Es kann quasi festgehalten werden, dass Rekursionen innerhalb eines Programms in COBOL nicht möglich sind. Anders sieht es dabei mit gesamten Programmen aus.


```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. FACULTY RECURSIVE.
3
4      DATA DIVISION.
5      WORKING-STORAGE SECTION.
6          01 WS-NUMBER PIC 9(4) VALUE 5.
7          01 WS-PRODUCT PIC 9(4) VALUE 0.
8      LOCAL-STORAGE SECTION.
9          01 LS-NUMBER PIC 9(4).
10
11     PROCEDURE DIVISION.
12         IF WS-NUMBER = 0
13             MOVE 1 TO WS-PRODUCT
14         ELSE
15             MOVE WS-NUMBER TO LS-NUMBER
16             COMPUTE WS-NUMBER = WS-NUMBER - 1
17             CALL "FACULTY"
18             COMPUTE WS-PRODUCT = LS-NUMBER * WS-PRODUCT
19         END-IF.
20         IF LS-NUMBER = 5
21             DISPLAY WS-PRODUCT
22         END-IF.
23         GOBACK.
24
25     END PROGRAM FACULTY.

```

Listing 4.20: Rekursion in COBOL

Listing 4.20 enthält analog zu gezeigtem Java-Beispiel auch ein Programm, welches Rekursiv die Fakultät einer Zahl errechnet und ausgibt. Wichtig ist hierbei vor allem die **RECURSIVE** Definition hinter dem Programmnamen in Zeile 2. Die **WORKING-STORAGE SECTION** enthält dabei Variablen, welche von jeder Instanz des rekursiv aufgerufenen Programms gemeinsam genutzt werden. In **LOCAL-STORAGE SECTION** finden sich Variablen, deren Gültigkeitsbereich sich auf die aktuelle Aufrufinstanz beschränken.

In COBOL ist es, anders als in Java, auch möglich eigenständige Unterprogramme aufzurufen. Wie bereits in Abschnitt 4.1 beschrieben dient die **LINKAGE SECTION** dazu im Unterprogramm zu definieren, welche Variablen übergeben werden. Mithilfe der **CALL**-Anweisung und des Programmnamens können. Wie in Abschnitt 3.3 kann die Angabe des Programmnamens auch pseudo-variabel geschehen um ein statisches Linken der Programmteile zu vermeiden.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. CALLING_PROGRAM.
3      DATA DIVISION.
4      WORKING-STORAGE SECTION.

```

```

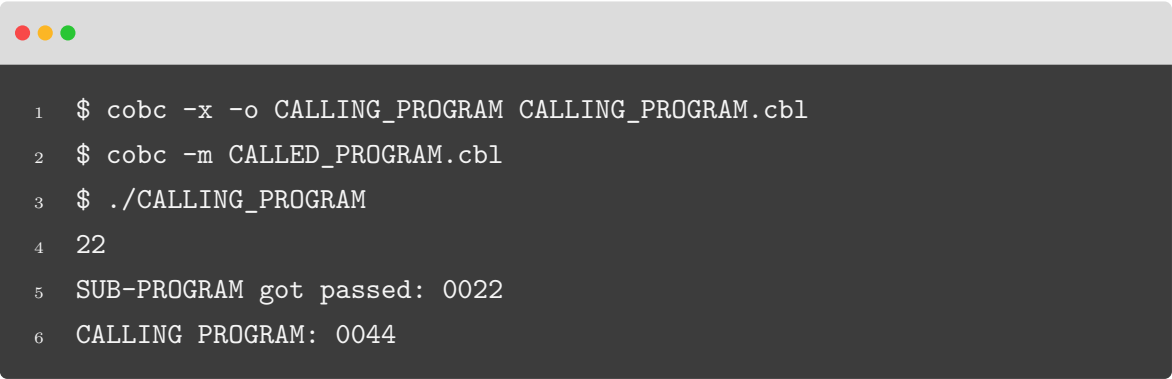
5          01 SUB-PROGRAM-PARAMS.
6              05 INPUT-NUMBER PIC 9(4).
7              05 RET-VALUE PIC 9(4).
8
9  PROCEDURE DIVISION.
10 MAIN-PROCEDURE.
11     ACCEPT INPUT-NUMBER.
12     CALL "CALLED_PROGRAM" USING SUB-PROGRAM-PARAMS.
13     DISPLAY "CALLING PROGRAM: " RET-VALUE.
14     STOP RUN.
15
16 END PROGRAM CALLING_PROGRAM.

```

```

1  IDENTIFICATION DIVISION.
2  PROGRAM-ID. CALLED_PROGRAM.
3  DATA DIVISION.
4  LINKAGE SECTION.
5      01 SUB-PROGRAM-PARAMS.
6          05 INPUT-NUMBER PIC 9(4).
7          05 RET-VALUE PIC 9(4).
8
9  PROCEDURE DIVISION USING SUB-PROGRAM-PARAMS.
10 MAIN-PROCEDURE.
11     DISPLAY "SUB-PROGRAM got passed: " INPUT-NUMBER.
12     COMPUTE RET-VALUE = INPUT-NUMBER * 2.
13     GOBACK.
14
15 END PROGRAM CALLED_PROGRAM.

```



```

1  $ cobc -x -o CALLING_PROGRAM CALLING_PROGRAM.cbl
2  $ cobc -m CALLED_PROGRAM.cbl
3  $ ./CALLING_PROGRAM
4  22
5  SUB-PROGRAM got passed: 0022
6  CALLING PROGRAM: 0044

```

Listing 4.21: Unterprogramme in COBOL

Listing 4.21 zeigt den Aufruf eines Unterprogramms und die Übergabe von Parametern. Diese Parameter werden per Referenz übergeben. Das heißt, dass Unterprogramme stets auf den gleichen Speicherbereich zugreifen wie das ursprüngliche Program und

so auch dessen Daten verändern. Dies gilt es zu beachten, da dadurch ungewünschte Seiteneffekte oder gar Fehler auftreten können. Das Definieren eines Rückgabewertes in einem Unterprogramm ist nicht möglich. Soll also ein Rückgabewert im aufgerufenen Programm gesetzt werden, so ist dieser innerhalb der übergebenen Datenstruktur zu setzen, auf die wie erwähnt auch das aufrufende Programm zugriff hat.

4.6 Dateien

Wie in Abschnitt 3.5 beschrieben wurde, stellen oftmals auch Dateien eine wichtige Datenressource dar. An dieser Stelle soll der Umgang – das Lesen und Schreiben – mit Dateien in Java und COBOL erläutert und gegenübergestellt werden.

Java bietet bereits mit Bibliotheksfunktionen des JDK umfangreiche Möglichkeiten Dateien zu lesen und zu schreiben. Dies geschieht dabei in der Regel zeilenweise, wobei auch byte- bzw. zeichenweises Lesen möglich ist. Das Beispiel Listing 4.22 zeigt dabei zusätzlich die Verwendung der Klasse `InputStream`. Diese sorgt dafür, dass die Datei nicht auf einmal in den Speicher geladen wird, sondern nur die gelesenen Daten im Speicher gehalten werden. Dies ist essenziell, um große Dateien zu lesen, da der Speicher des Systems ohne eine solche Methodik möglicherweise nicht ausreichend wäre um die gesamte Datei zu speichern und so eine `Exception` auftreten würde.

```

1  import java.io.BufferedReader;
2  import java.io.BufferedWriter;
3  import java.io.IOException;
4  import java.io.InputStream;
5  import java.io.InputStreamReader;
6  import java.nio.charset.Charset;
7  import java.nio.charset.StandardCharsets;
8  import java.nio.file.FileSystems;
9  import java.nio.file.Files;
10 import java.nio.file.Path;
11
12 public class FileInputOutput {
13
14     private final static Path filePath =
15         ↪ FileSystems.getDefault().getPath("/home/toni/Temporary/output.txt");
16
17     public static void main(String... args) {
18         writeFile();
19         readFile();
20     }
21 }

```

```

19     }
20
21     private static void writeFile() {
22         Charset charset = StandardCharsets.UTF_8;
23         String s = "I'm getting written to the File\n";
24
25         try (BufferedWriter writer = Files.newBufferedWriter(filePath,
26             ↪ charset)) {
27             writer.write(s, 0, s.length());
28         } catch (IOException x) {
29             System.err.format("IOException: %s\n", x);
30         }
31
32     private static void readFile() {
33         try (InputStream in = Files.newInputStream(filePath);
34             BufferedReader reader = new BufferedReader(new
35             ↪ InputStreamReader(in))) {
36             String line = null;
37             while ((line = reader.readLine()) != null) {
38                 System.out.println(line);
39             }
40         } catch (IOException x) {
41             throw new RuntimeException(x);
42         }
43     }

```

Listing 4.22: Datei-Ein- und Ausgabe in Java [18]

In Java sind darüberhinaus viele Bibliotheken erhältlich, die das Parsen von bestimmten, standardisierten Dateiformaten erleichtern können. Jedoch erfordert es ohne diese Bibliotheken stets eigene Implementierungen, da das JDK an dieser Stelle nicht viel mehr als die gezeigten Möglichkeiten bietet.

In COBOL hingegen wird der Zugriff auf Dateien auf Basis sogenannter *Records* bewerkstelligt. Dies entspricht weitestgehend dem gezeigten Java-Beispiel, jedoch hat der Entwickler hier die Möglichkeit zu definieren, wie eine Zeile der Datei aufgebaut ist. Dies setzt zwar voraus, dass der Dateiinhalte in einer Art Tabelle formatiert ist, sorgt allerdings dafür, dass kein Mehraufwand beim Parsen nötig ist. Listing 4.25 und ?? zeigen diese Verwendung der Datei-Ein- und Ausgabe in COBOL.

```

1  26 Monika Hofmann
2  21  Marc  Bauer
3  45 Sophia  Maier

```

Listing 4.23: Eingabedatei recordFile.txt

```

1      01 PERSON.
2          88 EOF  VALUE HIGH-VALUES.
3          05 AGE PIC 9(2).
4          05 FILLER PIC X.
5          05 FIRSTNAME PIC X(6).
6          05 FILLER PIC X.
7          05 SURNAME PIC X(7).

```

Listing 4.24: x

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. FILES.
3      ENVIRONMENT DIVISION.
4      INPUT-OUTPUT SECTION.
5      FILE-CONTROL.
6      SELECT RecordFile
7          ASSIGN TO "recordFile.txt"
8          ORGANIZATION IS LINE SEQUENTIAL.
9
10     DATA DIVISION.
11     FILE SECTION.
12     FD RecordFile.
13     COPY "PersonData".
14
15     PROCEDURE DIVISION.
16     MAIN-PROCEDURE.
17         PERFORM READ-FILE.
18
19     READ-FILE SECTION.
20         OPEN INPUT RecordFile.
21         PERFORM UNTIL EOF
22             READ RecordFile
23             AT END SET EOF TO TRUE
24         END-READ
25         IF EOF
26             EXIT PERFORM CYCLE
27         END-IF
28         DISPLAY "Age: " AGE SPACE "Name: " SURNAME ", " FIRSTNAME
29     END-PERFORM.
30     CLOSE RecordFile.
31
32     END PROGRAM FILES.

```

Listing 4.25: Datei-Ausgabe in COBOL [18]

complete
section

4.7 Generische Programmierung

Moderne Programmiersprachen wie Java oder C# erlauben eine Programmierung mit sogenannten *Generics*. Dabei handelt es sich um ein Konzept, bei dem Variablentypen generisch sein können, solange sie bestimmte Eigenschaften erfüllen. Diese Eigenschaften werden durch das Implementieren eines bestimmten Interfaces oder durch das Erben von einer bestimmten Klasse beschrieben. Dies soll Listing 4.26 verdeutlichen. Hieran wird auch deutlich, dass sowohl für Interfaces als auch für Oberklassen stets das Schlüsselwort **extends** verwendet werden muss.

```

1  import java.io.Serializable;
2
3  public class GenericExample {
4
5      public static void main(String... args) {
6          GenericClass<Exception, Float> genericObject = new GenericClass<>();
7
8          genericObject.decorate(genericObject);
9      }
10
11     static class GenericClass<T extends Serializable, S extends Number> {
12
13         T serializableObject;
14         S someNumber;
15
16         public <U> String decorate(final U someObject) {
17             return "~" + someObject.toString() + "~";
18         }
19     }
20
21 }
```

Listing 4.26: Generics in Java

In diesem Beispiel werden die drei generischen Typen *S*, *T* und *U* definiert. *T* muss dabei die Eigenschaft erfüllen das **Serializable** Interface zu implementieren. *S* muss eine Unterklasse von **Number** sein. Für *U* hingegen wird keine bestimmte Eigenschaft definiert. Das bedeutet, dass an dieser Stelle jede Klasse welche von der Klasse **Object** erbt – in Java also **jede** Klasse – verwendet werden kann.

Eine weitere Eigenschaft, die gezeigt werden soll ist, dass es sowohl generische Klassen als auch generische Methoden geben kann. Generische Typen die für eine Klasse definiert sind stehen in der gesamten Klasse zur Verfügung, müssen jedoch bereits beim instanzieren festgelegt werden. Generische Methoden hingegen definieren generische Typen nur für den eigenen Scope.

Dieses Konzept sorgt dafür, dass Algorithmen implementiert und als Bibliotheken bereitgestellt werden können, ohne Kenntnis über die tatsächlich verwendeten Datentypen zu haben, was die in Abschnitt 3.3 angesprochenen Modularisierungsmöglichkeiten unterstützt. Beispielsweise kann ein Sortieralgorithmus implementiert werden, welche generische Objekte entgegennimmt, die das **Comparable**-Interface – ein Java Interface, welches dafür sorgt, dass zwei Objekte in eine Größenbeziehung gesetzt werden können – implementieren. Folglich können mit diesem Algorithmus alle Objekte sortiert werden, die das Interface implementieren. Dieses Konzept ist in der objektorientierten Programmierung grundlegend und trägt maßgeblich zur Wiederverwendung und Kapselung bei.

4.8 Konventionen

Da bei Programmen oft eine Vielzahl von Entwicklern tätig ist und sich das Entwicklerteam auch über die Zeit ändern kann ist es nötig, dass alle Entwickler einen ähnlichen Stil verfolgen, sodass das Zurechtfinden in Code eines anderen Programmiers erleichtert wird. Diese Konventionen können zwar auch auf Projektebene festgelegt werden, jedoch entwickeln sich in Programmiersprachen oft Konventionen welche sich projekt-, personen- und unternehmensunabhängig etablieren. In diesem Kapitel werden die wichtigsten dieser Konventionen beschrieben.

4.8.1 Groß- und Kleinschreibung

Bei der Programmierung behilft man sich oftmals der Groß- und Kleinschreibung, um ein höheres Maß an Struktur und Lesbarkeit des Codes zu erreichen.

Java

Wichtig ist hierbei vorneweg zu beachten, dass Java »case-sensitive« ist, also zwischen Groß- und Kleinbuchstaben unterscheidet, während COBOL »case-insensitive« ist, also – außer in Strings – keinen Unterschied macht. In Java ist es üblich Methodennamen und veränderbare Variablen mit einem kleinen Buchstaben beginnend zu benennen. Besteht der Name aus mehreren Wörtern so wird dieser im »camel-case« geschrieben. Das heißt, dass stets Großbuchstaben für den Beginn eines neuen Wortes verwendet werden. Beispiele hierfür wären `getAdditionalData()` oder `int currentAmountOfMoney`. Klassennamen werden in dem selben Muster geschrieben, beginnen jedoch mit einem Großbuchstaben: `ToolBox`. Zu guter letzt sollten konstante Variablen und Werte innerhalb eines Aufzählungstyps durchgehend aus Großbuchstaben bestehen, wobei einzelne Wörter mit einem Unterstrich voneinander getrennt werden (`final int MULTIPLY_FACTOR = 2`).

COBOL

COBOL hingegen unterscheidet bei Variablennamen und Schlüsselwörtern nicht zwischen Groß- und Kleinschreibung. Es ist jedoch üblich sowohl Schlüsselwörter als auch Variablennamen komplett groß zu schreiben. Dies rührt daher, dass COBOL aus der Lochkartenzeit stammt, die meist lediglich Großbuchstaben im Zeichensatz hatten. Auch frühe Hostsysteme auf denen COBOL entwickelt und ausgeführt wurden, arbeiteten meist nur mit Großschreibung, sodass Programme auch über die Zeit hinweg nur so geschrieben wurden.

4.8.2 Affixe

Ein weiteres wichtiges Werkzeug bei der Strukturierung von Programmcode ist das Versehen mit Affixen (Prä- oder Suffixen).

Java

In Java ist es hierbei nicht ratsam Affixe zu verwenden. Da diese jedoch in einigen bestehende Codebasen Verwendung finden, werden an dieser Stelle die üblichsten behandelt. Oftmals werden Interfaces in Java mit einem vorangestellten »I« gekennzeichnet. Diese Konvention sorgt jedoch dafür, dass Implementierungen eines Interfaces namentlich nicht immer klar abgegrenzt und definiert sind. Wird beispielsweise ein Interface `IOutput` definiert so könnten valide Implementierungen `ConsoleOutput` oder `PrinterOutput` sein. Jedoch erlaubt dies namentlich auch die Interface-Implementierung namens `Output`, bei der nicht ausreichend klar ist was der Zweck dieser Klasse ist. Ein weiterer Codingstil der gelegentlich angewendet wird ist das Nutzen von »m« und »s« als Präfix von Variablen. Diese sollen kennzeichnen, dass eine Variable entweder Instanzvariable (**m**ember) oder statisch (**s**tatic) ist. Durch die Verwendung von modernen IDEs, die beide farblich unterschiedlich darstellen, und des Schlüsselwortes `this`, welches exakt für Referenzen auf Instanzvariablen gedacht ist, werden Variablen jedoch bereits ausreichend gekennzeichnet, sodass der Code durch die Verwendung dieser Präfixe unnötigerweise schwerer lesbar gemacht wird.

COBOL

In COBOL ist es in der Praxis dagegen sehr sinnvoll Affixe zu verwenden. Dadurch, dass es nicht möglich ist lokale Variablen zu definieren erlauben es Präfixe schnell und übersichtlich kenntlich zu machen, welche Variablen zu welchem Programmteil gehören. Dabei handelt es sich um eine Best-Practice-Methode, um den Code verständlicher und leichter lesbar zu machen, wie *Herr Streit* im Fachinterview betonte, auch wenn in der Praxis oftmals darauf verzichtet wird.

So kann man den Namen einer Section oder eines Paragraph mit einem Präfix versehen und die darin genutzten Variablen mit demselbigen kennzeichnen. Beispielsweise sollte eine Variable `PC-100-VALUE` nur in der Section `PC-100-PROCESS` verwendet werden.

Oft finden sich in bestehendem Code auch Präfixe welche die Art des Speichers – z.B. `WS-` für `WORKING-STORAGE` – kennzeichnen. Eine Benennung nach den genutzten Programmteilen ist jedoch vorzuziehen und sorgt für klarere Struktur und Lesbarkeit.

4.8.3 Schlüsselwörter

In COBOL ist es möglich bestimmte Schlüsselwörter zu verkürzen. Dabei kann jedoch nicht beliebig gekürzt werden. Es sind lediglich weitere Schlüsselwörter mit selber Funktion definiert, die genutzt werden können. Ein Beispiel dafür ist die **PICTURE**-Anweisung, die auch als **PIC** geschrieben werden kann. Dabei handelt es sich zwar nicht um eine Konvention im eigentlichen Sinne, jedoch findet sich in der Praxis oftmals COBOL-Code welcher gekürzte Schlüsselwörter verwendet.

In Java ist ein verkürzen von Schlüsselwörtern hingegen nicht möglich.

4.8.4 Formatierung des Quelltextes

Neben den in Abschnitt 4.1 beschriebenen fest vorgegebenen strukturellen Eigenschaften von Java- und COBOL-Programmen, lassen sich auch Konventionen beschreiben, die das Formatieren des Quelltextes betreffen.

Für COBOL und Java gilt hierbei gleichermaßen, dass pro Zeile genau ein Statement stehen sollte. Alles innerhalb eines Blocks – also eine oder mehrere untergeordnete Anweisungen – werden konventionell um einen Tabulatorsprung eingerückt. Außerdem sollten zwischen Blöcken und Anweisungen Leerzeilen eingefügt werden, die die Lesbarkeit erhöhen. In COBOL ist jedoch bei der Einrückung das bereits beschriebene Spaltenprinzip zu beachten.

4.9 Weitere Sprachkonzepte

4.9.1 Benannte Bedingungen

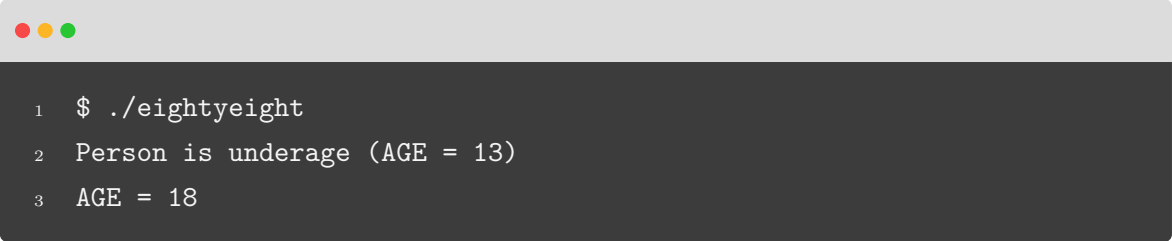
Neben den bereits angesprochenen Stufennummern stellt die *88* eine weitere Besonderheit in COBOL dar. Mit ihr ist es möglich einer Variable einen Wahrheitswert zuzuweisen, der von einem anderen Variablenwert abhängt. Es entsteht eine sogenannte benannte Bedingung.

Listing 4.27 zeigt die Verwendung der Stufennummer 88. Die Variable `AGE` kann dabei zweistellige numerische Werte enthalten die einem beispielhaften Alter entsprechen, welches zu Beginn mit 13 vorbelegt wird. Liegt der Wert zwischen 0 und 17 (`VALUE 0 THRU 17`) so weisen die Variablen `ISUNDERAGE` den Wahrheitswert `TRUE` und `ISADULT` den Wahrheitswert `FALSE` auf.

Der so entstandene Wahrheitswert kann folglich immer dann verwendet werden, wenn getestet werden soll, ob die Variable `AGE` im Bereich zwischen 0 und 17 bzw. zwischen 17 und 99 liegt. Also um zu testen, ob das Alter einer minder- oder volljährigen Person entspricht.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. EIGHTYEIGHT.
3
4      DATA DIVISION.
5      WORKING-STORAGE SECTION.
6          01 AGE PIC 9(2) VALUE 13.
7          88 ISUNDERAGE VALUE 0 THRU 17.
8          88 ISADULT VALUE 18 THRU 99.
9
10     PROCEDURE DIVISION.
11     MAIN.
12         IF ISUNDERAGE THEN
13             DISPLAY 'Person is underage (AGE = 'AGE')'
14         ELSE
15             DISPLAY 'Person is adult'
16         END-IF.
17         SET ISADULT TO TRUE
18         DISPLAY 'AGE = ' AGE.
19         STOP RUN.
20
21     END PROGRAM EIGHTYEIGHT.
```



```

1  $ ./eightyeight
2  Person is underage (AGE = 13)
3  AGE = 18
```

Listing 4.27: Beispiel für COBOL Stufennummer 88

Zeile 14 des Programms illustriert einen weiteren Anwendungsfall der benannten Bedingungen. So lässt sich der Wert der eigentlichen Variable setzen, indem der bedingten Variable der Wahrheitswert `TRUE` zugewiesen wird. Das Ergebnis dieser Zuweisung wird

in der Ausgabe von Listing 4.27 in Zeile 3 dargestellt. Zu beachten ist hierbei, dass die meisten COBOL-Compiler nur das Setzen des Wertes **TRUE** erlauben.

Dieses Verhalten wird oftmals ausgenutzt um Variablen mit bestimmten Werten zu belegen. Dieses Verhalten beschreibt Listing 4.28.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. VALUE-DEFAULT.
3
4      DATA DIVISION.
5      WORKING-STORAGE SECTION.
6          01 ERROR-MESSAGE PIC X(50) VALUE SPACE.
7              88 FIRST-ERROR VALUE "The first error occurred!".
8              88 SECOND-ERROR VALUE "The second error occurred!".
9              88 THIRD-ERROR VALUE "The third error occurred!".
10
11     PROCEDURE DIVISION.
12     MAIN.
13         SET SECOND-ERROR TO TRUE.
14         DISPLAY ERROR-MESSAGE.
15         STOP RUN.
16
17     END PROGRAM VALUE-DEFAULT.
```

Listing 4.28: Setzen von Werten mithilfe benannter Bedingungen

Abbildung in Java

Java besitzt kein Sprachkonstrukt, um die Funktionalität der Stufennummer 88 direkt nachzubilden. Eine Möglichkeit gleiches Verhalten darzustellen bietet allerdings die Implementierung spezieller Methoden. Dies soll Listing 4.29 veranschaulichen.

```

1      public class EightyEight{
2
3          public static void main(String[] args) {
4              AgeCheck ageCheck = new AgeCheck();
5              printAgeInformation(ageCheck);
6              ageCheck.setAge(18);
7              printAgeInformation(ageCheck);
8          }
9
10         public static void printAgeInformation(AgeCheck ageCheck)
11         {
```

```
12     System.out.println(  
13         String.format(  
14             "Age %s is %s!",  
15             ageCheck.getAge(),  
16             ageCheck.isAdult() ? "adult":"underage"  
17         )  
18     );  
19 }  
20  
21 static class AgeCheck  
22 {  
23     private int age;  
24  
25     public AgeCheck() {  
26         age = 13;  
27     }  
28  
29     public void setAge(int newAge) {  
30         age = newAge;  
31     }  
32  
33     public int getAge() {  
34         return age;  
35     }  
36  
37     public boolean isUnderage() {  
38         return (0 <= age && age <= 17);  
39     }  
40  
41     public boolean isAdult() {  
42         return (age >= 18);  
43     }  
44 }  
45 }
```

Listing 4.29: Bedingte Werte in Java

Die Methoden `isUnderage` und `isAdult` geben einen Wahrheitswert in Abhängigkeit des Variablenwertes zurück. Die Funktion `setAge` setzt wiederum das Alter.

Der Anwendungsfall, dass eine benannte Bedingung in COBOL verwendet wird um bestimmte Werte zu setzen lässt sich in Java am elegantesten über den Aufzählungstypen

enum realisieren. Wie der Typ `ErrorMessage` in Listing 4.30 zeigt, setzt jeder einzelne konstante Wert des Aufzählungstypen eine eigene Fehlernachricht, welche anschließend über die `getMessage`-Funktion verfügbar ist.

```

1  public class ValueSet {
2
3      enum ErrorMessage {
4          FIRST_ERROR("The first error occurred!"),
5          SECOND_ERROR("The second error occurred!"),
6          THIRD_ERROR("The third error occurred!");
7
8          private String message;
9
10         private ErrorMessage(String message){
11             this.message = message;
12         }
13
14         String getMessage(){
15             return this.message;
16         }
17     }
18
19     public static void main(String[] args) {
20         ErrorMessage errorMessage = ErrorMessage.SECOND_ERROR;
21         System.out.println(errorMessage.getMessage());
22     }
23 }

```

Listing 4.30: Setzen eines konstanten Wertes mit einem Enum in Java

4.9.2 Mehrfachverzweigungen

Ein wichtiges Konstrukt um den Programmfluß eines Programms zu steuern sind Mehrfachverzweigungen. Obwohl sowohl Java als auch COBOL Mehrfachverzweigungen bieten, sind diese doch leicht unterschiedlich zu verwenden. Im Folgenden sollen verschiedene Verwendungsmöglichkeiten der jeweiligen Konstrukte dargestellt werden.

In Java bildet das **switch-case**-Konstrukt eine Mehrfachverzweigung ab. Listing 4.31 zeigt dabei die wichtigsten Verwendungsmöglichkeiten.

```
1  public class MyCalendar {
2
3      public void printMonthDays(int month)
4      {
5          switch(month){
6              case 1: case 3:
7              case 5: case 7:
8              case 8: case 10:
9              case 12:
10                 System.out.println("This Month has 31 days.");
11                 break;
12
13             case 2:
14                 System.out.println("This Month has either 28 or 29 days.");
15                 break;
16
17             case 4: case 5:
18             case 9: case 11:
19                 System.out.println("This Month has 30 days.");
20                 break;
21
22             default:
23                 System.out.println("Month is not valid");
24                 break;
25             }
26         }
27     }
28 }
```

Listing 4.31: Mehrfachverzweigungen in Java

Zum einen ist zu beachten, dass Werte nur mit Literalen und Konstanten verglichen werden können. Ein Vergleich einer Variablen mit einer weiteren ist hierbei nicht zulässig, solange diese nicht als **final** deklariert ist. Auch der Vergleich auf bestimmte Wertebereiche oder nicht-primitive Datentypen ist nicht zulässig (Seit Java 7 sind vergleiche mit String-Literalen möglich).

Jedoch ist ein gewolltes »Durchfallen« möglich um bei verschiedenen Werten die gleichen Programmzweige zu durchlaufen. Dabei spielt das Schlüsselwort **break** eine entscheidende Rolle. Wird kein **break** am Ende einer **case**-Anweisung verwendet, so wird automatisch in den ausführbaren Block der darauffolgenden **case**-Anweisung gesprun-

gen. Dies zeigt sich in den Zeilen 6–9 und 17f. von Listing 4.31. Die Verwendung der **break**-Anweisung wird hingegen in den Zeilen 11, 15 und 20 genutzt um den **switch**-Block zu verlassen.

Das Fehlen eines **break** kann in der Praxis schnell zu unerwünschtem und unerklärlichem Verhalten führen. Deshalb folgt in der Regel jedem **case** ein **break**.

Das Pendant in COBOL stellt das Schlüsselwort **EVALUATE** dar. Wenngleich es den gleichen Sinn wie das **switch-case**-Konstrukt in Java erfüllen soll, ist es vielseitiger einsetzbar wie die folgenden Beispiele illustrieren sollen. Listing 4.32 und Listing 4.33 sind hierbei semantisch gleich, obwohl das **EVALUATE**-Konstrukt jeweils leicht anders verwendet wird.

Das folgende Listing 4.32 führt die Verwendungsmöglichkeit an, die dem **switch-case**-Konstrukt in Java am nächsten kommt. Nach dem **EVALUATE**-Schlüsselwort werden Variablennamen angegeben, deren Werte anschließend in einer **WHEN**-Bedingung betrachtet werden sollen.

Auch hier sieht man in den Zeilen 14, 17, 20, 23 ein gewolltes »Durchfallen« wie in Java. Einziger Unterschied hierbei ist, dass in COBOL jeder ausführbare Block nach einem **WHEN** eigenständig ist somit kein **break** notwendig ist. Ein »Durchfallen« ist also nur möglich, wenn der komplette Anweisungsblock leer ist.

Der **OTHER**-Zweig entspricht in COBOL dem aus Java bekannten **default**. Dieser Zweig wird ausgeführt wenn die Kriterien keines anderen zutreffen.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. SWITCH-CASE-EVALUATE.
3
4      DATA DIVISION.
5      WORKING-STORAGE SECTION.
6          01 AGE PIC 9(3).
7          01 SEX PIC X(1).
8
9      PROCEDURE DIVISION.
10     MAIN.
11         ACCEPT AGE.
12         ACCEPT SEX.
13         EVALUATE AGE ALSO SEX
14             WHEN 0 THRU 17 ALSO "M"
15             WHEN 0 THRU 17 ALSO "m"
16                 DISPLAY "Underage boy"
17             WHEN 0 THRU 17 ALSO "F"
18             WHEN 0 THRU 17 ALSO "f"
19                 DISPLAY "Underage girl"
20             WHEN 17 THRU 99 ALSO "M"
```



```

21         WHEN 17 THRU 99 ALSO "m"
22             DISPLAY "Adult man"
23         WHEN 17 THRU 99 ALSO "F"
24         WHEN 17 THRU 99 ALSO "f"
25             DISPLAY "Adult woman"
26         WHEN OTHER
27             DISPLAY "Unknown age or gender"
28     END-EVALUATE.
29     STOP RUN.
30
31 END PROGRAM SWITCH-CASE-EVALUATE.

```

Listing 4.32: Mehrfachverzweigungen in COBOL mit ALSO

Die erste Besonderheit, die Listing 4.32 illustrieren soll ist das Testen von jeweils zwei Bedingungen. Dies geschieht mithilfe des **ALSO**-Schlüsselworts. Während in Java lediglich die Evaluation einer einzelnen Bedingung möglich ist, erlaubt COBOL an dieser Stelle die Überprüfung beliebig vieler Kriterien.

Eine weitere Eigenheit zeigt sich in der Auswertung der Variable **AGE**. Hierbei werden im Beispiel Wertebereiche angegeben in denen die Variable liegen kann. Auch das ist in Java so nicht möglich, da wie bereits erwähnt nur mit Literale und Konstanten verglichen werden kann.

```

1     IDENTIFICATION DIVISION.
2     PROGRAM-ID. SWITCH-CASE-EVALUATE.
3
4     DATA DIVISION.
5     WORKING-STORAGE SECTION.
6         01 FIRST-NUM PIC 9(2).
7         01 SECOND-NUM PIC 9(2).
8
9     PROCEDURE DIVISION.
10    MAIN.
11        ACCEPT FIRST-NUM.
12        ACCEPT SECOND-NUM.
13        EVALUATE TRUE
14            WHEN FIRST-NUM EQUALS SECOND-NUM
15                DISPLAY "Both numbers are equal"
16            WHEN FIRST-NUM > SECOND-NUM*2 OR FIRST-NUM*2 < SECOND-NUM
17                DISPLAY "One number is more then twice the other"
18            WHEN FIRST-NUM < SECOND-NUM
19                DISPLAY "The first number is lower"
20            WHEN FIRST-NUM > SECOND-NUM
21                DISPLAY "The first number is greater"
22        END-EVALUATE.
23    STOP RUN.

```

Listing 4.33: Mehrfachverzweigungen in COBOL als EVALUATE TRUE

Listing 4.33 stellt weitere Unterschiede zu Mehrfachverzweigungen in Java dar. So ist diese **EVALUATE TRUE**-Variante (auch als **EVALUATE FALSE** möglich) mit geschachtelten **if**-Abfragen in Java zu vergleichen. Jede Bedingung hinter dem **WHEN**-Schlüsselwort entspricht hierbei einem vollständigen logischen Ausdruck. Daher ist es möglich wie in Zeile 16 gezeigt, Rechenoperationen durchzuführen oder logische Operatoren wie in diesem Fall das **OR** zu verwenden.

Abschließend kann noch erwähnt werden, dass sowohl Variablen-Vergleiche wie in Listing 4.32 als auch **TRUE**-Vergleiche wie in Listing 4.33 zusammen, mithilfe des **ALSO**-Schlüsselworts verwendet werden können.

4.9.3 Speicherausrichtung

Dieser Abschnitt soll einen kurzen Abriss über die COBOL Stufennummer *77* geben. Mit der Stufennummer *77* deklarierte Daten haben folgende beiden Eigenschaften:

- Die Variable kann nicht weiter untergruppiert werden.
- Die Variable wird an festen Grenzen des Speichers ausgerichtet.

Während die erste erwähnte Eigenschaft wenig Bewandnis in der Praxis hat, war es früher nötig Variablen für bestimmte Instruktionen an festen Speichergrenzen auszurichten. Üblicherweise mussten die Adressen dieser Grenzen je nach Compiler ganzzahlig durch 4 bzw. 8 teilbar sein. Dieses Verhalten ist heutzutage jedoch nicht mehr nötig. Dies und die Tatsache, dass durch dieses forcierte Speicherausrichtung Speicherbereiche zwischen Daten mit Stufennummer *77* ungenutz, aber jedoch reserviert bleiben, führen dazu, dass diese Stufennummer nicht mehr genutzt werden sollte. In Java gibt es kein vergleichbares Konzept, da die Speicherbelegung gänzlich abstrahiert ist und dem Entwickler nicht ermöglicht wird direkten Einfluss darauf zu nehmen.


4.9.4 Reorganisation von Daten

Wie bereits in Abschnitt 4.2 erwähnt beinhaltet COBOL drei Stufennummern denen eine besondere Rolle zuteilwird. Dieser Abschnitt soll daher kurz die COBOL Stufennummer 66 beschreiben.

In Listing 4.34 wird die Stufennummer 66 in Verbindung mit der **RENAMES**-Anweisung, was zwingend erforderlich ist, verwendet, um Teile der Personendaten neu zu gruppieren. Dies geschieht durch die Verwendung des **THRU**-Schlüsselworts. Ohne die Angabe dieses Bereichs können auch einzelne Variablen umbenannt werden. Wichtig ist hierbei zu erwähnen, dass lediglich eine neue Referenz auf den selben Speicherbereich erstellt, nicht jedoch neuer Speicher alloziert wird.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. COBOL-RENAMES-EXAMPLE.
3      DATA DIVISION.
4      FILE SECTION.
5      WORKING-STORAGE SECTION.
6      01 PERSON-DATA.
7          05 FIRST-NAME PIC X(10) VALUE "Max".
8          05 SURNAME PIC X(10) VALUE "Mustermann".
9          05 STREET PIC X(15) VALUE "Musterstraße".
10         05 HOUSENUMBER PIC X(5) VALUE "7a".
11         05 ZIP-CODE PIC X(6) VALUE "12345".
12         05 CITY PIC X(15) VALUE "Musterstadt".
13
14         66 PERSON-NAME RENAMES FIRST-NAME THRU SURNAME.
15         66 PERSON-ADDRESS RENAMES STREET THRU CITY.
16
17     PROCEDURE DIVISION.
18     MAIN-PROCEDURE.
19         DISPLAY PERSON-NAME.
20         DISPLAY PERSON-ADDRESS.
21         STOP RUN.
22
23     END PROGRAM COBOL-RENAMES-EXAMPLE.
```



```

1  Max      Mustermann
2  Musterstraße 7a  12345 Musterstadt
```

Listing 4.34: Stufennummer 66 und **RENAMES**-Befehl

Diese Stufennummer wird in der Praxis selten verwendet und auch ist in Java zu dieser Stufennummer kein exaktes Pendant zu finden.

Abbildung in Java

Die Gruppierung von Daten erfolgt in Java in eigenen Klassen, aus denen sich wiederum andere Objekte zusammensetzen können. Diese Aggregationsbeziehung ist im Diagramm in Abbildung 4.3 dargestellt.

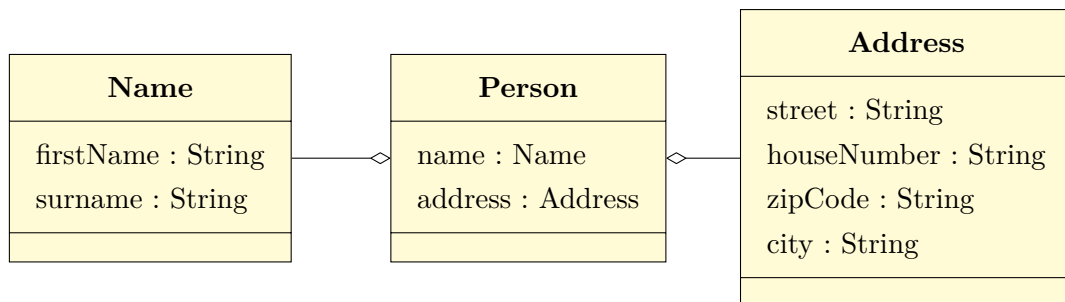


Abbildung 4.3: UML-Diagramm einer Aggregation

Die Verwendung der Stufennummer 66 als reines Umbenennen eines Datums entfällt in Java, da in diesem Fall neue Variablen mit anderen Namen deklariert werden können, welchen der ursprüngliche Wert der zugewiesen wird.

4.9.5 Implizierte Variablennamen

Dieser Abschnitt behandelt einen Fehler, der typischerweise zu Beginn in der Softwareentwicklung beobachtet werden kann. Listing 4.35 zeigt wie Anfänger häufig versuchen logische Ausdrücke zu konstruieren. Dies entspricht dem intuitiven Gedanken »Wenn Variable X größer 0 und kleiner 5 ist, dann ...« oder der mathematischen Definition » $0 < X < 5$ «.

```

1  public class IfVariableError {
2
3      public static void main(String[] args) {
4          long currentTimeMillis = System.currentTimeMillis();
5
6          if (currentTimeMillis > 0 && < Long.MAX_VALUE) {

```

```

7         System.out.println("We never get here!");
8     }
9
10    if (0 < currentTimeMillis < Long.MAX_VALUE) {
11        System.out.println("We never get here either!");
12    }
13 }
14
15 }

```

```

1  $ javac -Xmaxerrs 3 IfVariableError.java
2  IfVariableError.java:4: error: > expected
3      if (System.currentTimeMillis() > 0 && < Long.MAX_VALUE) {
4                                          ^
5  IfVariableError.java:4: error: ')' expected
6      if (System.currentTimeMillis() > 0 && < Long.MAX_VALUE) {
7                                          ^
8  IfVariableError.java:8: error: illegal start of type
9      if (0 < System.currentTimeMillis() < Long.MAX_VALUE) {
10     ^
11  3 errors

```

Listing 4.35: Keine implizierten Variablennamen in logischen Ausdrücken in Java

Versucht man Listing 4.35 zu kompilieren treten einige Fehler auf. In Java können nur vollständige Logische Ausdrücke mit logischen Operatoren verknüpft werden. Zum anderen kann innerhalb eines logischen Ausdrucks lediglich maximal einmal ein Vergleichsoperator verwendet werden.

```

1  public class IfVariableNoError {
2
3      public static void main(String[] args) {
4          long currentTimeMillis = System.currentTimeMillis();
5          if (0 < currentTimeMillis && currentTimeMillis < Long.MAX_VALUE) {
6              System.out.println("We get here everytime!");
7          }
8      }
9  }

```

```

1 $ javac IfVariableNoError.java
2 $ java IfVariableNoError
3 We get here everytime!

```

Listing 4.36: Verwendung von Variablennamen in logischen Ausdrücken in Java

Listing 4.36 demonstriert eine funktionsfähige Implementierung des vorhergehenden Beispiels. Dieser Code kann fehlerfrei kompiliert und ausgeführt werden wie die Ausgabe zeigt.

Implizierte Variablennamen in COBOL

In COBOL hingegen ist das Schreiben von logischen Ausdrücken mit implizierten Variablennamen möglich.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. IMPLICIT-VARIABLE-NAMES.
3
4      DATA DIVISION.
5      WORKING-STORAGE SECTION.
6          01 INPUT-NUMBER PIC 9(2).
7
8      PROCEDURE DIVISION.
9      MAIN-PROCEDURE.
10         ACCEPT INPUT-NUMBER.
11         IF INPUT-NUMBER >= 10 AND <= 20 THEN
12             DISPLAY "Your number is >= 10 and <= 20."
13         ELSE
14             DISPLAY "Your number is < 10 or > 20."
15         END-IF.
16         STOP RUN.
17
18     END PROGRAM IMPLICIT-VARIABLE-NAMES.

```

Listing 4.37: Implizierte Variablennamen in COBOL

Listing 4.37 stellt in Zeile 11 die Verwendung von implizierten Variablennamen in COBOL dar, die im Gegensatz zu Java möglich ist.

4.9.6 Modifier

In Java ist es, anders als in COBOL, möglich die Sichtbarkeit von Variablen, Funktionen und Klassen nach außen zu steuern. Dazu dienen sogenannte *Modifier*, die an dieser Stelle kurz erläutert werden.

Auf Funktionen oder Variablen, die mit dem Schlüsselwort **public** gekennzeichnet sind, kann von jeder Stelle des Programms aus zugegriffen werden.

protected beschränkt den Zugriff auf die enthaltende Klasse (mitsamt geschachtelter Klassen) und alle Unterklassen. Erbt eine Klasse eine solche Methode oder Variable kann sie diese also nutzen. Andere Klassen haben keinen Zugriff darauf.

Um maximal-restriktiven Zugriff auf einen Teil einer Klasse sicherzustellen bietet sich das Schlüsselwort **private** an. Dieses erlaubt nur Zugriffe aus der enthaltenden Klasse und aus geschachtelten Klassen.

Wird kein Schlüsselwort explizit genutzt so ist die Sichtbarkeit auf das aktuelle Package beschränkt (engl. *package-private*). Das bedeutet, dass alle Klassen des selben Package Zugriff erhalten, wohingegen alle Klassen anderer Packages keinen erhalten.

5 Typische Pattern in COBOL und Java

5.1 Komplexe Datenstrukturen

5.1.1 Listen

Während Abschnitt 4.3 Felder behandelt, welche wie angesprochen eine feste Größe haben, bietet das Konzept von Listen deutliche Vorteile, wenn die Anzahl der Elemente variabel sein soll und zum Zeitpunkt des Erstellens nicht bekannt ist.

Listen in Java

In Java kann das `List`-Interface implementiert werden bzw. ein Objekt dieser Implementierung instanziiert werden, um eine Liste variabler Größe zu erhalten. Die wohl gebräuchlichste Implementierung dieses Interfaces stellt die Klasse `ArrayList` dar. Intern hält diese – wie der Name schon vermuten lässt – ein Array, welches bei Bedarf in ein neues, größeres, Array kopiert wird. Die einfache Handhabung dieser Klasse wird in Listing 5.1 dargestellt.

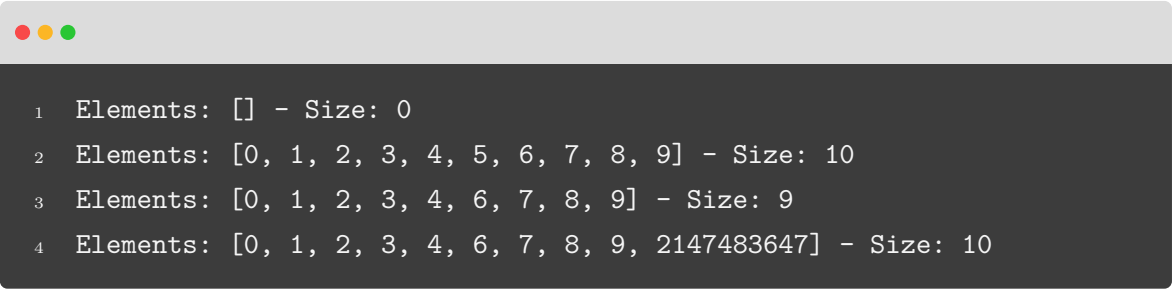
```
1  import java.util.ArrayList;
2  import java.util.Arrays;
3  import java.util.List;
4  import java.util.stream.IntStream;
5
6  public class ArrayListExample{
7
8      public static void main(String[] args) {
9          List<Integer> integerList = new ArrayList<>();
10         printListInformation(integerList);
11
12         IntStream.range(0, 10)
13             .forEach(value -> integerList.add(value)) ;
```



```

14     printListInformation(integerList);
15
16     integerList.remove(5);
17     printListInformation(integerList);
18
19     integerList.add(Integer.MAX_VALUE);
20     printListInformation(integerList);
21 }
22
23 private static void printListInformation(List<Integer> list) {
24     System.out.println(
25         "Elements: " + Arrays.toString(list.toArray()) +
26         " - Size: " + list.size()
27     );
28
29 }
30 }

```



```

1 Elements: [] - Size: 0
2 Elements: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] - Size: 10
3 Elements: [0, 1, 2, 3, 4, 6, 7, 8, 9] - Size: 9
4 Elements: [0, 1, 2, 3, 4, 6, 7, 8, 9, 2147483647] - Size: 10

```

Listing 5.1: ArrayList Beispiel in Java

Listen in COBOL

Eine exakte Abbildung von Listen ist in COBOL nicht möglich, da hier bereits zum Zeitpunkt des Kompilierens feststehen muss, wie groß ein Feld ist.

```


1 IDENTIFICATION DIVISION.
2 PROGRAM-ID. LIST-EXAMPLE.
3
4 DATA DIVISION.
5 WORKING-STORAGE SECTION.
6     01 LIST PIC 9(3) OCCURS 99 TIMES INDEXED BY L-IDX.
7     01 D-IDX PIC 9(2).
8     01 D-IDX-COUNT PIC 9(2).

```

```

9          01 D-IDX-COUNT-TMP PIC 9(2).
10         01 P-IDX PIC 9(2).
11         01 I-VAL PIC 9(3).
12
13     PROCEDURE DIVISION.
14     MAIN-PROCEDURE.
15         PERFORM PRINT-LIST.
16         MOVE 2 TO I-VAL. PERFORM INSERT-VALUE. PERFORM PRINT-LIST.
17         MOVE 4 TO I-VAL. PERFORM INSERT-VALUE. PERFORM PRINT-LIST.
18         MOVE 1 TO D-IDX. PERFORM DELETE-VALUE. PERFORM PRINT-LIST.
19         STOP RUN.
20
21     INSERT-VALUE SECTION.
22         MOVE I-VAL TO LIST(L-IDX).
23         IF L-IDX < 99 THEN
24             COMPUTE L-IDX = L-IDX + 1
25         END-IF.
26
27     DELETE-VALUE SECTION.
28         IF D-IDX <= 99 THEN
29             COMPUTE L-IDX = L-IDX - 1
30             PERFORM VARYING D-IDX-COUNT
31                 FROM D-IDX BY 1
32                 UNTIL D-IDX-COUNT = L-IDX
33                 COMPUTE D-IDX-COUNT-TMP = D-IDX-COUNT + 1
34                 MOVE LIST(D-IDX-COUNT-TMP) TO LIST(D-IDX-COUNT)
35             END-PERFORM
36         END-IF.
37
38     PRINT-LIST SECTION.
39         PERFORM VARYING P-IDX FROM 1 BY 1 UNTIL P-IDX = L-IDX
40             DISPLAY LIST(P-IDX)", " WITH NO ADVANCING
41         END-PERFORM.
42         COMPUTE P-IDX = L-IDX - 1.
43         DISPLAY " SIZE: " P-IDX.
44
45     END PROGRAM LIST-EXAMPLE.

```



```

1  SIZE: 00
2  002, SIZE: 01
3  002,004, SIZE: 02
4  004, SIZE: 01

```

Listing 5.2: Einfache Listen Implementierung in COBOL

Listing 5.2 zeigt jedoch beispielhaft eine einfache und unvollständige Implementierung einer Liste in COBOL. Hierbei sind lediglich Einfüge- und Löschoperationen realisiert.

Zu beachten ist, dass aus bereits genannten Gründen auch diese Liste eine maximale Größe hat, die unter Umständen nicht ausreichend ist. Weitere Funktionalitäten der Liste müssten analog implementiert werden.

5.1.2 Sets

Neben den in Unterabschnitt 5.1.1 beschriebenen Listen bieten Sets in der Programmierung eine weitere häufig genutzte Datenstruktur. Die zwei wesentlichen Unterschiede im Gegensatz zu Listen sind, zum einen eine fehlende Ordnung der Elemente und die Eigenschaft, dass ein und das selbe Element nur genau einmal innerhalb eines Sets vorkommen darf. Das Set entspricht somit weitestgehend der mathematischen Definition einer Menge.

Sets in Java

Wie für Listen bietet Java auch für Sets das `Set`-Interface. Die wohl am häufigsten genutzte Implementierung dieses Interfaces stellt die `HashSet`-Klasse dar, welche die `hashCode`-Methode eines Objektes nutzt um es pseudo-eindeutig identifizierbar zu machen.

```
1  import java.util.HashSet;
2  import java.util.Set;
3  import java.util.stream.IntStream;
4
5  public class HashSetExample {
6
7      public static void main(String[] args) {
8          Set<Integer> integerSet = new HashSet<>();
9
10         IntStream.range(0, 10).forEach(number -> integerSet.add(number));
11         integerSet.forEach(number -> System.out.print(number + " "));
12
13         System.out.println();
14
15         IntStream.range(5, 15).forEach(number -> integerSet.add(number));
16         integerSet.forEach(number -> System.out.print(number + " "));
17     }
18 }
```

```

1  0 1 2 3 4 5 6 7 8 9
2  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

```

Listing 5.3: HashSet Beispiel in Java

In Listing 5.3 soll gezeigt werden, dass das Einfügen von Elementen, welche bereits im Set enthalten sind keine Auswirkungen hat. Die Eigenschaft, dass ein Set ungeordnet ist, lässt sich leider nicht zeigen, da Java die Werte bei der gezeigten Ausgabe ordnet. Dieses Verhalten tritt auch auf, wenn die Werte in umgekehrter Reihenfolge dem Set hinzugefügt werden, was jedoch in keinem Fall bedeutet, dass es sich beim Set um eine stets sortierte Liste handelt, auch wenn es den Eindruck vermittelt!

Sets in COBOL

Auch an dieser Stelle kann analog zu Listen gesagt werden, dass eine Implementierung von Sets in COBOL nicht ohne weiteres möglich ist. Die Einschränkung der Größe der Datenstruktur bestünde auch hier.

```

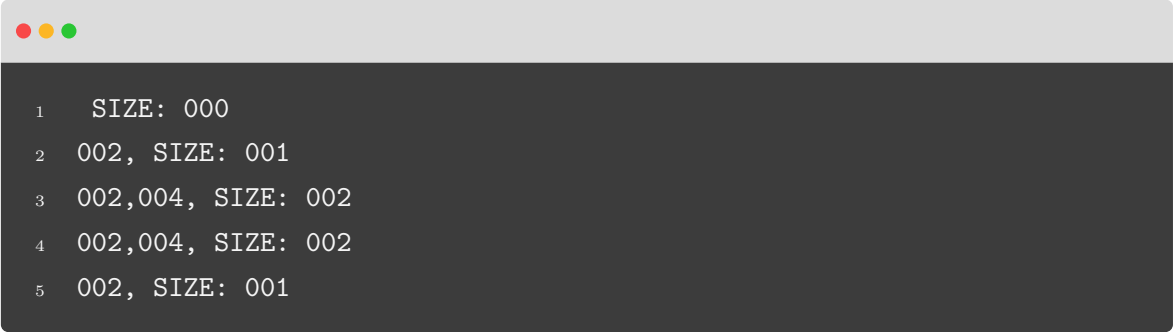
1  IDENTIFICATION DIVISION.
2  PROGRAM-ID. SET-EXAMPLE.
3
4  DATA DIVISION.
5  WORKING-STORAGE SECTION.
6      01 SET-STORAGE PIC 9(3) OCCURS 100 TIMES INDEXED BY S-IDX.
7      01 SET-NIL-VALUE PIC 9(3) VALUE 0.
8      01 SET-SIZE PIC 9(3) VALUE 000.
9      01 I-VAL PIC 9(3).
10     01 D-VAL PIC 9(3).
11
12  PROCEDURE DIVISION.
13  MAIN-PROCEDURE.
14     PERFORM INIT-SET.
15     PERFORM PRINT-SET.
16     MOVE 2 TO I-VAL. PERFORM INSERT-VALUE. PERFORM PRINT-SET.
17     MOVE 4 TO I-VAL. PERFORM INSERT-VALUE. PERFORM PRINT-SET.
18     MOVE 2 TO I-VAL. PERFORM INSERT-VALUE. PERFORM PRINT-SET.
19     MOVE 4 TO D-VAL. PERFORM DELETE-VALUE. PERFORM PRINT-SET.
20     STOP RUN.
21
22  INIT-SET SECTION.
23     PERFORM VARYING S-IDX FROM 1 BY 1 UNTIL S-IDX = 100
24         MOVE SET-NIL-VALUE TO SET-STORAGE(S-IDX)

```

```

25         END-PERFORM.
26
27     INSERT-VALUE SECTION.
28     SEARCH-EQUAL-VALUE.
29         PERFORM VARYING S-IDX FROM 1 BY 1
30         UNTIL S-IDX = 100 OR I-VAL = SET-NIL-VALUE
31             IF SET-STORAGE(S-IDX) = I-VAL THEN
32                 SET I-VAL TO SET-NIL-VALUE
33             END-IF
34         END-PERFORM.
35
36     INSERT-IF-NOT-ALREADY-PRESENT.
37         PERFORM VARYING S-IDX FROM 1 BY 1
38         UNTIL S-IDX = 100 OR I-VAL = SET-NIL-VALUE
39             IF SET-STORAGE(S-IDX) = SET-NIL-VALUE THEN
40                 MOVE I-VAL TO SET-STORAGE(S-IDX)
41                 SET I-VAL TO SET-NIL-VALUE
42                 COMPUTE SET-SIZE = SET-SIZE + 1
43             END-IF
44         END-PERFORM.
45
46     INSERT-VALUE-EXIT.
47         EXIT.
48
49     DELETE-VALUE SECTION.
50         PERFORM VARYING S-IDX FROM 1 BY 1
51         UNTIL S-IDX = 100 OR D-VAL = SET-NIL-VALUE
52             IF SET-STORAGE(S-IDX) = D-VAL THEN
53                 SET SET-STORAGE(S-IDX) TO SET-NIL-VALUE
54                 SET D-VAL TO SET-NIL-VALUE
55                 COMPUTE SET-SIZE = SET-SIZE - 1
56             END-IF
57         END-PERFORM.
58
59     PRINT-SET SECTION.
60         PERFORM VARYING S-IDX FROM 1 BY 1 UNTIL S-IDX = 100
61             IF NOT SET-STORAGE(S-IDX) = SET-NIL-VALUE THEN
62                 DISPLAY SET-STORAGE(S-IDX) ", " WITH NO ADVANCING
63             END-IF
64         END-PERFORM.
65         DISPLAY "SIZE: " SET-SIZE.
66
67     END PROGRAM SET-EXAMPLE.

```



```

1  SIZE: 000
2  002, SIZE: 001
3  002,004, SIZE: 002
4  002,004, SIZE: 002
5  002, SIZE: 001

```

Listing 5.4: Einfache Set Implementierung in COBOL

Listing 5.4 greift jedoch beispielhaft die Kernaspekte von Sets auf und zeigt eine mögliche Implementierung in COBOL. Wie auch in Listing 5.4 sind nur Einfüge- und Löschoptionen realisiert.

5.1.3 Maps

Maps erlauben den Zugriff auf bestimmte Elemente in konstanter Zeit, d.h. $O(n)$. Erreicht wird dies – vereinfacht dargestellt – dadurch, dass der Speicherbereich in dem ein Element der Datenstruktur gespeichert werden soll, mithilfe der Daten des Objekts berechnet wird. Das ganze nennt sich *Hash-Funktion*. Dies eignet sich vor allem wenn viele Elemente in der Datenstruktur gespeichert werden sollen oder konstante Zugriffszeit auf jedes Element notwendig ist.

In Java steht dafür das Interface `Map` und Implementierungen wie `HashMap` bereit. Damit werden generische Objekte anhand ihrer `hashCode`-Funktion in einer Map verwaltet.

COBOL bietet auch hier keine Konstrukte um eine Map abzubilden. Wie die Experten Bonev und Streit jedoch angaben wurden in der Praxis bereits Möglichkeiten entwickelt ein solches Verhalten in COBOL abzubilden. Eine einfache wäre eine eigene Hash-Funktion für den genauen Anwendungsfall zu implementieren, die Eingabewerte eindeutig auf Indizes eines Arrays fester Größe abbildet. Eine weitere Möglichkeit.

5.1.4 Verbunddatenstrukturen

Structs und *Unions* sind weitere in der Programmierung – vor allem mit älteren Sprachen – verbreitete Datenstrukturen. Beide Typen bilden sogenannte Verbunddatenstrukturen,

d.h. sie bieten eine Art Kapselung von mehreren Variablen, welche unterschiedlichen Typs sein können. Structs bezeichnen dabei einen Speicherbereich der von aus mehreren Variablen besteht, die so gruppiert gespeichert werden. Unions hingegen bezeichnen einen Speicherbereich, der auf verschiedene Weisen – in unterschiedlichen Datentypen – interpretiert werden kann. Sie gruppieren also mehrere Variablen von denen stets nur eine Gültigkeit aufweisen kann.

Structs und Unions in COBOL

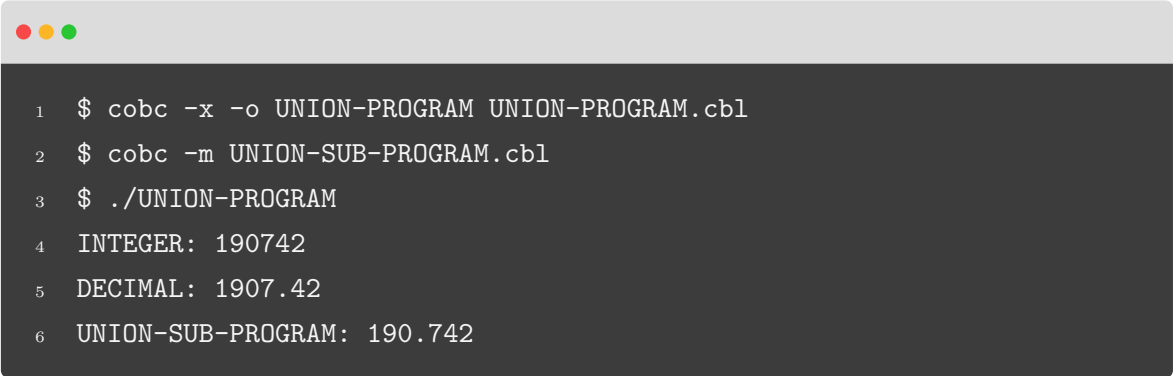
Das fundamentale Konzept von Variablendeklarationen in COBOL ist – wie bereits in Abschnitt 4.2 angesprochen – die Nutzung von Stufennummern und eine Untergliederung mithilfe dieser Stufennummern. Das sorgt dafür, dass jede untergliederte Variable in COBOL eine Verbunddatenstruktur darstellt.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID.  UNION-PROGRAM.
3      DATA DIVISION.
4      WORKING-STORAGE SECTION.
5          01 NUMBER-GROUP.
6              05 FIRST-NUMBER PIC 9(4) VALUE 1907.
7              05 SECOND-NUMBER PIC 9(2) VALUE 42.
8          01 NEW-NUMBER REDEFINES NUMBER-GROUP PIC 9(4)V9(2).
9
10     PROCEDURE DIVISION.
11     MAIN-PROCEDURE.
12         DISPLAY "INTEGER: " NUMBER-GROUP.
13         DISPLAY "DECIMAL: " NEW-NUMBER.
14         CALL "UNION-SUB-PROGRAM" USING NUMBER-GROUP.
15         STOP RUN.
16
17     END PROGRAM UNION-PROGRAM.
```

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID.  UNION-SUB-PROGRAM.
3      DATA DIVISION.
4      LINKAGE SECTION.
5          01 PASSED-VALUE PIC 9(3)V9(3).
6
7      PROCEDURE DIVISION USING PASSED-VALUE.
8      MAIN-PROCEDURE.
9          DISPLAY "UNION-SUB-PROGRAM: " PASSED-VALUE.
10         GOBACK.
11
12     END PROGRAM UNION-SUB-PROGRAM.
```



```
1 $ cobc -x -o UNION-PROGRAM UNION-PROGRAM.cbl
2 $ cobc -m UNION-SUB-PROGRAM.cbl
3 $ ./UNION-PROGRAM
4 INTEGER: 190742
5 DECIMAL: 1907.42
6 UNION-SUB-PROGRAM: 190.742
```

Listing 5.5: Structs und Unions in COBOL

Listing 5.5 zeigt sowohl die Verwendung als Struktur-Typ, als auch den Zugriff als Union-Typ in einem Unterprogramm und mithilfe der in Abschnitt 4.2 beschriebenen **REDEFINES** Anweisung. Wie bereits vormals in dieser Arbeit gezeigt, wird – in diesem Falle zur Ausgabe – auf eine gesamte Datenstruktur zugegriffen, die mehrere unterschiedliche Variablen enthält. Diese Datenstruktur stellt also ein *Struct* dar. Im gezeigten Unterprogramm hingegen wird der selbe Speicherbereich anders interpretiert

Dies wird laut Herrn Streit in der Praxis auch oft ausgenutzt um Erweiterungen von Programmen zu realisieren. Da Unterprogramme lediglich einen Zeiger auf Datenstrukturen übergeben bekommen können diese Strukturen erweitert werden ohne, dass das Unterprogramm geändert werden muss, solange neue Datenfelder an das Ende angehängt werden. Weitere Unterprogramme, welche an neue Datenfelder angepasst wurden können wiederum diese Daten nutzen, die hinten angehängt wurden. Zu beachten ist, dass das Einfügen von Feldern zwischen anderen unweigerlich zur Anpassung aller Komponenten sorgt.

Structs und Unions in Java

Java hingegen bietet keine direkten Konzepte um Verbunddatenstrukturen darzustellen. Allerdings können *Structs* sehr intuitiv als sogenannte *Java Beans*, eine Klasse die verschiedenen **public** Variablen kapselt, implementiert werden. Dies zeigt Listing 5.6. Je nach Anwendungsfall empfiehlt es sich jedoch auf Klassen mit privaten Variablen und getter- und setter-Methoden zurückzugreifen. *Unions* lassen sich in Java am besten durch, die in ?? beschriebenen, *Generics* darstellen.


```
1 public class Bean{  
2     public int age;  
3     public String firstName;  
4     public String surname;  
5 }
```

Listing 5.6: Java Bean



Die vorgestellten Datenstrukturen bieten eine Möglichkeit, je nach Anwendungsfall, Daten effizient und dynamisch zu speichern und bereitzustellen. Während Java Listen, Sets und Maps bereits im Sprachstandard unterstützt, müssen diese von Entwicklern in COBOL selbst implementiert werden, was selbstverständlich Wissen über die Algorithmik voraussetzt. Herr Streit betonte, dass etwaige Datenstrukturen manchmal in bestehenden COBOL-Programmen zu finden sind, jedoch – durch das Fehlen des entsprechenden Vokabulars in der Entstehungszeit – oft andere Namen tragen und daher als solches nicht zu erkennen sind. COBOL bietet Verbunddatenstrukturen die vielfältig genutzt, aber auch ausgenutzt werden können, als fundamentalen Teil des Variablenkonzepts. Bei der Nutzung ist darauf zu achten, dass

5.2 Entwurfsmuster

Vor allem in objektorientierter Programmierung sind Entwurfsmuster – engl. design pattern – von hoher Bedeutung. Dabei handelt es sich um Muster die sich häufig in der Planung und im Implementieren von Software bewährt haben und so als wiederverwendbare Vorgehensweise bei bestimmten Problemstellungen angewendet werden kann. Dieses Kapitel erläutert die gebräuchlichsten Entwurfsmuster in Java. Für einen umfassenden Überblick sei an dieser Stelle auf *Design Patterns* [11] von Gamma u. a. verwiesen, welches wohl als Grundlagenwerk der Informatik auf diesem Gebiet bezeichnet werden kann.

5.2.1 Callback-Muster

In der Programmierung ist es häufig nötig, dass bestimmte Ereignisse andere Aktionen auslösen. Dies ist insbesondere in Programmen sinnvoll, die parallele Verarbeitungsschritte beinhalten. Um auf diese Ereignisse zu reagieren gibt es im Allgemeinen zwei Möglichkeiten:

- **(Busy-)Polling**

Unter Polling versteht man das zyklische Abfragen eines Wertes oder eines Zustandes, durch einen Teil, der von diesem Wert abhängt.

- **Callbacks**

Callbacks sind Funktionen, die in einer bestimmten Art und Weise anderen Programmabschnitten zur Verfügung gestellt werden und bei Bedarf aufgerufen werden können, um z.B. über Zustandsänderungen zu benachrichtigen.

Im Gegensatz zum Polling, bei dem permanent Rechenzeit dafür aufgewendet werden muss aktiv eine Zustandsänderung zu überwachen, wird in der Praxis häufig das Entwurfsmuster der Callbacks verwendet. Damit wird erreicht, dass ein Teilprogramm nicht wie beschrieben aktiv Änderungen beobachten muss, sondern sich von einem anderen Programmabschnitt über Zustände benachrichtigen lassen kann.

Obwohl das klassische Callback-Muster aufgrund von fehlenden funktionalen Elementen in Java nicht direkt implementierbar ist, finden sich oft sehr ähnliche Abbildungen davon. Die standardisierten Schnittstellen **Observer** und **Observable** bieten dabei eine generische Möglichkeit zur Änderungsbenachrichtigung. Häufiger lassen sich jedoch sogenannte **Listener** beobachten, welche das Callback-Muster abbilden sollen. Hierbei werden eigene Interfaces definiert, was zwar die Generizität verringert, jedoch zu breiterem Funktionsumfang und leichter verständlichem Code führt.

Das Listing 5.7 zeigt die Handhabung des **Observer**-Pattern [11] in Java mithilfe der **Observer** und **Observable**-Interfaces. Wie gezeigt, kann die Implementierung des **Observer**-Interfaces auch in einer anonymen Klasse geschehen. Diese sprachlichen Konstrukte wurden bereits in Abschnitt 4.1 beschrieben.

```
1  import java.time.Instant;  
2  import java.util.Observable;  
3  import java.util.Observer;  
4  import java.util.concurrent.Executors;
```

```

5  import java.util.concurrent.ScheduledExecutorService;
6  import java.util.concurrent.TimeUnit;
7
8  public class ObserverPattern {
9
10     public static void main(String[] args) {
11         TimeObservable timeObservable = new TimeObservable();
12         Observer plainObserver = new Observer() {
13             public void update(Observable o, Object arg) {
14                 System.out.println("PlainTime: " + arg);
15             }
16         };
17         FormattedObserver formattedObserver = new FormattedObserver();
18         timeObservable.addObserver(plainObserver);
19         timeObservable.addObserver(formattedObserver);
20         timeObservable.startObservable();
21     }
22
23     static class FormattedObserver implements Observer {
24         public void update(Observable o, Object arg) {
25             System.out.println(
26                 "FormattedTime: " +
27                 Instant.ofEpochMilli((long) arg).toString());
28         }
29     }
30
31     static class TimeObservable extends Observable {
32
33         private ScheduledExecutorService executor =
34             ↪ Executors.newScheduledThreadPool(3);
35
36         public void startObservable() {
37             for(int delay : new int[]{1,3,5}) {
38                 executor.schedule(() -> {
39                     setChanged();
40                     notifyObservers(System.currentTimeMillis());
41                 }, delay, TimeUnit.SECONDS);
42             }
43         }
44     }
45 }

```

Listing 5.7: Observer und Observable in Java

Die sehr viel gebräuchlichere Variante ist es jedoch `Listener` zu verwenden. Dabei handelt es sich streng genommen um nichts anderes, als eine eigene Definition des `Observer`-Interfaces. Jedoch wird durch die klare Definition der Funktionalität deutlich spezifischerer Code geschrieben, der vor allem in puncto Typsicherheit und Lesbarkeit einige Vorteile gegenüber des Java-eigenen Interfaces bietet.

```

1  import java.time.Instant;
2  import java.util.ArrayList;
3  import java.util.List;
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.ScheduledExecutorService;
6  import java.util.concurrent.TimeUnit;
7
8  public class ListenerPattern {
9
10     public static void main(String[] args) {
11         TimeProducer timeProducer = new TimeProducer();
12         TimeListener plainListener = new TimeListener() {
13             @Override
14             public void printTime(long timeMillis) {
15                 System.out.println("PlainTime: " + timeMillis);
16             }
17         };
18         FormattedListener formattedListener = new FormattedListener();
19         timeProducer.registerListener(plainListener);
20         timeProducer.registerListener(formattedListener);
21         timeProducer.start();
22     }
23
24     interface TimeListener {
25         void printTime(long timeMillis);
26     }
27
28     static class FormattedListener implements TimeListener {
29         @Override
30         public void printTime(long timeMillis) {
31             System.out.println("FormattedTime: " +
32                 Instant.ofEpochMilli(timeMillis).toString());
33         }
34     }
35
36     static class TimeProducer {
37
38         private ScheduledExecutorService executor =
39             ↪ Executors.newScheduledThreadPool(3);
40         private List<TimeListener> timeListeners = new ArrayList<>();

```

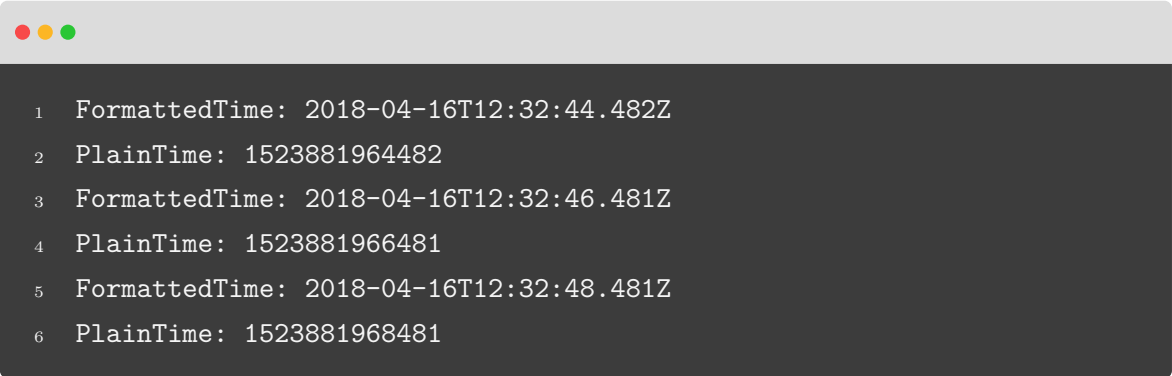
```

40
41     void registerListener(TimeListener timeListener) {
42         timeListeners.add(timeListener);
43     }
44
45     void notifyListeners(long value) {
46         timeListeners.forEach(listener -> listener.printTime(value));
47     }
48
49     void start() {
50         for (int delay : new int[] { 1, 3, 5 }) {
51             executor.schedule(
52                 () -> notifyListeners(System.currentTimeMillis()),
53                 delay, TimeUnit.SECONDS);
54         }
55     }
56 }
57
58 }

```

Listing 5.8: Listener in Java

Die Aufgabe beider Beispiele ist die gleiche. Jedoch verdeutlicht Listing 5.8 die Typsicherheit eigener Interfaces und kennzeichnet wie diese erreicht wird. Folgendes wäre eine beispielhafte Ausgabe beider Programme:



```

1  FormattedTime: 2018-04-16T12:32:44.482Z
2  PlainTime: 1523881964482
3  FormattedTime: 2018-04-16T12:32:46.481Z
4  PlainTime: 1523881966481
5  FormattedTime: 2018-04-16T12:32:48.481Z
6  PlainTime: 1523881968481

```

In COBOL hingegen ist ein solches Muster nicht möglich. Allerdings aufgrund der fehlenden Nebenläufigkeit und Objektorientierung auch nicht nötig.

5.2.2 Singleton-Muster

»Alles ist ein Objekt.« [24] Diese Aussage findet sich in unzähliger Hochschullektüre und Fachbüchern. Genauer wäre – mit Blick auf statische Elemente – zwar die Aussage, dass alles Teil einer Klasse sei, jedoch ist für die Idee dahinter beides richtig. Gemeint ist damit, dass es keine Funktionalität oder Eigenschaft gibt, die nicht Teil einer Klasse bzw. eines Objektes ist.

Manchmal jedoch wird eine Klasse zwar durch ein Objekt repräsentiert – ist also nicht statisch – allerdings existiert nur genau eine Instanz dieses Objekts. Im realen Leben könnte man z.B. den Papst als genau einmalig vorkommende Instanz ansehen. Diese spezielle Person gibt es stets nur einmal und so muss bei der Modellierung beachtet werden, dass es zu jedem Zeitpunkt nur eine Instanz der Klasse gibt.

```
1  public class Pope {
2
3      private static Pope instance = null;
4
5      private Pope() {}
6
7      public static Pope getInstance() {
8          if (instance == null) {
9              instance = new Pope();
10         }
11         return instance;
12     }
13
14     public static void die() {
15         instance = null;
16     }
17 }
```

Listing 5.9: Singleton in Java

Listing 5.9 modelliert das Beispiel des Papstes in Java. Um die angesprochenen Eigenschaften einer Singleton-Klasse zu realisieren werden verschiedene Mechanismen verwendet. Zum einen enthält die Klasse einen privaten Konstruktor um eine Instanziierung mittels `new`-Operators zu verhindern bzw. nur innerhalb der eigenen Klasse zuzulassen. Außerdem speichert und liefert die Klasse mit der statischen `getInstance()`-Methode

die aktuell gültige Instanz und legt ggf. eine neue an. Die `die()`-Funktion wurde beispielhaft für das Ableben eines Papstes implementiert, sodass in der `getInstance()`-Methode fortan eine neue Instanz erzeugt würde.

In der Praxis stellen zum Beispiel eine Datenbank-, Konfigurations- oder Hardware-schnittstellen oftmals ein Singleton-Objekt dar. Im Gegensatz zu statischen Klassen und Funktionen bieten Singletons einige Vorteile:

- Eine Singleton-Klasse kann Interfaces implementieren und von anderen Klassen erben.
- Singleton-Klassen können instanziiert werden, sobald sie gebraucht werden. Statische Klassen werden beim Starten des Programms initialisiert.
- Klassen können von Singleton-Klassen erben und erhalten damit die Member-Variablen und -Funktionen.

Allerdings gibt es auch Anwendungsfälle, in denen eine Klasse mit statischen Variablen und Funktionen genutzt werden sollte. Als Faustregel dafür gilt zum einen, dass die Klasse keinen Zustand repräsentiert und zum anderen lediglich eine Sammlung von Variablen und Funktionen gleicher Domäne – z.B. mathematische Operationen – bereitstellt.

5.2.3 Dependency Injection

In einer objektorientierten Programmierung werden Funktionalitäten in Klassen gekapselt und so die Wiederverwendbarkeit erhöht. Dies wurde bereits in Abschnitt 3.3 dargestellt. Um damit komplexere Probleme zu lösen und Abhängigkeiten herzustellen werden Klassen oftmals komponiert bzw. aggregiert. Dabei gibt es zwei Möglichkeiten wie diese Aggregationen aussehen können:

- Erzeugung aller nötigen Objekt-Instanzen innerhalb der Klasse die diese benötigt.
- Erzeugung der nötigen Objekt-Instanzen an zentraler Stelle und Injizieren in Objekte, welche diese benötigen (Dependency Injection).

Die erste der angesprochenen Vorgehensweise verletzt allerdings – je nach Ausnutzen und Implementierung – mehr oder weniger die Eigenschaft, dass eine Klasse für genau eine Aufgabe zuständig ist. Außerdem werden so implizit Abhängigkeiten zwischen Komponenten hergestellt, die schwer zu durchschauen sind. Dependency Injection sorgt dafür, dass Abhängigkeiten explizit hergestellt werden. Martin beschreibt in seinem Buch *Clean Code* [16] dieses Muster auch als »Inversion of Control«, da dabei die Kontrolle über die Instanziierung verlagert – quasi invertiert – wird. So entsteht eine losere Kopplung durch die Abhängigkeiten auch zur Laufzeit geändert und gesteuert werden können. Ein weiterer Vorteil zeigt sich beim Testen von Systemen. So lassen sich für Komponenten, welche Abhängigkeiten von Außen bekommen leichter Unit-Tests schreiben, da darunterliegende Strukturen extern gesteuert werden können.

```
1  public class DependencyInjection {
2
3      public static void main(String... args) {
4          Injectable injectable = () -> 42;
5          InjectionTarget injectionTarget = new InjectionTarget(injectable);
6          injectionTarget.printNumber();
7      }
8
9      interface Injectable {
10         int getNumber();
11     }
12
13     static class InjectionTarget {
14         private final Injectable injectable;
15
16         public InjectionTarget(Injectable injectable) {
17             this.injectable = injectable;
18         }
19
20         public void printNumber() {
21             System.out.println(injectable.getNumber());
22         }
23     }
24 }
```

Listing 5.10: Dependency Injection in Java

Listing 5.10 beinhaltet eine einfache Dependency Injection. Der Code macht gleichzeitig Gebrauch von Interfaces was bereits die einen Teil der Flexibilität der Dependency

Injection verdeutlicht. Neben der gezeigten Möglichkeit der Übergabe an den Konstruktor einer Klasse, erwähnt Martin [16] auch die Varianten diese Abhängigkeiten über Setter-Methoden oder Interfaces herzustellen.

Um Dependency Injection zu erreichen gibt es auch einige Frameworks mit deren Hilfe Objekte z.B. mittels Annotationen sehr komfortabel injiziert werden können. Allerdings ist in der Praxis davon abzuraten diese Frameworks zu verwenden, da sie meist die Logik zur Objektinstanziierung vor dem Entwickler verstecken, unter Umständen Konfigurationen erfordern, was Abhängigkeitszusammenhänge in Konfigurationsdateien bzw. -klassen verschiebt und der Code stark mit dem Framework verwoben wird.

5.3 Redundanzen durch Wertekopien

Abschnitt 4.2 beschreibt Variablendeklarationen und -definitionen in Java und COBOL. Wie erwähnt sind Datentyp und Repräsentation in COBOL eng miteinander verwoben. Dies sorgt in der Praxis häufig für redundanten Code, da nur zu Darstellungszwecken häufig Werte hin- und herkopiert werden. Dies hat nicht nur Auswirkungen auf die Lesbarkeit des Codes sondern auch auf die Performanz.

Laut Herrn Bonev ist beobachtbar, dass der dafür nötige **MOVE**-Befehl ca. 80% von COBOL-Code ausmacht, da häufig etliche Variablen für den selben Wert existieren und der Inhalt synchron gehalten werden muss. Auch beim Aufrufen von Unterprogrammen – wie in Abschnitt 4.5 beschrieben – wird dies, den Experten nach, häufig genutzt, um eine gewisse Sicherheit zu haben, dass Daten an anderer Stelle nicht verändert werden. Dazu werden Daten vor einem Aufruf in Variablen kopiert, das entsprechende Programm aufgerufen und anschließend wieder in die ursprünglichen Strukturen kopiert.

Eine Möglichkeit, welche laut Herrn Streit in Betracht gezogen werden sollte, ist es verschiedenen Variablen als *Union*-Verbunddatenstruktur, wie in Unterabschnitt 5.1.4 beschrieben, mithilfe des **REDEFINES**-Schlüsselworts anzulegen und so massiv **MOVE**-Befehle einzusparen. Diese Lösung bringt auch Sicherheit im Bezug auf Synchronisation, da das Kopieren nicht versehentlich vergessen werden kann. Häufig wird dies jedoch nicht eingesetzt, da sich Seiteneffekte und Fehler ergeben können, wenn Daten an unterschiedlichen Stellen modifiziert werden. Wird allerdings wie in Unterabschnitt 4.8.2 beschrieben auf klare Prä- und Suffixe geachtet, riet Herr Streit dazu **MOVE**-Befehle zu ersetzen um Code übersichtlicher und performanter zu gestalten.

Seltener lässt sich die Nutzung von **MOVE CORRESPONDING** beobachten. Dabei werden alle Untervariablen einer gegliederten Datenstruktur in eine andere Datenstruktur kopiert, die eine exakte namentliche Entsprechung in der Zielstruktur haben. Dies sei, den Experten nach, jedoch nicht weit verbreitet und sollte auch vermieden werden, da Programmlogik und -semantik abhängig von Variablennamen gemacht werden.

5.4 Externe Deklaration von Daten

Sowohl zur Wiederverwendbarkeit, als auch zur Erreichung einer gewissen Typsicherheit, ist es in vielen Programmiersprachen möglich, Datenstrukturen extern in einer eigenen Datei zu deklarieren und an verschiedenen Stellen wieder zu verwenden. Dies ist auch in COBOL und Java möglich, wenngleich sich die Ansätze stark unterscheiden. In Java ist diese Deklaration ein entscheidendes Sprachkonzept und so muss – wie bereits beschrieben – jede Klasse in einer eigenen Datei angelegt werden. In COBOL bietet dieses Konzept lediglich einen gewissen komfortableren Umgang mit Datendeklarationen, ist jedoch nicht zwingend notwendig.

In Java werden Klassen in sogenannten Packages organisiert. Um auf Klassen aus einem anderen Package zuzugreifen, ist ein Importieren der betreffenden Klasse notwendig. Dies geschieht wie in Listing 5.11 mithilfe des **import**-Statements.

```
1  package com.firstpackage;
2
3  import com.secondpackage.MySystemPart;
4
5  public interface MySystem {
6
7      public void doSomething(MySystemPart systemPart);
8
9  }
```

Listing 5.11: Import in Java

Das Interface `MySystem` nutzt dabei die in Package `com.secondpackage` enthaltene Klasse `MySystemPart`. Bei der Verwendung einer Klasse aus dem selben Package ist kein

zusätzliches **import**-Statement notwendig. Das Importieren geschieht dabei implizit über das Setzen des Packagenamens wie in Zeile 1 gezeigt.

COBOL bietet zur externen Deklaration von Daten das **COPY**-Schlüsselwort. Hiermit wird der Inhalt einer anderen Datei, eines sogenannten *COBOL-Copybook*, durch den Compiler an die Stelle des **COPY** kopiert. Das Verhalten ist also stark mit der Präprozessoranweisung **#include** in den Programmiersprachen C und C++ zu vergleichen.

```

1      01  ERROR-MESSAGES.
2          05  ERR-MSG PIC X(20) OCCURS 3 TIMES INDEXED BY MSG-INDEX.
3
4      01  ERROR-MESSAGES-INIT-VALUES.
5          05  FILLER PIC X(20) VALUE "Error 1 occurred".
6          05  FILLER PIC X(20) VALUE "Error 2 occurred".
7          05  FILLER PIC X(20) VALUE "Error 3 occurred".

```

Listing 5.12: COBOL-Copybook Datei (COPYBOOK.cpy)

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. COPY-EXAMPLE.
3
4      DATA DIVISION.
5      WORKING-STORAGE SECTION.
6          COPY COPYBOOK
7              REPLACING ERR-MSG          BY MSG
8                  "Error 1 occurred" BY "First error occurred!".
9
10     PROCEDURE DIVISION.
11     MAIN-PROCEDURE.
12         MOVE ERROR-MESSAGES-INIT-VALUES TO ERROR-MESSAGES.
13         ACCEPT MSG-INDEX.
14         DISPLAY MSG(MSG-INDEX).
15         STOP RUN.
16
17     END PROGRAM COPY-EXAMPLE.

```

Listing 5.13: Nutzung eines COBOL-Copybook

Listing 5.12 zeigt den Inhalt des Copybooks. Hier werden die Datenstrukturen definiert, die fortan an anderer Stelle genutzt werden sollen. In Listing 5.13 Zeile 6 wird das COBOL-Copybook in die **WORKING-STORAGE SECTION** kopiert, sodass die deklarierten Daten fortan im Programm genutzt werden können.

In diesem speziellen Fall wird zusätzlich Gebrauch der Schlüsselwörter **REPLACING** und **BY** gemacht. Dieses kann dazu verwendet werden, um Variablennamen oder Strings im dem Copybook auszutauschen.

Am Rande sei hier auch die Nutzung eines **FILLER** erwähnt. Hierbei handelt es sich um eine Variable, die nicht direkt verwendet werden kann, da sie keinen Namen hat. Wie im Beispiel sind FILLER oftmals Teile von größeren Strukturen.

Das Einbinden eines Copybooks ist jedoch auch innerhalb der **PROCEDURE DIVISION** möglich wie Listing 5.14 und Listing 5.15 darstellen. Somit wird es möglich Logik in Form von Code der wiederverwendet werden soll auszulagern und an verschiedenen Stellen zu nutzen.

```
1      EVALUATE VAR
2          WHEN 1 DISPLAY "First error occurred!"
3          WHEN 2 DISPLAY "Second error occurred!"
4          WHEN 3 DISPLAY "Third error occurred!"
5          WHEN OTHER DISPLAY "Unknown error code!"
6      END-EVALUATE.
```

Listing 5.14: COBOL-Copybook Datei (COPYBOOK-EVALUATE.cpy)

```
1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. COPY-EVALUATE.
3
4      DATA DIVISION.
5      WORKING-STORAGE SECTION.
6          01 INPUT-NUMBER PIC 9(2).
7
8      PROCEDURE DIVISION.
9      MAIN-PROCEDURE.
10         ACCEPT INPUT-NUMBER.
11         COPY COPYBOOK-EVALUATE REPLACING VAR BY INPUT-NUMBER.
12         STOP RUN.
13
14      END PROGRAM COPY-EVALUATE.
```

Listing 5.15: Nutzung von COPYBOOK-EVALUATE.cpy

❗ In Java ist die Auslagerung von Daten und Logik fester Bestandteil der Sprache. Dieses Konzept spiegelt sich auch in der Fähigkeiten der Modularisierung – vgl. Abschnitt 3.3 – wieder. COBOL bietet hierfür das Schlüsselwort **COPY**. Wenngleich dieses Konzept deutlich weniger mächtig ist wie das **import**-Statement in Java so ist es doch eine Möglichkeit Daten und Logik zu kapseln und auszulagern. Wie Herr Streit und Herr Bonev betonten, lässt sich entsprechende Nutzung jedoch in der Praxis nur selten beobachten und sollte weitaus öfter Verwendung finden.

6 Fazit (Sprechender Name je nach Ergebnissen)

Literatur

- [1] Franck Barbier. *Cobol Software Modernization*. Hoboken, NJ: ISTE Ltd/John Wiley and Sons Inc, 2014. ISBN: 978-1-84821-760-7. URL: <http://file.allitebooks.com/20160118/COBOL%20Software%20Modernization.pdf>.
- [2] Beat Balzli. *Cobol-Programmierer gesucht: Diese Unternehmen setzen auf IT-Rentner*. 19. Apr. 2017. URL: <https://www.wiwo.de/unternehmen/banken/cobol-programmierer-gesucht-diese-unternehmen-setzen-auf-it-rentner/19679936.html> (besucht am 25.04.2018).
- [3] Mo Budlong. *Teach Yourself Cobol in 21 Days*. 2nd ed. Indianapolis, IN: Sams Pub, 1997. 1056 S. ISBN: 978-0-672-31137-6.
- [4] John C Byrne und Jim Cross. *Java for COBOL Programmers*. OCLC: 567983849. Boston, MA: Charles River Media, 2009. ISBN: 978-1-58450-618-8. URL: <http://public.eblib.com/choice/publicfullrecord.aspx?p=3136100>.
- [5] Scott Colvey. „Cobol Hits 50 and Keeps Counting“. In: *The Guardian. Technology* (8. Apr. 2009). ISSN: 0261-3077. URL: <http://www.theguardian.com/technology/2009/apr/09/cobol-internet-programming> (besucht am 14.12.2017).
- [6] Michael Coughlan. *Beginning COBOL for Programmers*. The expert's voice in COBOL. OCLC: ocn874119151. Berkeley, California?: Apress, 2014. 556 S. ISBN: 978-1-4302-6253-4.
- [7] Georg Disterer, Hrsg. *Taschenbuch der Wirtschaftsinformatik: mit 71 Tabellen*. OCLC: 76162058. München: Fachbuchverl. Leipzig im Carl Hanser Verl, 2000. 663 S. ISBN: 978-3-446-21051-6.
- [8] E. Reed Doke u. a. *COBOL Programmers Swing with Java*. OCLC: 60573589. Cambridge; New York: Cambridge University Press, 2005. ISBN: 978-0-511-08240-5. URL: <http://www.books24x7.com/marc.asp?isbn=0521546842> (besucht am 31.01.2018).
- [9] Edsger Wybe Dijkstra. *How Do We Tell Truths That Might Hurt?* 18. Juni 1975. URL: <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD498.html> (besucht am 25.04.2018).
- [10] Florian Hamann. *In Banken Leben Dinosaurier: Cobol Kaum Totzukriegen*. 31. Jan. 2017. URL: <https://news.efinancialcareers.com/de-de/272568> (besucht am 25.04.2018).
- [11] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 978-0-201-63361-0.
- [12] IBM Corporation. *Enterprise COBOL for z/OS Language Reference*. 2013. URL: https://www.ibm.com/support/knowledgecenter/SSQ2R2_9.0.1/com.ibm.ent.cbl.zos.doc/PGandLR/igy5lr10.pdf (besucht am 04.05.2018).

- [13] *Ist Cobol die Programmiersprache der Zukunft?* 26. Jan. 1979. URL: <https://www.computerwoche.de/a/ist-cobol-die-programmiersprache-der-zukunft,1191656> (besucht am 23.04.2018).
- [14] Jia Walker. *COBOL Programming Fundamental*. Nov. 2004. URL: http://yusman.staff.gunadarma.ac.id/Downloads/files/33460/COBOL_Programming_Fundamental.pdf.
- [15] Stephen Kelly. *Cobol – Still Doing the Business after 50 Years*. 10. Juli 2009. URL: <https://www.ft.com/content/9c40ed12-569c-11de-9a1c-00144feabdc0> (besucht am 14.12.2017).
- [16] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1. Aufl. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 978-0-13-235088-4.
- [17] Michael Löwe. *Betriebliche Informationssysteme*. URL: <http://ux-02.ha.bib.de/daten/L%C3%B6we/Master/IIS/BetrieblicheInformationssysteme.pdf> (besucht am 05.05.2018).
- [18] Oracle. *Reading, Writing, and Creating Files (The Java™ Tutorials > Essential Classes > Basic I/O)*. URL: <https://docs.oracle.com/javase/tutorial/essential/io/file.html> (besucht am 06.05.2018).
- [19] Christian Rehn. *Tutorials Schreiben Oder: Wenn Sich Programmierer in Prosa Versuchen*. 29. Sep. 2009. URL: <http://www.christian-rehn.de/wp-content/uploads/2009/09/tutorials2.pdf> (besucht am 12.11.2017).
- [20] R. M. Richards. „Enhancing Cobol Program Structure: Sections vs. Paragraphs“. In: *ACM SIGCSE Bulletin* 16.2 (1. Juni 1984), S. 48–51. ISSN: 00978418. DOI: 10.1145/989341.989353. URL: <http://portal.acm.org/citation.cfm?doid=989341.989353> (besucht am 13.03.2018).
- [21] Uwe Rozanski. *Cobol 2002 ge-packt*. 1. Aufl. Die ge-packte Referenz. OCLC: 76681426. Bonn: mitp-Verl, 2004. 492 S. ISBN: 978-3-8266-1363-0.
- [22] Paul Rubens. *Why It's Time to Learn COBOL*. 1. Apr. 2016. URL: <https://www.cio.com/article/3050836/developer/why-its-time-to-learn-cobol.html> (besucht am 23.04.2018).
- [23] Nancy B. Stern, Robert A. Stern und James P. Ley. *COBOL for the 21st Century*. 11th ed. Hoboken, NJ: John Wiley & Sons, 2006. 1 S. ISBN: 978-0-471-72261-8.
- [24] Susanne Hackmack. *Objekte, Typen, Typhierarchien, Instanzen, Klassen*. 2018. URL: <http://www.fb10.uni-bremen.de/homepages/hackmack/clst/pdf/klassen.pdf> (besucht am 28.04.2018).
- [25] Patrick Thibodeau. *Should Universities Offer Cobol Classes?* 8. Apr. 2013. URL: <https://www.cio.com/article/2386947/education/should-universities-offer-cobol-classes-.html> (besucht am 23.04.2018).
- [26] *TIOBE Index / TIOBE - The Software Quality Company*. URL: <https://www.tiobe.com/tiobe-index/> (besucht am 25.04.2018).
- [27] University of Limerick; Department of Computer Science & Information Systems. *COBOL Programming - Tutorials, Lectures, Exercises, Examples*. URL: <http://www.csis.ul.ie/cobol/> (besucht am 14.04.2018).

Todo

Tests

4. -> viel als Schlüsselwort in COBOL, in Java nicht, sondern als Library

Kap. 4 COBOL Arithmetische Ausdrücke: Stellen eingeschränkt und unüblich -> Rundung

SetImplementierung in COBOL ineffizient und nicht generisch

S. 2 Abs. 2 Wieso teuer und riskant? -> Erklärungen, Belege!

Am Ende prüfen

Präzise Formulierungen: Was ist häufig zu sehen und was selten? Was ist Beobachtung was ist Empfehlung?

Keine Vorwärtsreferenzen

In der Literatur werden nicht alle Autoren aufgeführt

Caption Abstände

Seitenumbrüche + Paragraphenden "schön"

Overfull & underfull + Steht etwas über den Rand?

Keine TODOs mehr

Liste der noch zu erledigenden Punkte

3. -Langlebigkeit&Wartbarkeit -> Infrastruktur, nicht Sprache	11
3. -Verlässlichkeit -> Infrastruktur, nicht Sprache	11
3.3 Storyline	11
datenquellen, datenbanken	18
3. Batchbetrieb -> Schnittstellen (Host nicht COBOL)	18
EVA-Prinzip mit Zitat!	19
complete section	54
Tests	X
4. -> viel als Schlüsselwort in COBOL, in Java nicht, sondern als Library . .	X
Kap. 4 COBOL Arithmetische Ausdrücke: Stellen eingeschränkt und unüblich -> Rundung	X
SetImplementierung in COBOL ineffizient und nicht generisch	X
S. 2 Abs. 2 Wieso teuer und riskant? -> Erklärungen, Belege!	X
Präzise Formulierungen: Was ist häufig zu sehen und was selten? Was ist Beob- achtung was ist Empfehlung?	X
Keine Vorwärtsreferenzen	X
In der Literatur werden nicht alle Autoren aufgeführt	X
Caption Abstände	X
Seitenumbrüche + Paragraphenden "schön"	X

Overfull & underfull + Steht etwas über den Rand?	X
Keine TODOs mehr	X