

Universität Augsburg
Fakultät für angewandte Informatik
Institute for Software & Systems Engineering

Masterarbeit
Informatik und Multimedia

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

Antonio Grieco

Matrikelnummer: 1498410

antonio.grieco@gmx.de

Rosenaustraße 70

86152 Augsburg

16. April 2018

Erstgutachter: Prof. Dr. Alexander Knapp

Zweitgutachter: Prof. Dr. Bernhard Bauer

Betreuer: Jonathan Streit

Zusammenfassung

Abstract

Inhaltsverzeichnis

Zusammenfassung	I
Abstract	II
Inhaltsverzeichnis	III
Abbildungsverzeichnis	VII
Listings	VIII
1. Einleitung	1
1.1. Problemstellung	1
1.1.1. Wichtigkeit von COBOL	1
1.1.2. Bedeutung in der Lehre	2
1.1.3. Kontroverse Beurteilungen von COBOL	2
1.2. Ziel der Arbeit	2
2. Methodik	4
2.1. Vorhandene Literatur	4

2.2. Experten-Interviews	5
2.3. Entwicklungsumgebungen	6
3. Herausforderungen für COBOL und Java in betrieblichen Informationssystemen	7
3.1. Rechengenauigkeit	7
3.2. Dimensionierung	9
3.3. Modularisierung, Wiederverwendbarkeit & Variabilität	9
3.4. Schnittstellen	11
3.5. Datenquellen	11
3.6. Reporting	12
4. Vergleich wichtiger Sprachkonzepte	13
4.1. Konventionen	13
4.2. Programmablauf und Kontrollfluß	16
4.3. Deklaration von Variablen und Datentypen	23
4.3.1. Felder	26
4.4. Programmstruktur	28
4.5. Funktionen und Rückgabewerte	33
4.6. Dateien	37
4.7. Weitere Sprachkonzepte	37
4.7.1. Benannte Bedingungen	37

4.7.2. Mehrfachverzweigungen	41
4.7.3. Speicherausrichtung	45
4.7.4. Reorganisation von Daten	45
4.7.5. Implizierte Variablennamen	47
5. Pattern in COBOL und Java	50
5.1. Komplexe Datenstrukturen	50
5.1.1. Listen	50
5.1.2. Sets	53
5.1.3. Maps	56
5.1.4. Verbunddatenstrukturen	56
5.2. Wertezuweisungen	56
5.3. Callback-Muster	56
5.4. Singleton	60
5.5. Delegator	62
5.6. Dependency Injection	62
5.7. Externe Deklaration von Daten	64
6. Fazit (Sprechender Name je nach Ergebnissen)	67
Literatur	X

A. Zeitplan

XVII

Abbildungsverzeichnis

4.1. Strukturelle Bestandteile eines Java-Programms	31
4.2. Strukturelle Bestandteile eines COBOL-Programms	32
4.3. UML-Diagramm einer Aggregation	46

Listings

3.1.	Addition von float und double Variablen in Java	8
4.1.	Java main-Methode	16
4.2.	Beispiele für die Verwendung von break und continue in Java	18
4.3.	Verwendung von break und continue mit Label in Java	19
4.4.	Programmablauf in COBOL	21
4.5.	Programmablaufunterschiede in COBOL mit Sections und Paragraphs . .	22
4.6.	Variablendeklarationen in Java	23
4.7.	Variablendeklarationen in COBOL	25
4.8.	Felder in Java	27
4.9.	Felder in COBOL	27
4.10.	Variablendeklarationen mit verschiedenen Scopes	30
4.11.	Anonyme Klassen und Funktion in Java	32
4.12.	Methoden in Java	34
4.13.	Rekursion in Java	35
4.14.	Rekursion in COBOL	36
4.15.	Beispiel für COBOL Stufennummer 88	38
4.16.	Setzen von Werten mithilfe benannter Bedingungen	39
4.17.	Bedingte Werte in Java	40
4.18.	Setzen eines konstanten Wertes mit einem Enum in Java	41
4.19.	Mehrfachverzweigungen in Java	42
4.20.	Mehrfachverzweigungen in COBOL mit ALSO	43
4.21.	Mehrfachverzweigungen in COBOL als EVALUATE TRUE	44
4.22.	Stufennummer 66 und RENAMES -Befehl	46
4.23.	Keine implizierten Variablennamen in logischen Ausdrücken in Java	48
4.24.	Verwendung von Variablennamen in logischen Ausdrücken in Java	48
4.25.	Implizierte Variablennamen in COBOL	49
5.1.	ArrayList Beispiel in Java	51
5.2.	Einfache Listen Implementierung in COBOL	52
5.3.	HashSet Beispiel in Java	54
5.4.	Einfache Set Implementierung in COBOL	56

5.5. Observer und Observable in Java	58
5.6. Listener in Java	60
5.7. Singleton in Java	61
5.8. Dependency Injection in Java	63
5.9. Import in Java	64
5.10. COBOL-Copybook Datei (COPYBOOK.cpy)	65
5.11. Nutzung eines COBOL-Copybook	65
5.12. COBOL-Copybook Datei (COPYBOOK-EVALUATE.cpy)	66
5.13. Nutzung von COPYBOOK-EVALUATE.cpy	66

1. Einleitung

1.1. Problemstellung

Der folgende Abschnitt soll die Problemstellung verdeutlichen, welche der Arbeit zu Grunde liegt. Dazu wird erläutert, welche Wichtigkeit COBOL genießt und anschließend mit der Bedeutung, die der Sprache in der Lehre tatsächlich beigemessen wird und den Folgen davon für den Arbeitsmarkt, gegenübergestellt.

1.1.1. 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.”¹ 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 noch häufig an. In der britischen Tageszeitung The Guardian, zitiert der Autor Scott Colvey in seinem Artikel [4] 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-Leiters 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 damaligen Zeitpunkt wurden geschätzt

¹Dr. Horst Strunz, Fachbereichsleiter Standard-Systeme und Training des mbp, Mathematischer Beratungs- und Programmierungsdienst GmbH, Dortmund in *Ist Cobol die Programmiersprache der Zukunft?*. <https://www.computerwoche.de/a/ist-cobol-die-programmiersprache-der-zukunft,1191656>

200-mal mehr COBOL-Transaktionen ausgeführt als Google Suchanfragen verzeichnen konnte. [9] Nicht nur, dass COBOL im Jahr 2009 also einen enormen Marktanteil ausmachte wird also deutlich, sondern auch die Bedeutung für die Zukunft: Wieso sollte funktionierender Code mit Hilfe von teuren und riskanten Prozessen ersetzt werden? Da sich viele Unternehmen der 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 Anzahl

Beleg

des sich weltweit in Produktion befindlichen COBOL-Codes über die vergangenen Jahre sogar noch weiter an.

1.1.2. Bedeutung in der Lehre

Wie viele Unis bieten COBOL an?

Rente von COBOL-Entwicklern

Im TIOBE-Index² für Dezember 2017 rangiert COBOL auf Platz 29 mit einem Rating von 0.961%. Dieser Index wird auf Basis von Suchanfragen nach den entsprechenden Programmiersprachen, auf den meist frequentiertesten Internetseiten, erstellt. COBOL ist somit Teil von weniger als 1% der Suchanfragen.

Zusammenfassung

1.1.3. Kontroverse Beurteilungen von COBOL

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

²<https://www.tiobe.com/tiobe-index>

gegenübergestellt und verglichen werden. Außerdem sollen So soll es möglich sein vorhandenes Wissen über Softwareentwicklung 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

2. Methodik

2.1. Vorhandene Literatur

Die aufgeführte Literatur gibt oftmals mehr einen gesamtheitlichen Einblick in COBOL und bietet Hilfestellungen mit »Nachschlage-Charakter«. So wie in beispielsweise *Teach yourself Cobol in 21 days* [2], **university_of_limerick_department** [**university_of_limerick_department**] oder *COBOL Programming Fundamental* [8] werden häufig viele der möglichen Konstrukte in COBOL vorgestellt, mit Beispielen beschrieben und so ihre Verwendung gezeigt.

Neuere Literatur wie *COBOL for the 21st century* [14] betrachten dabei häufig zusätzlich Neuerungen wie die objektorientierte Verwendung von COBOL. *Cobol 2002 ge-packt* [13] hingegen stellt mehr ein Syntax-Wörterbuch dar, als eine wirkliche Beschreibung oder Einführung in COBOL.

Beginning COBOL for programmers [5] bietet den wohl umfassendsten Überblick, 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 dieser 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 zu finden.

Diese Arbeit soll daher 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.

Java for COBOL programmers [3]

COBOL programmers swing with Java [6]

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 wichtigen Feinheiten 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«.³ 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 um repräsentative Meinungen zu bestimmten vorausgewählten Fragen, sondern um die individuelle Einschätzung der Schwierigkeiten und Stolpersteine bei der Entwicklung, Wartung und dem Verständnis von bestehenden und neuen COBOL-Systemen, sowie der Einschätzung zu Parallelen und Diskrepanzen mit Java.

³<https://itestra.com/leistungen/software-renovation/>

Daher wurde kein Fragenkatalog ausgearbeitet sondern offener Input der erwähnten Personen gefordert, um die gewünschten subjektiven Meinungen zu bekommen. Abschriften dieser Interviews finden sich in ??.

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 Eclipse⁴ Umgebung verwendet. Dabei handelt es sich um einen etablierte IDE, welche eine Vielzahl von Funktionen zur Entwicklung und zum Debugging liefert.

Der COBOL-Code dieser Arbeit wurde in der OpenCobolIDE⁵ entwickelt. Dabei handelt es sich um eine minimalistische IDE, welche zum Beispiel Syntax-Highlighting oder eine übersichtliche Darstellung von Fehlern bietet. Der darunterliegende Compiler GnuCOBOL⁶ 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 ein reeller Kunde in Betrieb hat und der Zugriff oft mühsam, teuer oder schlichtweg nicht möglich ist.

⁴<http://www.eclipse.org/>

⁵<https://github.com/OpenCobolIDE/OpenCobolIDE>

⁶<https://sourceforge.net/projects/open-cobol/>

3. Herausforderungen für COBOL und Java in betrieblichen Informationssystemen

Bei der Entwicklung von betrieblichen Informationssystemen sehen sich Entwickler mit ein paar grundlegenden Fragen und Anforderungen an die einzusetzenden Technologien und Programmiersprachen konfrontiert. Dieses Kapitel soll einen Überblick über die wichtigsten Entscheidungskriterien geben und aufzeigen, welchen Herausforderungen sich COBOL und Java in diesen Informationssystemen stellen müssen.

3.1. Rechengenauigkeit

Vor allem in betrieblichen Informationssystemen, die oftmals an bestimmten Stellen 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.

Rechengenauigkeit in Java

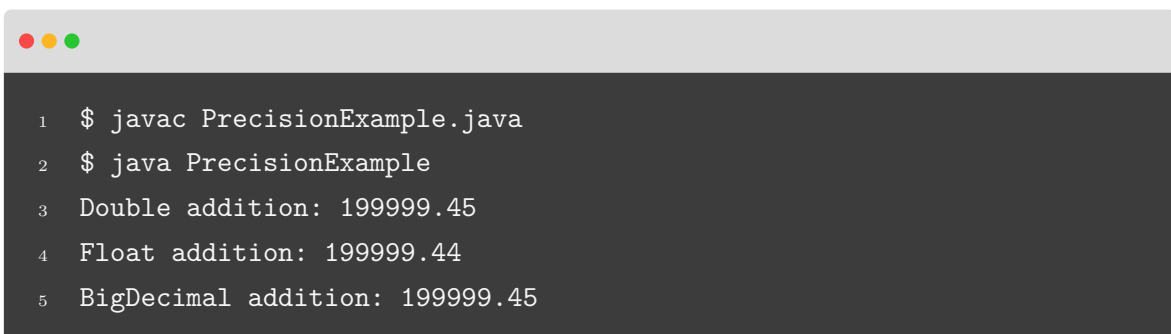
Java verwendet zur Speicherung von Fließkommazahlen, wie viele andere moderne Programmiersprachen, den *IEEE 754*-Standard. Ohne näher auf diesen eingehen zu wollen, legt dieser einen Algorithmus fest, mit dessen Hilfe Variablen in einem Speicherbereich repräsentiert werden. Dieser Speicherbereich kann sich je nach Datentyp und Programmiersprache unterscheiden, ist jedoch an sich stets von fester Größe. Daher ist es nicht möglich beliebig genaue Werte abzubilden.

```
1  import java.math.BigDecimal;
2
3  public class PrecisionExample {
4
```

```

5     public static void main(String[] args) {
6         double dValue1 = 99999.8;
7         double dValue2 = 99999.65;
8         System.out.println("Double addition: " + (dValue1 + dValue2));
9
10        float fValue1 = 99999.8f;
11        float fValue2 = 99999.65f;
12        System.out.println("Float addition: " + (fValue1 + fValue2));
13
14        BigDecimal bValue1 = new BigDecimal("99999.8");
15        BigDecimal bValue2 = new BigDecimal("99999.65");
16        System.out.println("BigDecimal addition: " +
17            ↪ (bValue1.add(bValue2).toPlainString()));
18    }

```



```

1  $ javac PrecisionExample.java
2  $ java PrecisionExample
3  Double addition: 199999.45
4  Float addition: 199999.44
5  BigDecimal addition: 199999.45

```

Listing 3.1.: Addition von float und double Variablen in Java

Listing 3.1 zeigt das bereits anhand eines sehr einfachen Beispiels. Bereits die Addition von zwei Zahlen mit einer bzw. zwei Dezimalstellen, welche zugegebenermaßen bewusst so gewählt wurden, legt die Problematik offen. So unterscheiden sich die Ergebnisse der Berechnung je nach Datentyp, was auf die Abbildung der Zahlen im Speicher zurückzuführen ist. Um Sicherheit bei der Berechnung zu erhalten ist es stets nötig den Datentyp `BigDecimal`, mitsamt des damit verbundenen Speicher- und Laufzeit-Overheads, wie im letzten Teil des Beispiels gezeigt, zu verwenden. Wichtig hierbei ist, dass die Definition wenn möglich über den Konstruktor erfolgt, der eine String-Repräsentation eines Wertes erhält. Andere Konstruktoren wie z.B. `BigDecimal(double)` können zu Problemen führen, da der übergebene Wert durch die Speicherung als z.B. `double` bereits an Genauigkeit verlieren kann.

Rechengenauigkeit in COBOL

Wie später in Abschnitt 4.3 noch genauer ausgeführt ist der Entwickler und nicht die Sprache in COBOL dafür zuständig genau festzulegen, wie viele Dezimalstellen eine Variable speichern soll. Diese Eigenschaft in Verbindung mit der Speicherrepräsentation der Daten in COBOL führt zu einer praktisch absolut exakten Genauigkeit. Jedes Zeichen wird in COBOL separat gespeichert. 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 wieder auch zeichenweise gespeichert und lassen so keine Rundungsfehler oder Fehler aufgrund von unzureichendem Speicherplatz zur Abbildung zu.

3.2. Dimensionierung

3.3. Modularisierung, Wiederverwendbarkeit & Variabilität

wiederverwendbarkeit 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 dabei nicht mehrmals geschrieben werden, was auch das spätere Einarbeiten in ein Projekt erleichtert, da der Projektumfang deutlich geringer wird. Zudem ist, sei es um verschiedene Mandanten oder verschiedene 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 zum Beispiel Kapselung, Polymorphie oder Aggregation/Komposition sorgen dafür, dass Code in hohem Maße wiederverwendet werden kann. Dabei ist vor allem die Gliede-

rung in Klassen und Funktionen (siehe Abschnitt 4.5) ausschlaggebend. Des weiteren können Bibliotheken als Java Archive (oft 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 nur über Packages (siehe Abschnitt 4.4) 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 eine gewisse Variabilität des Programmes 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. Das sorgt dafür, dass kompilierte Programme auf allen Systemen mit JVM ausführbar sind und so weiterverteilt werden können ohne neu kompiliert zu werden. Voraussetzung dafür ist wie bereits erwähnt eine JVM. Diese 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 befragter Experte betonte, sieht man daher 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 »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 4.1 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 in COBOL firm sind. Daher wird diese Spracherweiterung in der vorliegenden Arbeit nicht behandelt.

3.4. Schnittstellen

3.5. Datenquellen

Strukturierte Daten, Dateien, Datenbanken

3.6. Reporting

4. Vergleich wichtiger Sprachkonzepte

4.1. Konventionen

Im folgenden Abschnitt sollen bestimmte Konventionen dargestellt werden, die in COBOL bzw. Java gelten.

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.

Grund?

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 laut Experten um eine Best-Practice-Methode, um den Code verständlicher und leichter lesbar zu machen.

So ist es üblich den Namen einer **SECTION** oder eines Paragraph mit einem Präfix zu versehen und die darin genutzten Variablen mit demselbigen zu kennzeichnen. Beispielsweise sollte eine Variable 100-**VALUE** nur in der **SECTION** 100-**PROCESS** verwendet werden.

Oft finden sich 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.

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. Oftmals findet sich in der Praxis COBOL-Code welcher gekürzte Schlüsselwörter verwendet.

In Java ist ein verkürzen von Schlüsselwörtern hingegen nicht möglich.

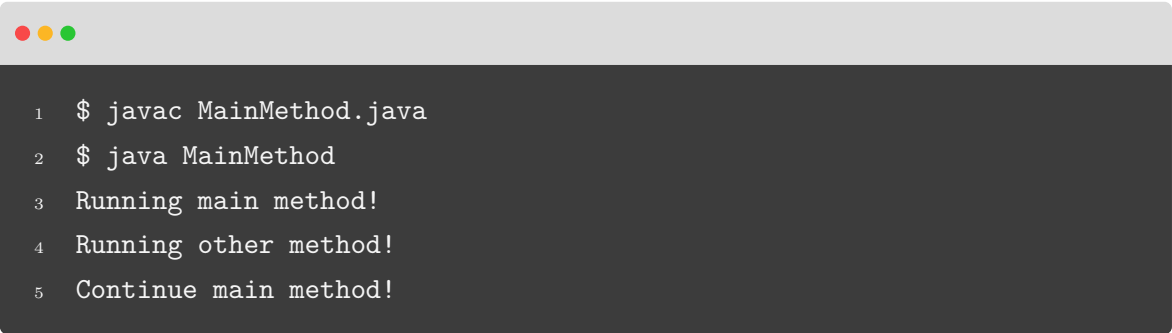
Ein Statement pro Zeile, Einrückungen... -> Struktur / Aufbau des Codes (Verweis auch auf das Strukturkapitel, welches Strukturen enthält die nicht konventionell sondern fest vorgegeben sind.)

4.2. Programmablauf und Kontrollfluß

Durch die in Abschnitt 4.5 erläuterten Unterschiede ergeben sich auch im Programmablauf Diskrepanzen. Während Java einen definierten Einstiegspunkt in ein Programm hat (`main`-Methode) wird ein COBOL-Programm stets sequenziell von oben nach unten abgearbeitet und durchlaufen. Diese Abarbeitung beginnt am Anfang der **PROCEDURE DIVISION**.

Programmablauf in Java

```
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  Continue main method!
```

Listing 4.1.: Java main-Methode

Listing 4.1 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`

reference exception section

– wird stets mit der nächsten Anweisung nach dem Funktionsaufruf fortgefahren.

Weitere Schlüsselwörter die den Kontrollfluß 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 4.2 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 4.2.: 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 4.7.2 zeigt eine weitere Verwendung des **break**-Statements.

```

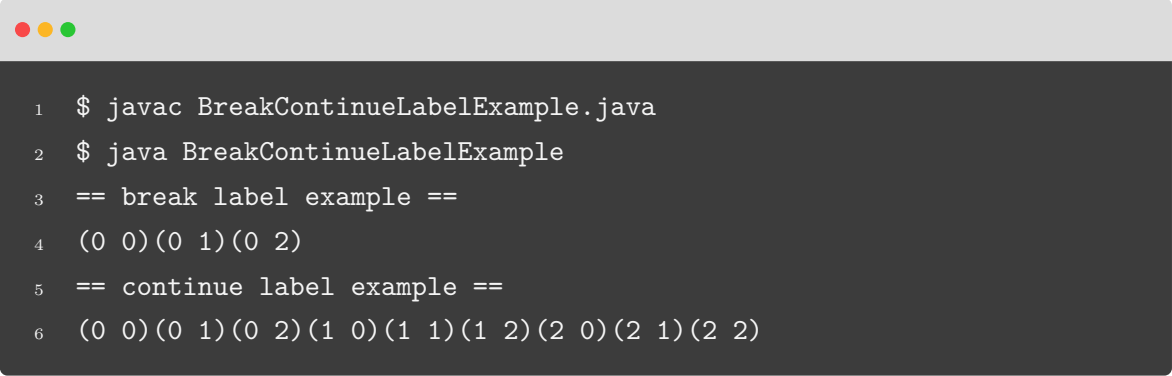
1  public class BreakContinueLabelExample {
2
3      public static void main(String[] args) {
4          breakLabelExample();
5          continueLabelExample();
6      }
7
8      private static void breakLabelExample() {
9          System.out.println("\n== break label example == ");
10         outerLoop:
11         for (int oCounter = 0; oCounter < 10; oCounter++) {
12             for (int iCounter = 0; iCounter < 10; iCounter++) {
13                 if (oCounter > 2 || iCounter > 2)
14                     break outerLoop;
15                 System.out.print(String.format("(%s %s)", oCounter, iCounter));
16             }
17         }
18     }
19
20     private static void continueLabelExample() {
21         System.out.println("\n== continue label example == ");
22         outerLoop:
23         for (int oCounter = 0; oCounter < 10; oCounter++) {
24             for (int iCounter = 0; iCounter < 10; iCounter++) {
25                 if (oCounter > 2 || iCounter > 2)
26                     continue outerLoop;
27                 System.out.print(String.format("(%s %s)", oCounter, iCounter));

```

```

28         }
29     }
30 }
31 }

```



```

1 $ javac BreakContinueLabelExample.java
2 $ java BreakContinueLabelExample
3 == break label example ==
4 (0 0)(0 1)(0 2)
5 == continue label example ==
6 (0 0)(0 1)(0 2)(1 0)(1 1)(1 2)(2 0)(2 1)(2 2)

```

Listing 4.3.: Verwendung von break und continue mit Label in Java

Deutlich unüblicher – jedoch nicht weniger relevanter – 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. Dies zeigt Listing 4.3.

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.

Programmablauf in COBOL

In COBOL gestaltet sich der Programmablauf gänzlich anders. 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 STATEMENT**

NEXT STATEMENT erklären

-Anweisungen verändert werden kann. An dieser Stelle sei ausdrücklich erwähnt, dass die Verwendung des **GO TO**-Befehls unter allen Umständen unerlassen werden sollte, da ansonsten sehr schwer verständlicher und wartbarer Code entsteht! Leider findet man sich in der Praxis oftmals mit Code konfrontiert, der **GO TO**-Befehle zur Steuerung des Ablaufs verwendet.

Die Ausführung eines COBOL-Programms endet beim Erreichen einer **STOP RUN**-Anweisung oder mit dem Ende des Programms (**END PROGRAM**).

Ein Unterprogramm wird mit **CALL** aufgerufen und gibt hingegen mit **GOBACK** die Kontrolle zurück an das aufrufende Programm.

```

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 4.4.: Programmablauf in COBOL

Die beiden Ausführungen von Listing 4.4 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 4.4 beschrieben besteht ein COBOL-Programm aus verschiedenen strukturellen Komponenten. Diese haben auch einen gewissen Einfluss auf den Programmablauf. Dies soll das Beispiel in Listing 4.5 veranschaulichen.

```

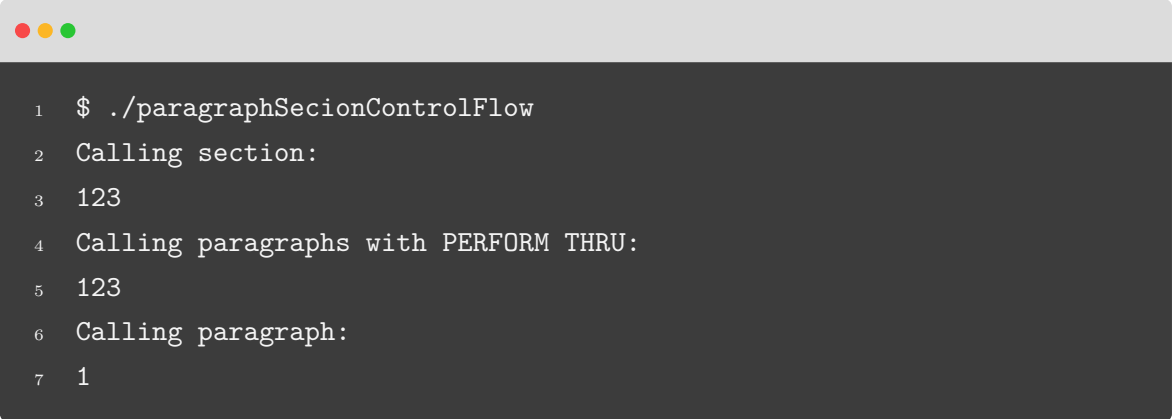
1      IDENTIFICATION DIVISION.
2      PROGRAM-ID. PARAGRAPH-SECTION-CONTROL-FLOW.
3
4      PROCEDURE DIVISION.
5      MAIN-PROCEDURE.
6          DISPLAY "Calling section:".
7          PERFORM TEST-SECTION.
8          DISPLAY SPACE.
9          DISPLAY "Calling paragraphs with PERFORM THRU:".
10         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  $ ./paragraphSectionControlFlow
2  Calling section:
3  123
4  Calling paragraphs with PERFORM THRU:
5  123
6  Calling paragraph:
7  1

```

Listing 4.5.: 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. Dieses Vorgehen wurde auch von Richards bereits 1984 als best-practice beschrieben [12]. 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.

In COBOL ist die bereits angesprochene nebenläufige Ausführung nicht möglich. Ein COBOL-Programm führt Verarbeitungsschritte stets sequenziell aus und erlaubt keine parallelen Ausführungen. Über einen Transaktionsmonitor

Transaktionsmonitor beschreiben

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.

4.3. Deklaration von 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 wie z.B. `int` oder `double`, aber auch komplexe Objekttypen sein.

```

1  public class VariableExample {
2
3      int primitiveClassVariable = 0;
4      VariableExample complexClassVariable = null;
5
6      public static void main(String[] args) {
7          int primitiveLocalVariable = 1;
8          VariableExample complexLocalVariable = new VariableExample();
9      }
10
11 }
```

Listing 4.6.: Variablendeklarationen in Java

Dabei werden in Listing 4.6 einige Konzepte deutlich gemacht:

- Variablen können wie in Abschnitt 4.4 sowohl als Teil einer Klasse als auch lokal innerhalb einer Methode deklariert 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 einer Variable eines bestimmten Datentyps 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.


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** 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 4.7.: Variablendeklarationen in COBOL

Listing 4.7 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.

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 \equiv 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**.

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.

Weitere mögliche Bestandteile dieser Deklaration werden an den entsprechenden Stellen dieser Arbeit erläutert.

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.

4.3.1. Felder

Eine zentrale Datenstruktur in der Programmierung stellen Felder oder Arrays 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 4.8.: Felder in Java

Zeile 4 in Listing 4.8 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 4.9. 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.

```

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 4.9.: 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. Jedes Element der ersten Dimension 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. Programmstruktur

Dieser Abschnitt behandelt die strukturellen Unterschiede von Java- und COBOL-Programmen. Dazu wird auf den Scope von Variablen eingegangen und erläutert in welche Einheiten sich die Programme der jeweiligen Sprache aufteilen lassen. Der sog. Scope, zu deutsch Gültigkeitsbereich, gibt in der Programmierung an, in welchem Bereich eine Variable gültig ist.

Enum und Interfaces beschreiben

Java

Scope

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 4.10.

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 4.10.: Variablendeklarationen mit verschiedenen Scopes

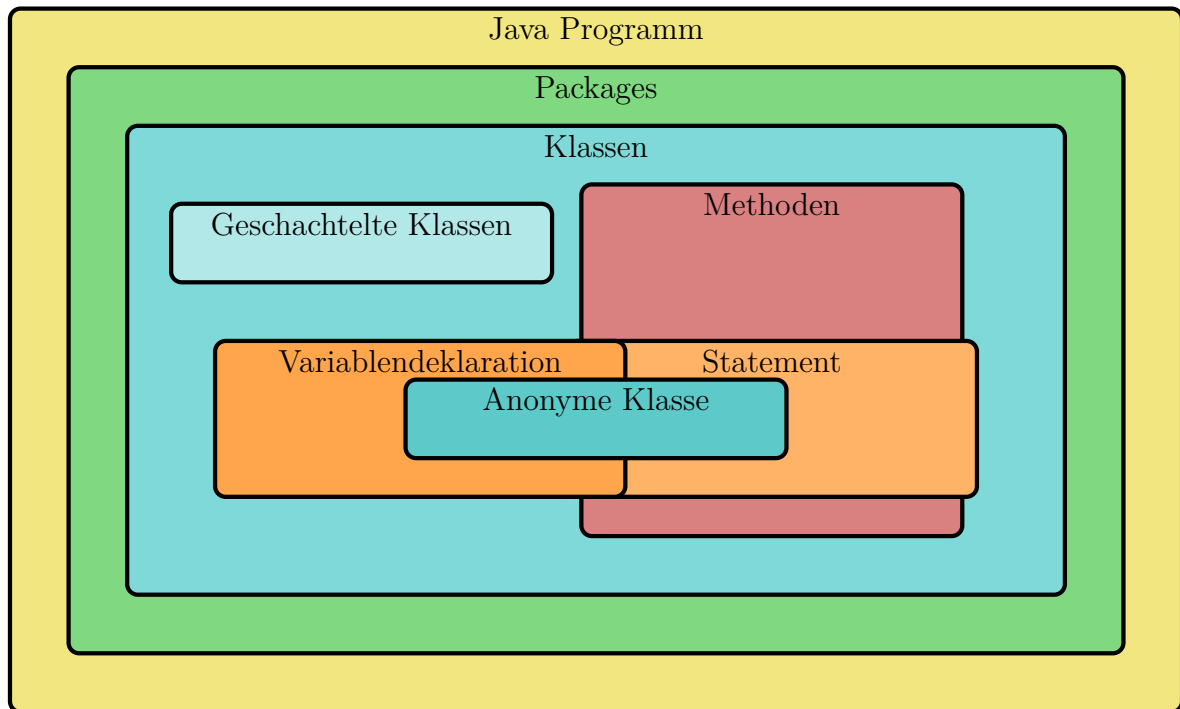
Struktur

Auch zeigt das Beispiel in Listing 4.10 die wichtigsten Konzepte der Strukturierung von Java-Code. 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.



Align diagram components right

Abbildung 4.1.: Strukturelle Bestandteile eines Java-Programms

Methoden wiederum bestehen aus einzelnen Statements. Zu erwähnen ist, dass Variablendeklarationen auch ein Statement darstellen, jedoch sind, als Teil einer Klasse, keine anderen Statements als Variablendeklarationen gültig, weshalb auch an dieser Stelle diese Unterscheidung getroffen wird. Statements die aus Variablendeklarationen, Zuweisungen oder Methodenaufrufen bestehen, müssen im Gegensatz zu Block-Statements, wie z.B. `if`-Statements, stets mit einem Semikolon beendet werden.

```

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

```

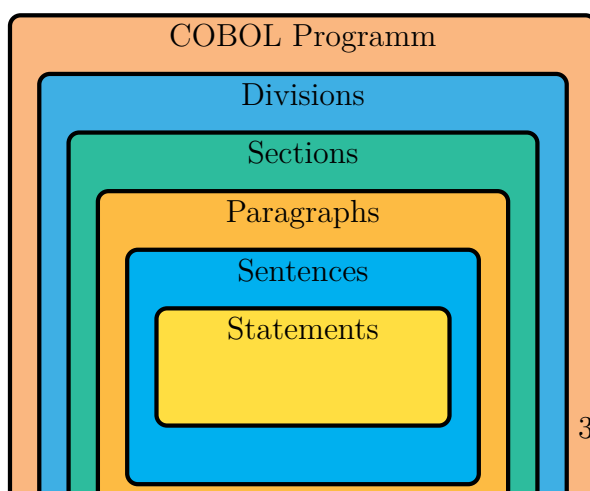
Listing 4.11.: Anonyme Klassen und Funktion in Java

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 4.11 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.

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

COBOL

Darstellung überarbeiten



Nebestehende Abbildung 4.2 zeigt die strukturellen Bestandteile eines COBOL-Programms. Dabei gibt es folgende vier fest definierte **DIVISIONS**:

- **IDENTIFICATION DIVISION** Hier werden grundlegende Daten zum Pro-

gramm, 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.

DIVISIONS, SECTIONS, PARAGRAPHS, SENTENCES, STATEMENTS
-> TERMINATED WITH DOT.

Einteilung einer Zeile. Erste X Zeichen reserviert usw. 80 Zeichen länge....

Variablen sind immer global in COBOL

4.5. Funktionen 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 4.2 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 4.12.: Methoden in Java

Listing 4.12 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 übergebenden Parametern. Um eine Funktion zu verlassen wird das Schlüsselwort `return` verwendet. In Verbindung mit einer Variablen oder einem Literal gibt dieses Statement einen Rückgabewert an die aufrufende Funktion zurück.

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 4.13.: Rekursion in Java

Die Funktionen `facultyRecursive` und `facultyIterative` in Listing 4.13 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 stets 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. 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 Abschnitt 4.4 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 4.14.: Rekursion in COBOL

Listing 4.14 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.

4.6. Dateien

4.7. Weitere Sprachkonzepte

4.7.1. 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 4.15 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.

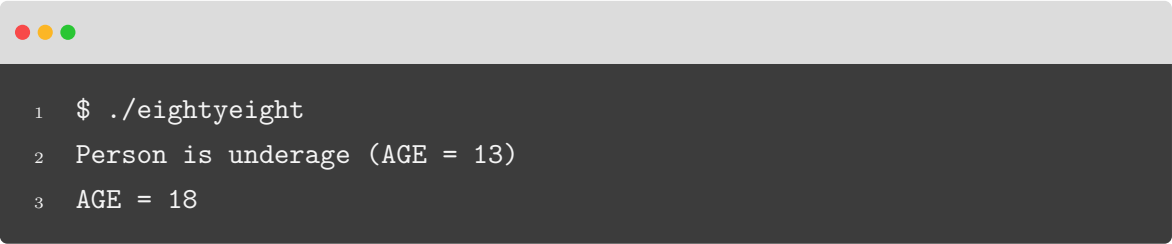
Der so entstandene Wahrheitswert kann folglich immer dann verwendet werden, wenn getestet werden soll, ob die Variable **AGE** im Bereich zwischen 0 und 17 bzw. zwischen 17 und 99 liegt. Also um zu testen, ob das Alter einer minder- oder volljährigen Person entspricht.

```
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 4.15.: 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 4.15 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 4.16.

```

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 4.16.: 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 4.17 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 4.17.: 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 4.18 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 4.18.: Setzen eines konstanten Wertes mit einem Enum in Java

4.7.2. 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 4.19 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
23         default:
24             System.out.println("Month is not valid");
25             break;
26     }
27 }
28 }
```

Listing 4.19.: 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 gesprungen. Dies zeigt sich in den Zeilen 6–9 und 17f. von Listing 4.19. 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 4.20 und Listing 4.21 sind hierbei semantisch gleich, obwohl das **EVALUATE**-Konstrukt jeweils leicht anders verwendet wird.

Das folgende Listing 4.20 führt die Verwendungsmöglichkeit an, die dem **switch-case**-Konstrukt in Java am nächsten kommt. Nach dem **EVALUATE**-Schlüsselwort werden Variablenamen 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 4.20.: Mehrfachverzweigungen in COBOL mit ALSO

Die erste Besonderheit, die Listing 4.20 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 4.21.: Mehrfachverzweigungen in COBOL als EVALUATE TRUE

Listing 4.21 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 4.20 als auch **TRUE**-Vergleiche wie in Listing 4.21 zusammen, mithilfe des **ALSO**-Schlüsselworts verwendet werden können.

4.7.3. 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 ungenutzt, 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.

4.7.4. Reorganisation von Daten

Wie bereits in Abschnitt 4.3 erwähnt beinhaltet COBOL drei Stufennummern denen eine besondere Rolle zuteilwird. Dieser Abschnitt soll daher kurz die COBOL Stufennummer 66 beschreiben.


In Listing 4.22 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 4.22.: 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.

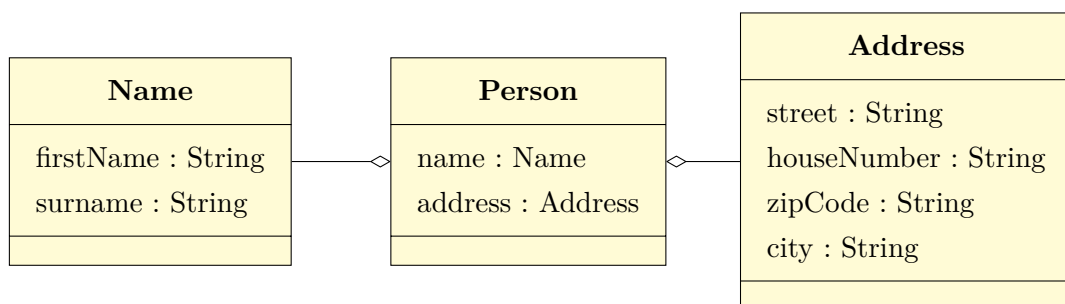


Abbildung 4.3.: UML-Diagramm einer Aggregation

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.

4.7.5. Implizierte Variablennamen

Dieser Abschnitt behandelt einen Fehler, der typischerweise zu Beginn in der Softwareentwicklung beobachtet werden kann. Listing 4.23 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 4.23.: Keine implizierten Variablennamen in logischen Ausdrücken in Java

Versucht man Listing 4.23 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 4.24.: Verwendung von Variablennamen in logischen Ausdrücken in Java

Listing 4.24 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 4.25.: Implizierte Variablennamen in COBOL

Listing 4.25 stellt in Zeile 11 die Verwendung von implizierten Variablennamen in COBOL dar, die im Gegensatz zu Java möglich ist.

5. Pattern in COBOL und Java

5.1. Komplexe Datenstrukturen

5.1.1. Listen

Während Unterabschnitt 4.3.1 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

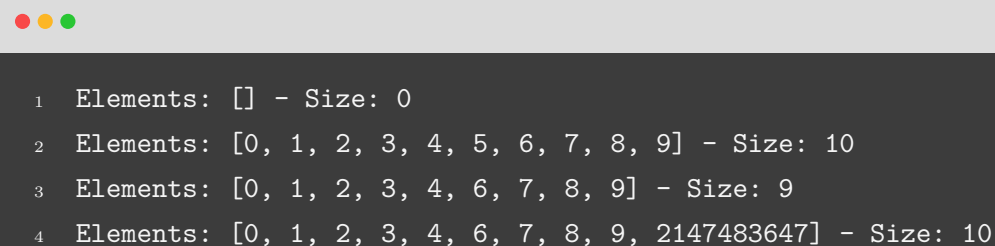
In Java kann das `List`-Interface implementiert werden bzw. ein Objekt dieser Implementierung instanziiert werden, um eine Liste variabler Größe zu erhalten. Die wohl gebräuchlichste Implementierung dieses Interfaces stellt die Klasse `ArrayList` dar. Intern hält diese – wie der Name schon vermuten lässt – 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 Löschooperationen realisiert.

5.1.3. Maps

5.1.4. Verbunddatenstrukturen

union- & struct-like types

5.2. Wertezuweisungen

MOVE in COBOL

5.3. 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 5.5 zeigt die Handhabung des `Observer`-Pattern [7] 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 4.4 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 5.5.: 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<>();
41
42         void registerListener(TimeListener timeListener) {
43             timeListeners.add(timeListener);
44         }
45
46         void notifyListeners(long value) {

```

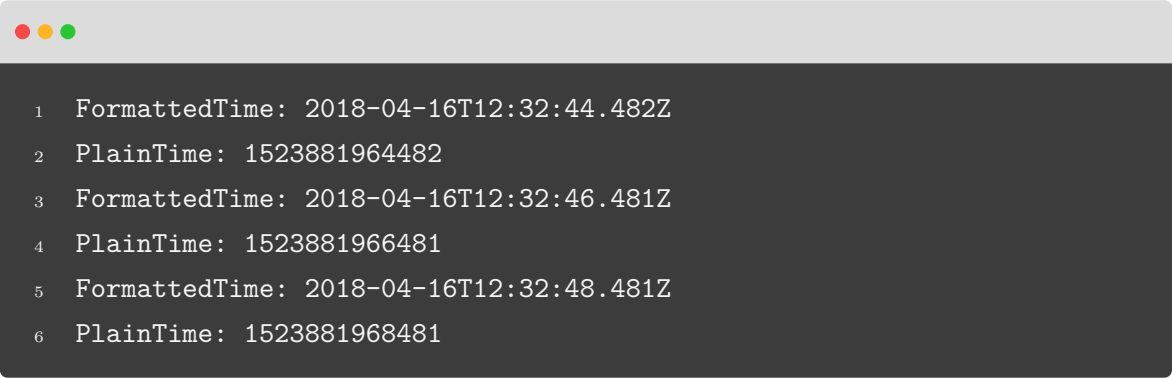
```

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 5.6.: Listener in Java

Die Aufgabe beider Beispiele ist die gleiche. Jedoch verdeutlicht Listing 5.6 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

```

5.4. Singleton

»Alles ist ein Objekt.«⁷ 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

⁷<http://www.fb10.uni-bremen.de/homepages/hackmack/clst/pdf/klassen.pdf>

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 5.7.: Singleton in Java

Listing 5.7 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.

5.5. Delegator

5.6. Dependency Injection

In einer objektorientierten Programmierungsumgebung werden Funktionalitäten in Klassen gekapselt und so die Wiederverwendbarkeit erhöht. Dies wurde bereits in ?? 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* [10] 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 von außerhalb 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 5.8.: Dependency Injection in Java

Listing 5.8 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 [10] 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.

5.7. 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 Konzept und so muss 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 5.9 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 5.9.: Import in Java

Das Interface `MySystem` nutzt dabei die Klasse `MySystemPart` aus dem Package `com.secondpackage`. 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 5.10.: 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 5.11.: Nutzung eines COBOL-Copybook

Listing 5.10 zeigt den Inhalt des Copybooks. Hier werden die Datenstrukturen definiert, die fortan an anderer Stelle genutzt werden sollen. In Listing 5.11 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 5.13 und Listing 5.12 darstellt.

```

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 5.12.: 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 5.13.: Nutzung von COPYBOOK-EVALUATE.cpy

Somit wird es möglich Code der wiederverwendet werden soll auszulagern und an verschiedenen Stellen zu nutzen.

6. Fazit (Sprechender Name je nach Ergebnissen)

Literatur

- [1] Franck Barbier. *Cobol software modernization*. Hoboken, NJ: ISTE Ltd/John Wiley und Sons Inc, 2014. ISBN: 978-1-84821-760-7. URL: <http://file.allitebooks.com/20160118/COBOL%20Software%20Modernization.pdf>.
- [2] Mo Budlong. *Teach yourself Cobol in 21 days*. 2nd ed. Indianapolis, IN: Sams Pub, 1997. 1056 S. ISBN: 978-0-672-31137-6.
- [3] 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>.
- [4] Scott Colvey. „Cobol hits 50 and keeps counting“. In: *The Guardian* (8. Apr. 2009). ISSN: 0261-3077. URL: <http://www.theguardian.com/technology/2009/apr/09/cobol-internet-programming> (besucht am 14.12.2017).
- [5] 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.
- [6] E. Reed Doke u. a. *COBOL programmers swing with Java*. OCLC: 60573589. Cambridge; New York: Cambridge University Press, 2005. ISBN: 978-0-511-08240-5 978-0-511-08150-7 978-0-511-54698-3 978-0-521-83781-1 978-1-280-43206-4. URL: <http://www.books24x7.com/marc.asp?isbn=0521546842> (besucht am 31.01.2018).
- [7] 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.
- [8] Jia Walker. *COBOL Programming Fundamental*. Nov. 2004. URL: file:///home/toni/Downloads/COBOL_Programming_Fundamental.pdf.
- [9] Stephen Kelly. *Cobol – still doing the business after 50 years*. Financial Times. 10. Juli 2009. URL: <https://www.ft.com/content/9c40ed12-569c-11de-9a1c-00144feabdc0> (besucht am 14.12.2017).
- [10] 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.
- [11] 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>.
- [12] 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).

- [13] 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.
- [14] 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.
- [15] 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).

Nicht von Kununu zitieren

Generics beschreiben

Mein erstes COBOL-Programm -> IDE-Kapitel

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

3. -Verlässlichkeit -> Infrastruktur, nicht Sprache

3. Batchbetrieb -> Schnittstellen (Host nicht COBOL)

5.3 -> 4. Deutlicher, dass Java-Klassen "mächtiger" sind

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

Einleitungsüberschrift ändern

Interviews: Umfang ansprechen und rausnehmen + zitieren

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

S. 28 (Java Komponenten) Initializer ({}) in Klassen

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

SetImplementierung in COBOL ineffizient und nicht generisch

Bibkonzept in Java: 1. Erwähnen, dass es das generell gibt mit Bibs zu arbeiten 2. oo führt zu Algorithmen mit beliebigen Objekten, die eine bestimmte Eigenschaft erfüllen

TIOBE: vorbelastet, weil Hostentwickler Internet nicht als nr.1 quille haben

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

Intention der Arbeit klar machen: Nicht Java-Entwickler umschulen, sondern schulen für Migration, Wartung und Flexibilität

S. 5 “wichtige Feinheiten”

S. 4 “gesamtheitlich” falscher Begriff -> sehr detailliert

S. 5 “repräsentativ” -> keine Stichprobe / Fragenkatalog

3.3 Storyline

S. 14 Präfixe (Identifier nicht mit Zahl)

Keine Vorwärtsreferenz!

Rechengenauigkeit: unterschied zwischen binärem/dezimalem Fließ-/Festkomma => Geld -> Festkommasemantik

Programmablauf ist gar nicht so unterschiedlich, sieht aber so aus. Nicht mit unterschieden einsteigen sondern mit Gemeinsamkeiten!

S. 18 Gliedern oder komprimieren: goto/labels Randnotizen in Java, Continue Unterschiede COBOL/Java, goto in COBOL kann unterschiedlich verwendet werden. Sprünge auf EXIT Paragraphen, da kein return.

Liste der noch zu erledigenden Punkte

Beleg	2
Wie viele Unis bieten COBOL an?	2
Rente von COBOL-Entwicklern	2
Zusammenfassung	2
<i>Java for COBOL programmers</i> [3]	5
<i>COBOL programmers swing with Java</i> [6]	5
Strukturierte Daten, Dateien, Datenbanken	11
Grund?	14
Ein Statement pro Zeile, Einrückungen... -> Struktur / Aufbau des Codes (Verweis auch auf das Strukturkapitel, welches Strukturen enthält die nicht konventionell sondern fest vorgegeben sind.)	15
reference exception section	17
NEXT STATEMENT erklären	19
Transaktionsmonitor beschreiben	22
Enum und Interfaces beschreiben	28
Align diagram components right	31
Darstellung überarbeiten	32
DIVISIONS, SECTIONS, PARAGRAPHS, SENTENCES, STATEMENTS -> TERMINATED WITH DOT.	33
Einteilung einer Zeile. Erste X Zeichen reserviert usw. 80 Zeichen länge.... . . .	33

Variablen sind immer global in COBOL	33
union- & struct-like types	56
MOVE in COBOL	56
Nicht von Kununu zitieren	XII
Generics beschreiben	XII
Mein erstes COBOL-Programm -> IDE-Kapitel	XII
3. -Langlebigkeit&Wartbarkeit -> Infrastruktur, nicht Sprache	XII
3. -Verlässlichkeit -> Infrastruktur, nicht Sprache	XII
3. Batchbetrieb -> Schnittstellen (Host nicht COBOL)	XII
5.3 -> 4. Deutlicher, dass Java-Klassen “mächtiger” sind	XII
4. -> viel als Schlüsselwort in COBOL, in Java nicht, sondern als Library	XII
Einleitungsüberschrift ändern	XII
Interviews: Umfang ansprechen und rausnemen + zitieren	XII
Präzisere Formulierungen: Was ist häufig zu sehen und was selten? Was ist Beobachtung was ist Empfehlung?	XII
S. 28 (Java Komponenten) Initializer ({}) in Klassen	XII
Kap. 4 COBOL Arithmetische Ausdrücke: Stellen eingeschränkt und unüblich -> Rundung	XII
SetImplementierung in COBOL ineffizient und nicht generisch	XII
Bibkonzept in Java: 1. Erwähnen, dass es das generell gibt mit Bibs zu arbeiten 2. oo führt zu Algorithmen mit beliebigen Objekten, die eine bestimmte Eigenschaft erfüllen	XII
TIOBE: vorbelastet, weil Hostentwickler Internet nicht als nr.1 qule haben	XII
S. 2 Abs. 2 Wieso teuer und riskant? -> Erklärungen, Belege!	XII

Intention der Arbeit klar machen: Nicht Java-Entwickler umschulen, sondern schulen für Migration, Wartung und Flexibilität	XII
S. 5 “wichtige Feinheiten”	XIII
S. 4 “gesamtheitlich” falscher Begriff -> sehr detailliert	XIII
S. 5 “repräsentativ” -> keine Stichprobe / Fragenkatalog	XIII
3.3 Storyline	XIII
S. 14 Präfixe (Identifizier nicht mit Zahl)	XIII
Keine Vorwärtsreferenz!	XIII
Rechengenauigkeit: unterschied zwischen binärem/dezimalen Fließ-/Festkomma => Geld -> Festkommasemantik	XIII
Programmablauf ist gar nicht so unterschiedlich, sieht aber so aus. Nicht mit unterschieden einsteigen sondern mit Gemeinsamkeiten!	XIII
S. 18 Gliedern oder komprimieren: goto/labels Randnotizen in Java, Continue Unterschiede COBOL/Java, goto in COBOL kann unterschiedlich verwen- det werden. Sprünge auf EXIT Paragraphen, da kein return.	XIII

A. Zeitplan

