Programmiersprache

Java

2021 / Teil 1

Detlef Wilkening
www.wilkening-online.de

Programmiersprache Java

1 Allgemeines

Thema

- Objektorientiertes Programmieren in Java
- Achtung – die Sprache ist so umfangreich, dass die Vorlesung aus Zeitmangel bei weitem nicht alle Sprachelemente abdecken kann, und bei den abgedeckten auch viele Details auslässt.
- Achtung – die Bibliotheken von Java sind extrem umfangreich – auch hier wird die Vorlesung nur einzelne kleine Teile vorstellen können.

Voraussetzungen

- Kenntnisse einer beliebigen Programmiersprache.
- Die Vorlesung setzt voraus, dass Sie die Grundlagen der Programmierung kennen. Sie sollten z.B. wissen was eine Variable, ein Typ oder eine Funktion ist, und warum und wofür man sie benutzt. Sie sollten die Begriffe Scope (Block), Lebensdauer, Speicher, Stack und Heap zumindest ungefähr zuordnen können. Über den Vorteil einer konsistenten und durchgängigen vorstellen Formatierung und Benamung sollte ich kein Wort mehr verlieren müssen. Ihnen sollte klar sein, warum die Auftrennung in möglichst unabhängige Module sehr sinnvoll ist. Und zu guter Letzt sollten Sie zumindest in Ansätzen in der Lage sein, ein Problem in Teil-Probleme zu zerlegen, es zu strukturieren, und einfache Algorithmische

Lösungen zu verstehen und erarbeiten zu können.

Praktikum
- Kleine und große Aufgaben in Java lösen

Tools
- Sie benötigen für die Vorlesungen folgende Tools auf Ihrem Rechner
  - Java Development Toolkit in mindestens Version 8
  - Java IDE oder Programmier-Editor

Ich empfehle Ihnen:
- Java Development Toolkit in der neusten Version für Java 15 (JDK 15.0.2)
  - Java SE Development Kit 15 & Dokumentation
  - https://www.oracle.com/java/technologies/javase-downloads.html
- Integrierte Entwicklungsumgebung Eclipse IDE 2021-03 R
  - Eclipse IDE for Java Developers
  - https://www.eclipse.org/downloads/packages/release/2021-03/r/eclipse-ide-java-developers
- Natürlich funktioniert die Java Entwicklung auch mit anderen JDKs

Warum sollte ich Java lernen?
- Im Augenblick die wohl wichtigste und verbreiteste Programmiersprache.
- Relativ einfach zu lernen.
- Im großen Maße plattform-unabhängig und weit verbreitet.
- Für viele Anwendungsfälle fertige Bibliotheken enthalten.
- Gute Unterstützung an Tools.
- Gute Unterstützung an Bibliotheken, Literatur, usw.

Literatur und WWW

Es gibt hunderte von Büchern zu Java, Zeitschriften, und zig-Millionen Artikel im Web. Die erste Anlaufstelle sollte die offizielle Doku von Oracle sein:
- https://www.oracle.com/java/technologies/
- http://www.oracle.com/us/technologies/java/index.html

2 Java

Dieses Kapitel führt in die Philosophie und Historie von Java ein, und stellt wichtige Schlagwörter und (Marketing-) Begriffe der Java Welt vor. Betrachtet werden z.B. die Implikationen und Vor- und Nachteile einer virtuellen Maschine, wie Java sie einsetzt oder die Historie der JDKs. Aber es wird eben auch erklärt, was eine virtuelle Maschine ist, was JDKs sind, oder was Applets oder Midlets sind.

Alle, die dieses Hintergrund-Wissen nicht interessiert und die heiß auf die Sprache Java an sich und praktisches Programmieren sind, können dieses Kapitel überspringen.

2.1 Einleitung

Java wurde von James Gosling bei Sun entwickelt. Die erste offizielle Vorstellung von Java war im März 1995. Java entwickelte sich in den kommenden Jahren – aus verschiedenen Gründen – zu einer der wichtigsten Programmier-Sprachen auf dem Markt, und ist seit einigen Jahren eine der am häufigsten eingesetztesten Programmier-Sprache der Welt. Mittlerweile hat Oracle Sun übernommen – und damit liegt die Weiterentwicklung von Java primär in den Händen von Oracle.

Bevor wir aber ab Kapitel 3 tiefer in die Sprache Java einsteigen, wollen wir einige andere Dinge klären, die Java von vielen anderen Sprachen unterscheidet, und zum Teil auch für den Erfolg von Java verantwortlich sind.

So ist Java mehr als nur die objektorientierte Programmier-Sprache, als die Sie den Namen Java möglicherweise kennengelernt haben. Mit dem Begriff Java verbindet man auch häufig die sogenannte virtuelle Maschine „JVM“. Und manchmal wird Java sogar als Plattform – quasi als eine Art Betriebssystem – bezeichnet. Alle drei Sichtweisen haben ihre Berechtigung – und wir werden auch gleich sehen, warum.

2.2 Java als Plattform

Um die Aussage „Java als Plattform“ zu verstehen, wollen wir uns etwas detaillierter anschauen, wie *„normale“* Compiler-Sprachen arbeiten (Kapitel 2.2.1), und dies dann mit Java vergleichen (Kapitel 2.2.2).

2.2.1 Compiler-Sprachen

2.2.1.1 Executable

Bei einer Compiler-Sprache wie z.B. C oder C++ schreibt der Programmierer sein Programm in der entsprechenden Programmiersprache (eben z.B. C oder C++), und ein anderes Programm (der Compiler) übersetzt dies in direkt ausführbaren Maschinen-Code.

Damit ist klar, dass das eigentlich Executable (d.h. die ausführbare Datei auf Ihrer Festplatte) fest an einen Prozessor (bzw. kompatible Prozessoren) gebunden ist, denn nur diese Prozessoren verstehen diesen Maschinen-Code. Z.B. ein Executable, compiliert für einen Sparc-Prozessor, läuft nicht auf einem Power-PC Prozessor – auch wenn das gleiche Betriebssystem vorhanden ist.

Weiterhin hat jedes Betriebssystem ein spezielles Format, in dem ausführbare Datei aufgebaut sein müssen. Der Compiler muss beim Executable erzeugen natürlich dieses Format berücksichtigen. Damit ist ein Executable auch ein konkretes Betriebssystem (oder ein kompatibles) gebunden. Ein Executable, z.B. compiliert für Windows, läuft nicht unter Linux –

auch wenn der gleiche Prozessor vorhanden ist.

Unser Executable ist also an eine Plattform bestehend aus Prozessor und Betriebssystem gebunden.

Abb. 2-1 : Executable-Abhängigkeit zu einer konkreten Plattform

Natürlich kann man sein Programm auch für eine andere Plattform übersetzen, wenn man denn einen Compiler für diese Plattform hat, und das Programm plattform-unabhängig geschrieben ist (siehe Kapitel 2.2.1.2). Aber für jede gewünschte Ziel-Plattform (d.h. jede gewünschte Kombination aus Prozessor und Betriebssystem) muss ein eigenes Executable erstellt werden.

Abb. 2-2 : Executables für mehrere Plattformen

2.2.1.2 Programmierung

Aber nicht nur der Compiler bestimmt die Ziel-Plattform unserer Programm-Entwicklung. Auch die Programmierung selber kann hier bestimmte Randbedingungen festlegen. Ein Programm existiert ja nicht im luftleeren Raum, sondern soll später mit dem Computer und dem Benutzer interagieren. Dazu greift es z.B. auf das Datei-System zu, und macht Ein- und Ausgaben.

Hier hat der Programmierer drei Möglichkeiten:

1. Er beschränkt sich auf die Sprach- und Bibliotheks-Elemente, die im Umfang seiner Programmier-Sprache enthalten sind. Unter C und C++ ist dies nur ein extrem kleiner Bereich, der von den ISO Standards abgedeckt ist – z.B. gehören grafische Oberflächen oder Netzwerk-Bereiche nicht dazu.
   - Der Programmierer ist sehr eingeschränkt in seinen Möglichkeiten.
   - Dafür hat er ein Programm geschrieben, das prinzipiell für viele Plattformen compiliert werden kann.
2. Der Programmierer greift direkt auf das Betriebssystem zu. Dann hat er alle Möglichkeiten des Betriebssystems zur Verfügung – er kann also z.B. grafische Oberflächen implementieren, oder Netzwerk-Funktionalitäten nutzen.
   - Der Programmierer hat alle Möglichkeiten des Betriebssystems.
   - Das Programm ist an dieses eine Betriebssystem gebunden, also nicht portabel.
3. Letztlich kann der Programmierer auch fremde (3-party) Bibliotheken nutzen (kommerzielle oder freie), die betriebssystem-abhängige Funktionalität kapselt, und diese unter mehreren Betriebssystemen zur Verfügung stellt.
   - Der Programmierer muss entsprechende Bibliotheken für seinen Compiler finden, und er muss sie einsetzen können und dürfen.
   - Das Programm funktioniert dann für alle Plattformen, für die die Bibliothek vorhanden ist.

Abb. 2-3 : Executable für mehrere Plattformen entwickeln

2.2.1.3 Portabilität

Portabilität ist bei Compiler-Sprachen also ein komplexes Thema, da dort die Hardware, der Prozessor, das Betriebssystem und alle genutzten 3-party Bibliotheken eine Rolle spielen.

2.2.2 Java Virtual Machine – JVM

Während Compiler-Sprachen also ein Executable erzeugen, dass an eine ganz konkrete Plattform (Hardware, Prozessor, Betriebssystem) gebunden ist, geht Java einen anderen Weg.

Java compiliert die Programme für eine sogenannte virtuelle Maschine – die „Java Virtual Machine“ (JVM), d.h. eine Plattform, die es in Silizium gegossen gar nicht gibt. Stattdessen wird die virtuelle Maschine in Software geschrieben, und simuliert die Java Plattform auf einer beliebigen anderen Plattform, z.B. Windows, Linux oder Mac OS-X.

Abb. 2-4 : Idee einer virtuellen Maschine

Damit macht sich Java in einem hohen Maße von den Portabilitäts-Problemen von Compiler-Sprachen unabhängig. Es gibt nur noch eine Plattform – die Java Virtual Machine – egal auf welcher *echten* Plattform das Programm hinterher laufen soll. Solange auf der *echten* Plattform eine JVM existiert, können Java Programme dort ablaufen.

Daher erzeugt ein Java Compiler auch keinen Maschinen-Code und kein Executable, sondern sogenannten Byte-Code. Und die Datei bzw. die Dateien, die dabei erzeugt werden, sind natürlich auch keine Executables, sondern Dateien, die eine JVM zur Ausführung benötigten.

Dieser Ansatz einer virtuellen Maschine hat natürlich einige Konsequenzen – im Positiven wie im negativen Sinne, die wir im Folgenden diskutieren wollen.

2.2.2.1 Portabilität

Fangen wir mit einem klaren Plus an – die Portabilität. Java Byte-Code ist natürlich extrem portabel. Er enthält keinerlei Abhängigkeiten zu irgendeiner Hardware, zu irgendeinem Prozessor oder irgendeinem Betriebssystem. Auch 3-party Bibliotheken machen hier keine Probleme, da sie selber auch wieder als Java Byte-Code vorliegen. Solange auf der eigentlichen Plattform eine JVM existiert, kann der Java Byte-Code dort ausgeführt werden.

Und dazu muss der Byte-Code auch nicht neu compiliert werden, denn es gibt keine bzgl. der Byte-Code Definition keinerlei Unterschiede zwischen den JVMs verschiedener echter Plattformen.

Das Ganze geht sogar soweit, dass man Byte-Code von verschiedenen Quellen problemlos mischen kann, und problemlos woanders laufen lassen kann. Z.B. könnte ein Java Projekt zum Teil unter Windows mit Eclipse entwickelt werden, ein anderer Teil unter Solaris mit dem Java-Compiler von Sun, und ein dritter Teil unter Linux mit einem beliebigen anderen Compiler. Der

Byte-Code aller drei Teil-Projekte kann einfach zusammengebracht werden, und läuft dann z.B. auch auf Mac OS-X ohne jede Änderung. Denn der Byte-Code und die Schnittstelle der JVM sind genau normiert.

Abb. 2-5 : Portabilität mit einer VM und genormtem Byte-Code

Das ist Portabilität in einer ganz anderen Dimension im Vergleich zu den herkömmlichen Compiler-Sprachen.

2.2.2.2 Sandbox

Zusätzlich zu der Portabilität ermöglicht der Ansatz einer VM auch noch andere Optionen. Da der Byte-Code nicht direkt auf der Hardware und dem Prozessor läuft, und er auch nicht direkt auf das Betriebssystem zugreifen kann, lässt sich ein Java Programm prinzipiell vollständig kontrollieren. Alle Aktionen des Programms müssen letztlich durch die VM ausgeführt werden. Wenn die VM aber sicherheits-kritische Aktionen wie z.B. den Zugriff auf das Dateisystem nicht zulässt, kann ein Java-Programm hier nichts kaputt machen.

Eine virtuelle Maschine kann also für ein Java-Programm wie ein Sandkasten („Sandbox“) sein, in dem es sich austoben, aber auch keinen Blödsinn anrichten kann. Genau dies beherrscht die JVM auch. Im Kontext eines Browsers als Applet ausgeführte Java Programme haben eingeschränkte Rechte. Nur über Zertifikate und explizite Nutzer-Bestätigungen können Sie an mehr Rechte kommen – ansonsten müssen sie in ihrem Sandkasten bleiben.

Hinweis – die Idee der Sandbox funktioniert natürlich nur, wenn die JVM Implementierung keine Fehler aufweist, durch die die Programme den Sandkasten verlassen können.

2.2.2.3 Browser-Anwendungen

Ideal ist diese Portabilität und das Sandbox-Prinzip natürlich z.B. für Programme aus dem Internet, die in jedem Browser laufen können sollen. Denken Sie sich eine Aufgabe, für die einfache Text-Darstellung im Browser nicht reicht – hier wäre es schön, im Browser des Nutzers ein kleines Programm ablaufen lassen zu können. Dies ist mit Java möglich, da es für alle halbwegs wichtigen Browser Java-Plugins gibt, d.h. JVMs für den Browser.

Ein Server kann also ein Java-Programm als Byte-Code an einen Browser ausliefern, ohne sich Gedanken um die Ziel-Plattform machen zu müssen (Portabilität), und Sie als Nutzer können so ein Programm bedenkenlos in Ihrem Browser ausführen, da es sich nur im Sandkasten der JVM mit eingeschränkten Rechten *bewegen* darf.

So ein Java-Programm, das im Browser läuft, ist übrigens kein ganz normales Java-Programm, sondern ein sogenanntes **Applet**, das sich minimal von normalen Java Desktop Programmen unterscheidet.

Hinweis – während in den Anfangszeiten von Java gerade die Applets als das Besondere von Java herausgestellt wurden und ein Grund für den Erfolg von Java waren – sind Applets heute nur noch sehr selten anzutreffen. Seit Java 11 werden sie auch nicht mehr unterstützt (seit Java 9 waren sie „veraltet“.
Zum einen gibt es mittlerweile viele andere Technologien, die Java im Browser überflüssig machen, zum anderen hat sich die JVM im Browser als ein echtes Sicherheitsproblem herauskristallisiert – da die Sandbox nie wirklich sicher war. Die JVM hatte und hat immer mit schweren Lücken im Sandbox-Prinzip zu kämpfen, wodurch Applets den Sandkasten verlassen und dann mit den Rechten des Browsers auf dem System agieren konnten.

2.2.2.4 Portable Umgebungen

Nicht nur Desktops bieten eine große Auswahl an Plattformen, auch portable Geräte wie z.B. PDAs oder Handys sind eine interessante Umgebung für portable Programme. Sie bieten vielfältige Hardware und unterschiedliche Betriebssystem – es wäre aber interessant Programme für alle diese Plattformen schreiben zu können.

Dafür wurde in der Java Familie „JME“ („Java Mobile Edition“) entwickelt. Dies ist eine eingeschränkte JVM, die nicht alle Sprach-Features von Java anbietet, und auch bei weitem nicht alle Bibliotheken von Java unterstützt. Aber JME ist wieder eine genau definierte Umgebung mit klar abgesteckten Fähigkeiten, die an die Anforderungen von einfachen portablen Geräten angepasst sind. Java-Programm für die JME-VM nennen sich **Midlets**.

Achtung – Java Mobile ist schon einige Jahre alt und adressierte portable Umgebungen, die nicht besonders leistungsfähig waren – z.B. Handys aus einer Vor-Smartphone-Area. Heutige portable Geräte wie Smartphones oder Tablets haben die Leistungsfähigkeit von normalen Desktop Rechnern – hier kann eine fast ganz normale JVM problemlos laufen.

2.2.2.5 Android

Viele heutige portable Geräte laufen unter dem Betriebssystem Android – und die Haus- und Hofsprache von Android ist Java. Wenn Sie eine App für Android entwickeln wollen, dann werden Sie dies in den meisten Fällen in Java machen. Natürlich sind auch andere Sprachen möglich – allen voran C++ – aber die normale Sprache für Android-Apps ist Java.

Das Java von Android ist nicht abgespeckt wie JME, sondern ein ganz normales Java. Nur für die grafische Oberfläche wird nicht AWT, Swing oder JavaFX eingesetzt, sondern ein spezielles Android-Spezifisches GUI Framework.

Leider sprengt die Entwicklung von Android Apps unseren Rahmen – obwohl es natürlich „cool“ wäre, ein Programm zu schreiben, das auf vielen Smartphones läuft.

2.2.2.6 Server-Umgebungen

Was für Browser und PDAs recht ist, kann für Server-Infrastrukturen auch passend sein. Auch

dort findet man häufig – historisch bedingt – eine heterogene und gewachsene Infrastruktur von Plattformen der verschiedensten Art. Früher konnten hierbei Programme nicht einfach von einem System auf das nächste verschoben werden, um den wechselnden Anforderungen zu genügen – mit Java ist dies möglich. Solange ein JVM zur Verfügung steht, kann ein Programm problemlos von einem Rechner auf den anderen verschoben oder kopiert werden.

In Server-Umgebungen ist dies nicht das einzig interessante Kriterium. Wichtig ist z.B. auch die Unterstützung von Persistenz, Datenbanken, Transaktionen, u.v.m. Hierfür wurde in die Java Familie „Java EE“ („Java Enterprise Edition“) aufgenommen. Dies ist eine Erweiterung von Java um spezielle Bibliotheken und Fähigkeiten (z.B. „Java Enterprise Beans“ als ein Komponenten-Konzept für Server-Anwendungen) für eben solche Themen.

Auch andere Server-Themen fanden ihren Niederschlag in der Java Familie. So gibt es in Java z.B. **Servlets** – d.h. Java-Programme, die in einem Web-Server ausgeführt werden und dynamische HTML Seiten bereitstellen können. Oder auch **Portlets**, die speziell für die Programmierung von Web-Portalen entwickelt wurden.

Im Tutorial wird gar nicht auf spezielle Server-Elemente von Java eingegangen – dazu fehlt wieder mal die Zeit.

2.2.3 Plattform „Java“

Wie man sieht, stellt Java in Form der JVM wirklich eine Plattform dar – denn sie simuliert einen Java Prozessor mit einem Java Betriebssystem – dem typischen Verständnis einer Plattform.

Aber auch mit einem breiter gefassten Verständnis von Plattform trifft der Begriff auf Java zu. Java adressiert neben den normalen Desktop-Programmen auch Applets, die im Browser laufen. Dann mit JME Midlets für einfache portable Geräte, und mit speziellen GUI Android Smartphones. Und für Server-Anwendungen gibt es Java EE mit u.a. Servlets und Portlets. Dies ist ein breites Spektrum an Einsatzgebieten, was in dieser Vielfalt von kaum einer anderen Programmier-Sprache abgedeckt wird.

2.3 Technologien

Die JVM enthält einige sehr interessante Technologien, die heutzutage die Leistungsfähigkeit von Java mitbestimmen und daher zum Erfolg von Java beigetragen haben. Zwei dieser Technologien möchte ich hier detaillierter vorstellen, da sie im bisherigen Kapitel schon mehrfach erwähnt wurden, und sehr aufschlussreich sind.

- Speicherverwaltung und Garbage-Collector
- Performance und Hotspot

2.3.1 Speicherverwaltung und Garbage-Collector

Die Sprache Jave und die JVM unterstützen eine Technik, die Garbage-Collection genannt

wird. Dies meint einfach nur, dass Sie beliebig Objekte erzeugen können, und sich nicht um die Freigabe des von den Objekten belegten Speichers kümmern müssen. Dies macht dann der Garbage-Collector – ein Teil der JVM. Er erkennt nicht mehr benötigte Objekte und gibt deren Speicher wieder frei. Dies entlastet Sie als Programmierer, da Sie sich um dieses Thema nicht mehr kümmern müssen – und es verhindert den Quell von Fehlern der mit manueller Speicherverwaltung verbunden ist. Falls Sie jemals z.B. „C“ programmiert haben, dann wissen Sie sicher, dass Zeiger und manuelle Speicherverwaltung eine Büchse der Pandora sind, deren Problemen man kaum Herr wird. All das gibt es in Java nicht – das macht die Sprache einfacher und die Programme fehlerfreier.

Eigentlich könnte man hier aufhören, denn damit ist das Thema „Speicherverwaltung und Garbage-Collection“ oberflächlich ausreichend beschrieben. Und uns fehlt die Zeit detailliert in die heutigen Techniken und Algorithmen von Garbage-Collectoren einzusteigen, obwohl es ein ungeheuer interessantes und faszinierendes Thema ist. Aber ein paar Dinge möchte ich Ihnen doch noch mit auf den Weg geben – heutige Garbage-Collectoren wie z.B. im JDK 1.8 arbeiten ungeheuer effizient, d.h.:

- Sie erkennen zuverlässig nicht mehr referenzierte Objekte.
- Sie arbeiten sehr schnell. Programme mit Garbage-Collections können den gleichen Programmen mit manueller Speicherverwaltung in der Performance überlegen sein.
- Seit dem JDK 1.5 kann ein Großteil der Garbage-Collector Techniken auch parallel zum normalen Programm-Fluss stattfinden – daher die JVM nutzt z.B. neuere Multi-Core Maschinen sehr gut aus.

Einziger Nachteil heutiger Garbage-Collector Techniken ist, dass sie einen gewissen Speicheroverhead haben, d.h. mehr Speicher verbrauen als im Idealfall notwendig wäre. Bei reinen Mark&Copy Garbage-Collectoren kann dieser Overhead fast die Hälfte des Speichers ausmachen – aber auch hier sind heutige JVM’s viel effizienter geworden.

2.3.2 Performance und HotSpot

Ein immer wieder heißes Thema in der Diskussion um Java ist die Performance. Es ist ja auch *scheinbar* sehr einleuchtend, dass Byte-Code auf einem in Software geschriebenen Prozessor nicht so schnell ausgeführt werden kann wie Maschinen-Code, der direkt auf einem echten Prozessor ausgeführt wird. Aber das ist nur der erste Eindruck – mittlerweile gibt es über 40 Jahre Erfahrung und Ideen im Bau von virtuellen Maschinen, und viele dieser Ideen betreffen der Performance. Die beiden interessantesten Technologien sind hier JIT-Compiling und HotSpot.

JIT steht für „Just-in-Time“, d.h. ein JIT-Compiler ist ein Just-in-Time Compiler. Damit ist gemeint, dass die virtuelle Maschine statt einfach den Byte-Code auszuführen diesen bei der ersten Ausführung in Maschinen-Code übersetzt und dann dieser direkt vom Prozessor ausgeführt wird. Da dieser Übersetzungs-Vorgang Zeit dauert, wird die Ausführungs-Dauer beim Übersetzen natürlich stark gebremst – danach steht aber die native Prozessor-Performance zur Verfügung.

Prinzipiell ließe sich natürlich der gesamte Java Byte-Code in Maschinen-Code übersetzen, aber es gibt Bereiche, wo es sich nicht lohnt, der Aufwand sehr hoch ist, oder aus anderen Gründen nicht wirklich gut praktikabel ist. Außerdem darf auch compilierter Byte-Code nicht die Sicherheits-Prinzipien von Java wie z.B. die Sandbox verletzen.

Auf der anderen Seite gibt es aber auch Bereiche wo sich mit Informationen aus dem Kontext des Byte-Codes sehr radikale Optimierungen machen ließen – z.B. wer ruft eine Funktion auf, bzw. mit welchen Parametern wird sie immer aufgerufen. Und hier kommt HotSpot zum Tragen.

HotSpot ist der Name der JVM seit dem JDK 1.3. Bestandteil von HotSpot ist u.a. ein JIT-Compiler. Das Besondere an HotSpot ist aber, dass hier die VM das Programm zur Laufzeit beobachtet, und in Abhängigkeit von diesen Beobachtungen weitere Optimierungen durchführt. Ein offensichtlicher Hintergrund dieser Vorgehensweise ist die Erkennung sogenannter Hot-Spots, daher von Programm-Teilen, die besonders häufig ausgeführt werden und bei denen sich die Optimierung besonders lohnt. Aber HotSpot kann z.B. auch dynamische Optimierungen durchführen, die die Semantik des Codes verändern, aber im konkret vorliegenden Kontext funktionieren.

Hinter HotSpot steht mittlerweile eine Menge Erfahrung und Know-How, und die Optimierungen werden mit jedem JDK verbessert. Mit JIT-Compiling und den weiteren HotSpot Optimierungen spielt Java heute von der Performance her auf dem Niveau von normalen compilierten Sprachen wie z.B. C++ mit. Es gibt sogar ernst zu nehmende Stimmen, die die Meinung vertreten, dass virtuelle Maschinen viel besser optimieren können als jeder Compiler, da ihnen viel mehr Informationen zur Verfügung stehen kann.

2.3.2.1 Bemerkung

Eigentlich ist hiermit alles Wichtige gesagt. Natürlich ließe sich noch manches zu den Techniken von VMs, JIT-Compilern und vor allem HotSpot schreiben – aber eine solche Vertiefung würde den Rahmen des Tutorials bei weitem sprengen. Ich möchte aber noch ein bisschen zum Thema Performance von Java sagen, da es scheinbar ein extrem emotional besetztes Thema mit vielen Gerüchten und Halb-Wahrheiten ist.

Viele Kritiker von Java haben bis heute nicht erkannt bzw. erkennen wollen, dass sich die Performance von Java seit dem JDK 1.0 immens verbessert hat, und viele ihrer Kritiken heute nicht mehr zutreffend sind. Auf der anderen Seite gibt es auch eine Menge Java-Fanatiker, die unreflektiert Java für die performanteste Sprache der Welt halten. Und wenn diese aufeinander treffen, dann knallt es – manchmal kann das sehr unterhaltsam sein.

Ich kann Ihnen nur empfehlen, wenn Sie mal Langeweile haben – suchen Sie sich z.B. eine C++ Newsgroup und posten Sie etwas polemisch hinein, dass Java viel schneller ist als C++. Oder posten Sie umgekehrt in eine Java Newsgroup dass C++ viel schneller als Java ist. Dann lehnen Sie sich zurück und genießen den Sprachen-Krieg, den Sie angezettelt haben. An langen verregneten Abenden kann dies viel unterhaltsamer als jeder Film sein.

Und wo liegt die Wahrheit? Nirgendwo oder überall. Ich kann Ihnen problemlos ein Programm schreiben, wo C++ Java schlägt, und auch umgekehrt. Aber was sagt das schon aus? Nichts! Es kann schon unglaublich komplex und schwierig sein, *„objektiv“* und reproduzierbar die unterschiedliche Performance zweier unterschiedlicher Implementierungen der gleichen Sprache in der gleichen Umgebung zu beurteilen. Viel schwieriger ist es schon, verschiedene Bibliotheken zu vergleichen. Wie schwierig ist es dann wohl, akzeptable Vergleiche zwischen zwei Sprachen zu bekommen? Und ist das überhaupt sinnvoll? Man entscheidet sich doch nicht für oder gegen eine Sprache nur aufgrund der Performance, sondern auch aufgrund von hunderten anderer Kriterien.

Ich kann Ihnen nur empfehlen: solange Sie nicht wirklich richtig Ahnung von der Materie haben – halten Sie sich aus der Performance Diskussion raus. Java ist heute, gerade auf heutigen Rechnern, verdammt schnell – aber das gilt auch für viele andere Sprachen. Und wenn Sie wirklich mal ein Performance-Problem haben, suchen Sie nach besseren Algorithmen oder anderen Implementierungen bevor Sie die Sprache wechseln. In fast allen Fällen reicht das aus. Und wenn das nicht reicht, dann haben Sie wahrscheinlich ein Problem, das ein einfacher Sprachenwechsel auch nicht so einfach beseitigt.

2.4 Fazit

Wie Sie sehen, ist gerade die Plattform Java in Form der JVM ein faszinierender Ansatz, der viele Probleme z.B. bzgl. Portabilität oder Speicherverwaltung radikal beseitigt. Hinzu kommt eine relativ einfache Sprache, mit der sich das restliche Tutorial beschäftigt, und eine wunderbare riesige Bibliothek, die viele Probleme löst. Später werden wir dann noch einige sehr mächtige Tools kennen lernen, die uns beim Programmieren stark unterstützen und auch noch frei sind. Programmierer-Herz – was willst du mehr?

Ach ja, es sollte Ihnen klar sein, dass dieses Kapitel bei weitem nicht alle Java Wörter vorgestellt hat. Wundern Sie sich also nicht, wenn Sie mit welchen konfrontiert werden, die hier fehlen. Das Java Universum ist groß und wächst jeden Tag – es gibt viel zu entdecken...

3 Mini-Einführung

In Java können als ausführbare Einheiten Programme, Applets, Midlets und Servlets erstellt werden. Wir werden hier hauptsächlich Programme erstellen. Der Unterschied zwischen Applets und Anwendungen ist nicht sehr groß, so dass dies im Augenblick keine praktische Bedeutung hat.

Dieses Kapitel stellt einige Grundlagen vor, mit denen Sie ruck-zuck konfrontiert werden, um überhaupt kleine sinnvolle Programme schreiben zu können, und um die Praktikums-Aufgaben lösen zu können. Von daher brauchen wir sie, und darum werden sie hier gleich erklärt. Auf der anderen Seite sind viele dieser Dinge „fortgeschrittenen Themen“, die erst im Laufe der Zeit kommen. Nehmen Sie sie darum hier einfach hin, und wenden Sie sie pragmatisch an, ohne zuviel darüber nachzudenken. Im Laufe der Zeit klärt sich alles auf.

3.1 Applikation

3.1.1 Beispiel 1

In einer Java Applikation muss es mindestens eine public-Klasse geben, die folgende Klassen-Funktion **main** enthält:

```
public class Kap_03_01_Bsp_01_HalloWelt {

   public static void main(String[] args) {
      System.out.println("Hallo Welt");
   }
}
```

Jede öffentliche Klassen-Funktion („public static“) mit dem Namen „main“, dem Rückgabetyp „void“, und der Parameterliste „String[ ]“ stellt einen Einsprungpunkt in ein Java Programm dar. Beim Aufruf der JVM geben Sie eine Klasse an, in der die JVM nach einer solchen Klassen-Funktion sucht – und wenn sie gefunden wird, beginnt dort das Programm.

Typischerweise gibt es in einem Java-Programm eine Klasse, die die Anwendung repräsentiert, eine solche „main“ Funktion hat, und den zentralen Einsprungpunkt in die Anwendung darstellt.

Achtung – der Quelltext der Klasse **MUSS** in einer Datei mit dem Namen der Klasse selber und der Extension „java“ stehen. Dies fordert die Sprache! Das gleiche Verfahren gibt es auch noch bei der Verwendung von Packages. Das obige Beispiel **muss** also in der Datei „Beispiel.java“ stehen.

3.1.2 Beispiel 2

Um zum einen schon einen Eindruck von der Einfachheit und Leistungsfähigkeit von Java zu bekommen, und zum anderen zu testen, ob ihre Java Installation problemlos funktioniert hat, hier noch ein zweites einfaches Programm.

```
import javax.swing.JFrame;

public class Kap_03_01_Bsp_02_Gui {

   public static void main(String[] args) {
      JFrame frame = new JFrame("Mein erstes GUI Fenster");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setLocation(200, 200);
      frame.setSize(300, 100);
      frame.setVisible(true);
   }

}
```

Im Gegensatz zum ersten arbeitet es nicht nur auf der Kommandozeile, sondern öffnet zusätzlich ein einfaches leeres GUI-Fenster mit dem Titel „Mein erstes GUI Fenster“ – siehe folgende Abbildung:

Abb. 3-1 : einfaches Beispiel GUI-Fenster mit Java Swing

Hinweis – erzeugte Objekte (hier „frame“) müssen nicht gelöscht werden. Dies macht die JVM automatisch. Der entsprechende Vorgang nennt sich Garbage-Collection. Je nachdem, von welcher Programmiersprache Sie kommen, ist dies normal oder ungewöhnlich für Sie. Sprachen wie z.B. C# oder Ruby, haben auch einen Garbage-Collector, und verhalten sich hier wie Java. Kommen Sie dagegen von Sprachen wie z.B. Pascal, C oder C++, so ist dieses Verhalten ungewöhnlich, da in diesen Sprachen der Programmierer Speicher explizit wieder freigeben muss (selbst wenn dies oft nicht direkt zu sehen ist). Java hat hier den Ansatz des Garbage-Collectors gewählt, der zu weniger Fehlern und mehr Komfort für den Programmierer führt. Übrigens: Sie müssen erzeugte Objekte in Java nicht nur „*nicht explizit*“ löschen, Sie können es natürlich auch gar nicht. Java hat kein Sprachmittel zu Speicherfreigabe – wozu auch?

3.2 Quelltext

3.2.1 Aufbau

Java Quelltext ist formlos bzw. Block-, Anweisungs- und Token-orientiert aufgebaut – d.h. **nicht** zeilen- oder einrückungs-orientiert. Sie können also beliebig Leerzeilen, Leerzeichen, Tabulatoren und Zeilenumbrüche in ihren Quelltext einfügen, solange sie keine Token auseinander reißen. Token sind quasi die kleinste syntaktische Einheit von Java, z.B. Symbole (Namen, Bezeichner,...), Schlüsselwörter (class, import, public,...), Operatoren (+, -, +=, &&, <<<) und Sonderzeichen ({, }, (,), [,...)

3.2.2 Literale: Zahlen, Zeichen und Texte

Literale sind Zahlen-, Zeichen- und Text-Konstanten im Quelltext.

- Integer Literale sind normale Zahlen.
- Gleitkomma-Literale werden an dem Dezimal-Punkt erkannt.
- Einzelne Zeichen-Konstanten einfache
- Zeichenketten-Konstanten (Texte) werden in doppelte Hochkommata eingeschlossen. Sie müssen spätestens am Zeilenende beendet werden.

```
2          // Integer Konstante
3.14       // Fliesskomma Konstante
.23        // Fliesskomma Konstante
2.         // Fliesskomma Konstante
'c'        // Zeichen-Konstante
"Hallo"    // Zeichenketten-Konstante
```

```
""        // Leere Zeichenketten-Konstante (Leer-String)
```

3.2.3 Kommentare

Java unterstützt drei Arten von Kommentaren:

1. Bereichs-Kommentare, die mit **/*** beginnen und mit ***/** enden, und dabei beliebige Bereiche überdecken dürfen – eine Schachtelung ist nicht erlaubt.
2. Zeilenend-Kommentare, die mit **//** beginnen und für den Rest der Zeile gelten.
3. Spezielle Java-Doc Kommentare, die mit **/**** beginnen, mit ***/** enden und spezielle Tags enthalten – eine Schachtelung ist auch hier nicht erlaubt. Mit dem Tool Java-Doc können aus diesen Kommentaren automatisch Referenz-Dokumente erzeugt werden. Daher werden diese Kommentare häufig auch „Java-Doc Kommentare“ genannt. Java-Doc Kommentare werden in der Vorlesung aus Zeitmangel nicht besprochen.

```
/*
Dies ist ein Kommentar
*/

double d /* Kommentar */ = 3.1;

int i = 5;     // Noch ein Kommentar
```

Hinweis – die Nutzung von Java-Doc Kommentaren ist sehr zu empfehlen, da auch aktuelle Entwickluns-Umgebungen wie z.B. Eclipse oder NetBeans diese direkt während der Entwicklung auswerten und z.B. in Views oder Tooltips anzeigen.

3.2.4 Symbole

Java unterscheidet bei Symbolen für z.B. Variablen, Funktionen oder Klassen zwischen Groß- und Kleinschreibung. „Fenster“, „fenster“ und „FENSTER“ sind drei unterschiedliche Symbole.

Erlaubte Symbole bestehen u.a. aus den Buchstaben ‚a’ - ‚z’, ‚A’ - ‚Z’, Ziffern und dem Underscore. Hierbei darf das Symbol nicht mit einer Ziffer beginnen. Außerdem sind viele weitere Unicode Zeichen erlaubt, die Buchstaben und Ziffern darstellen. Sie sollten sie in der Praxis (zumindest in unserem Sprachraum) aber nicht benutzen. Aber wenn Sie wollen, können Sie in Java also japanische Variablen- und chinesische Funktions-Namen vergeben – ich kann es aber nicht empfehlen.

3.2.5 Konventionen

In Java existiert die Konvention, dass

- Package-Namen klein beginnen und klein geschrieben werden,
- Klassen- und Interface-Namen groß beginnen und kapitalisiert geschrieben werden,
- Funktions- und Variablen-Namen klein beginnen und kapitalisiert geschrieben werden,
- Klassen-Konstanten GROSS mit Underscores geschrieben werden.

Beispiele:

Package-Namen	mypackage
	metadataauthoring
Klassen- bzw. Interface-Namen	MyClass
	MetaDataManager
	AbstractGuiModel
Funktions-Namen	getBackgroundColor
	calculateAverage
Variablen-Namen (auch lokale Konstanten)	pidCount
	bitLength
Konstanten-Namen (nur Klassen-Konstanten)	DEFAULT_COLOR
	START_WIDTH

Selbst wenn sie jetzt noch keine z.B. Packages oder Interfaces kennen, nehmen sie schon mal zur Kenntnis, daß es für alle Namen in Java Konventionen bzgl. der Schreibweise gibt, an die sie sich halten sollten.

Damit können wir nun schlussfolgern, dass in der Zeile:

```
System.out.println("Java");
```

System eine Klasse, out ein Attribut von System, und println eine Elementfunktion von out sein muss.

Achtung – in Java sind diese (und andere) Konventionen wichtig, da viele Mechanismen (z.B. bei Beans) und Tools darauf basieren. Sie sollten sich also konsequent daran halten. Dies betrifft nicht nur die Names-Konventionen, sondern auch andere, die wir noch kennen lernen werden. Viele moderne Entwicklungs-Umgebungen unterstützen Sie durch Warnungen und Hinweise bei der Einhaltung dieser Konventionen.

3.3 Ausgabe

In einer Applikation kann man Zeichenketten, Konstanten, Variablen, usw. auf die Konsole ausgeben. Dafür existiert in der Klasse „System“ der Ausgabestream (PrintStream) „out“, für den es u.a. die Element-Funktionen „print“ und “println“ gibt.

```
System.out.print(<ein Argument>);
System.out.println(<ein Argument>);     // Ist ein 'print' inkl. Zeilenumbruch
```

Beide Elementfunktionen erwarten einen Argument beliebigen Typs und geben diesen auf der Konsole aus. Die Elementfunktion „println“ erzeugt zusätzlich zur Ausgabe analog zu „print“ noch einen Zeilenumbruch.

```
int i = 7;
System.out.print("->");
System.out.println(i);
System.out.println("Java");
System.out.println(42);
```

```
Ausgabe
->7
Java
```

```
42
```

Den Funktionen können beliebige Argumente übergeben werden. Im folgenden Beispiel sieht man das z.B. an der Ausgabe von „System.out“ – die Ausgabe ist sicher nur bedingt sinnvoll und hilfreich und vor allem ist sie nicht immer gleich (die Zahl hinter dem @ kann anders sein) – aber sie zeigt beispielhaft, dass sich wirklich jedes Argument in einen String wandeln und dann ausgeben läßt.

```
System.out.print(7);
System.out.print('c');
System.out.println(78);
System.out.println("Hallo");
System.out.println(System.out);    // <- es laesst sich wirklich alles ausgeben
```

```
Mögliche Ausgabe (die genaue Ausgabe von System.out ist nicht definiert)
7c78
Hallo
java.io.PrintStream@1a7bf11
```

Es können auch mehrere Elemente gleichzeitig ausgegeben werden: wenn ein Element ein String ist, wird beim Operator + das andere Element immer automatisch in einen String umgewandelt und danach beide Strings konkateniert.

```
int i = 7;
System.out.println("Java " + 42 + " " + i);    // Ausgabe: Java 42 7
System.out.println(42 + " Java " + i);         // Ausgabe: 42 Java 7
```

```
Ausgabe
Java 42 7
42 Java 7
```

Bemerkung – die Auswertungsreihenfolge des Operators + ist in Java von links nach rechts definiert. Und auch die Addition zweier Zahlen entspricht der *normalen* Erwartung.

```
int i = 4;
System.out.println(i + 42);          // Ausgabe: 46
System.out.println(4 + 5 + 7);       // Ausgabe: 16
```

```
Ausgabe
46
16
```

Der Aufruf von „System.out.println“ ohne Parameter erzeugt einfach nur einen Zeilenumbruch, sprich eine Leerzeile.

```
System.out.println();    // Erzeugt eine Leerzeile
```

3.4 Strings

Neben vielen anderen Typen kennt Java auch einen Datentyp für Texte (Zeichenketten). Dies ist der Typ „String“. String ist kein elementarer Datentyp, kann aber trotzdem einfach benutzt werden.

Strings werden im Detail später besprochen. Wir führen sie hier ganz kurz und pragmatisch ein, da sie der Typ der Kommandozeilen-Argumente (siehe Kapitel 3.5) und der Rückgabe-Typ

beim Einlesen einer Zeile von der Kommandozeile (siehe Kapitel 3.10) sind.

Strings können mit Literalen initialisiert werden, und können einander zugewiesen werden. Ihre Länge bestimmt man mit der Element-Funktion „length“.

```
String s = "Java";
System.out.println("Laenge von \"" + s + "\" ist " + s.length());
```

```
Ausgabe
Laenge von "Java" ist 4
```

3.5 Arrays und Kommandozeilen-Argumente

In den Übungs-Aufgaben werden häufiger Kommandozeilen-Argumente benutzt. Sie werden der main-Funktion in einem String-Array übergeben. Um das Array auswerten zu können, benötigen Sie folgende Informationen:

- Die Größe des Arrays kann über das Attribut length abgefragt werden – Zugriff über den Punkt-Operator ‚.‘.
- Der Zugriff auf die Elemente des Arrays geschieht über den Index-Operator [] (die eckigen Klammern) mit Index in den Klammern.
- Arrays sind in Java null-basiert, d.h. z.B. das erste Element hat den Index ‚0‘

Jedes Element des String-Arrays für die Kommandozeilen-Argumente ist ein String (vergleiche Kapitel 3.4), und kann als solcher direkt genutzt werden, oder auch einer String Variablen zugewiesen werden.

```
public class Example {

   public static void main(String[] args) {
      if (args.length==0) {
         System.out.println("Kein Argument");
         return;
      }
      System.out.println(args.length + " Argument(e)");
      System.out.println("- 1. Arg: \"" + args[0] + "\"");
      if (args.length>1) {
         String arg2 = args[1];
         System.out.println("- 2. Arg: \"" + arg2 + "\"");
      }
   }

}
```

Hinweis – der Zugriff auf ein nicht-existentes Array-Element (d.h. Index kleiner Null oder Index zu groß) führt zu einer Exception.

3.6 Klassen

Klassen sind das elementare Strukturierungsmittel von Java.

Pro Datei **muss genau eine** öffentliche Klasse vorhanden sein, die der Datei ihren Namen gibt. Eine Klasse wird folgendermaßen definiert:

```
[Modifizierer] class <klassen-name> {
   [Elementfunktionen, Attribute,...]
}
```

Eine Klasse wird öffentlich, indem sie den Modifizierer „public“ bekommt. Eine minimale öffentliche Klasse sieht also so aus:

```
public class Klasse {
}
```

Die Datei, in der die Klasse definiert ist, **muss** genauso heißen wie die Klasse - abgesehen von der Dateiextension ‘.java’. Die öffentliche Klasse „Klasse“ **muss** also in einer Datei „Klasse.java“ stehen.

Hinweis – liegt eine Klasse in einem oder mehreren Packages, so muss die Datei in einer Verzeichnisstruktur liegen, die der Package Struktur der Klasse entsprecht.

3.7 Funktionen

Jeder ausführbare Code steht immer in einer Funktion.
Funktionen sind immer Bestandteil einer Klasse.

Syntax:

```
Modifizierer <Rückgabetyp> <Fkt-Name> ( <Parameterliste> ) {
   <Implementierung>
}
```

Modifizierer – z.B.: public, private, static, final, ...

Eine Parameterliste ist eine durch Komma getrennte Auflistung von Parametern (sie kann auch leer sein. Ein Parameter besteht aus Typ und Name – Bsp.:

```
public static void f() { ... }
protected final int doit(StringBuffer sb) { ... }
private static String calcName(String s, int i) { ... }
```

Aufruf:

```
<Fkt-Name>( <Argumentliste> );
```

Bsp:

```
f();
calcName("", 1);
```

Rückgaben können beim Funktions-Aufruf ignoriert werden – siehe im Beispiel der Aufruf der Funktion „calcName“, die einen String zurückgibt – der beim Aufruf aber ignoriert wird.

Achtung

- Die meisten Funktionen in Java sind sogenannte Elementfunktionen. Sie können **nur** mit

Objektbezug aufgerufen werden.
- Nur Funktionen mit dem Modifier „static“ (sogenannte Klassenfunktionen) können direkt benutzt werden.
- Solange wir noch nicht tiefer in Klassen eingestiegen sind, werden alle unsere selbstgeschriebenen Funktionen Klassen-Funktionen sein – daher den Modifier „static“ enthalten. Bitte denken Sie daran – vergessen Sie das „static“ wird ihr Programm in den meisten Fällen nicht compilieren.

3.8 Packages

Es ist sinnvoll, ihre Programme in Module zu ordnen. Das Sprachmittel hierfür sind Packages. Um eine Klasse in ein Package zu legen, muss die Datei in einem entsprechenden Verzeichnis (mit Namen des Package) liegen, und die Datei muss als erste Anweisung (d.h. nicht Kommentar oder Leerzeilen) eine package-Anweisung enthalten. Eine package-Anweisung beginnt mit dem Schlüsselwort „package“, dann folgt der Package-Name, und das ganze muss mit einem Semikolon abgeschlossen sein.

```
package mypackage;
```

Bei verschachtelten Packages müssen auch die Verzeichnisse entsprechend verschachtelt sein. In der Package-Anweisung werden die Package-Namen dann durch Punkte getrennt.

```
package mypackage.nocheins.innerespackage;
```

Analog zu Klassen, deren Namen zu Datei-Namen korrespondieren müssen, **müssen** also auch die Package-Namen zu Verzeichnissen korrespondieren. Dies verlangt die Sprache, und sie müssen sich daran halten.

Hier das „Hallo-Welt“ Beispiel aus Kapitel 3.1.1 in leicht modifizierten Versionen:
- Einmal in einem einfachen Package „pack“.
- Und dann in einem verschachtelten Package „aussen“ in „mitte“ in „innen“.

Beispiel 1 – Achtung, die Datei „Beispiel.java“ muss in einem Verzeichnis „pack“ liegen.

```
package pack;

public class Beispiel {

   public static void main(String[] args) {
      System.out.println("Hallo Welt");
   }

}
```

Beispiel 2 – Achtung, die Datei „Beispiel.java“ muss in einem Verzeichnis „innen“ liegen, das in einem Verzeichnis „mitte“ liegen muss, und das wiederum in einem Verzeichnis „aussen“ liegen muss.

```
package aussen.mitte.innen;

public class Beispiel {
```

```
   public static void main(String[] args) {
      System.out.println("Hallo Welt");
   }

}
```

3.9 Exceptions

Es gibt immer Dinge, die können schief gehen – z.B. eine Eingabe oder ein Array-Zugriff. Solche Probleme werden in Java immer durch Exceptions gemeldet. Da Sie am Anfang sicher viele Fehler machen werden (jeder Fehler ist gut, denn er zeigt, dass Sie Java angewendet haben), werden Sie in Java von Anfang an häufig mit Exceptions konfronitert werden. Im Augenblick reicht uns – neben dem Thema „Checked-Exceptions“ – siehe gleich – hier das Wissen, dass mit Exceptions Fehler in unserem Programm angezeigt werden. Irgendetwas haben Sie falsch gemacht, oder ist einfach schief gelaufen. Am Anfang gehen wir erstmal davon aus, dass alles gut geht – außer wir wollen den Fehlerfall explizit nutzen, wie z.B. in Kapitel 3.11. Daher könnten wir Exceptions eigentlich erstmal ignorieren, aber...

Aber ein Teil der Exceptions können in Java nicht ignoriert werden, nämlich alle Checked-Exceptions. Daher: immer wenn eine Funktion ein *„Problem“* mit einer solchen Checked-Exception melden *könnte*, dann müssen sie sich darum kümmern. Ein Beispiel dafür ist die Funktion „read“ in der Anweisung „System.in.read()“ im nächsten Kapitel.

Schreiben Sie im Augenblick die problematische(n) Anweisung(en) einfach in einen try-Block und fügen noch einen leeren catch-Block an. Außerdem merken sie sich, dass der Programmfluss im Fehlerfall in den catch-Block verzweigt. Beispiele hierfür finden wir gleich in den Kapiteln 3.10 und 3.11.

```
System.out.println("vor try");
try {
   System.out.println("vor read");
   // "read()" kann schief gehen - d.h. koennte eine Exception werfen
   System.in.read();
   System.out.println("nach read");
} catch (Exception x) {
   // Hierhin verzweigt der Programmfluss im Fehlerfall
  System.out.println("Fehlerbehandlung");
}
System.out.println("nach try/catch");
```

Ein zweiter Fall, in dem wir uns auch im Augenblick schon um Exceptions kümmern müssen, ist wenn Fehler möglich sind, und diese durch Exceptions gemeldet werden. Dies findet sich z.B. bei Konvertierungen (siehe Kapitel 3.11) oder bei fehlerhaften Array-Zugriffen (siehe Kapitel 3.5).

3.10 Eingabe

Die Eingabe von der Kommandozeile war unter Java lange Zeit recht kompliziert, da mehrere Streams und Reader miteinander verbunden werden mussten, und Checked-Exceptions

beteiligt waren. Prinzipiell ist der Mechanismus mit den Streams und Readern sehr leistungsfähig – nur für einen Anfänger und eine so alltägliche Aufgabe nicht angemessen. Mit dem JDK 1.5 konnte das Einlesen mit Hilfe der Klasse „Scanner“ und der Element-Funktion „nextLine()“ leicht vereinfacht werden. Seit dem JDK 1.6 kann direkt auf die Kommandozeile zugegriffen und eine Zeile als String eingelesen werden.

Das folgende Beispiel (für das JDK 1.6) zeigt, wie man eine Zeile als String von der Kommandozeile einliest und auswertet. Ignorieren Sie erstmal die Dinge, die wir noch nicht eingeführt haben wie z.B. die Kontrollstrukturen „while“, „if“ und „break“. Der grundsätzliche Ablauf des Programms sollte auch so klar sein. Es ist ein Echo-Programm, das alle Eingaben des Nutzers direkt wieder in doppelten Anführungs-Zeichen auf der Kommandozeile ausgibt – mit der Information wieviele Zeichen eingelesen wurden.

```
public class Chapter0310Ex01 {

   public static void main(String[] args) {
      System.out.println("Echo-Programm - JDK 1.6");
      while (true) {
         System.out.print("> ");
         String in = System.console().readLine();
         if (in.length() == 0) {
            break;
         }
         System.out.println(" \"" + in + "\" - " + in.length() + " Zeichen");
      }
      System.out.println("Programm-Ende");
   }

}
```

```
mögliche Ausgabe
Echo-Programm - JDK 1.6
> Hallo
 "Hallo" - 5 Zeichen
> Ich lerne jetzt Java
 "Ich lerne jetzt Java" - 20 Zeichen
>
Programm-Ende
```

Nehmen Sie das Beispiel erstmal so hin, falls nicht alles klar ist – im Laufe der Vorlesung werden sich die einzelnen Fragen dazu aufklären. Und wenn sie Eingabe machen müssen, benutzen sie das Beispiel ganz pragmatisch als Vorlage und passen es im Rahmen ihrer Kenntnisse und Bedürfnisse an. Die jeweils eingelesene Zeile findet sich innerhalb der While-Schleife im String „in“, und kann dann von ihnen genutzt werden.

Achtung – das obige Beispiel funktioniert problemlos auf der Konsole, aber nicht in der Eclipse. Die Eclipse startet ihr Java-Programm im Hintergrund mit einer Umgebung ohne echter Konsole. Der Aufruf von „console()“ liefert dort „null“ zurück und erzeugt dann eine Null-Pointer-Exception. Dies ist ein bekannter Bug in der Eclipse

- https://bugs.eclipse.org/bugs/show_bug.cgi?id=122429
- http://stackoverflow.com/questions/104254/java-io-console-support-in-eclipse-ide

Darum sind alle Beispiele in diesem Tutorial noch mit dem folgenden JDK 1.1 Mechanismus umgesetzt.

Aus historischen Gründen – und da Ihnen dieser Code in der Praxis noch häufig begegnen könnte (z.B. hier im Tutorial, da ich die Eclipse benutze) – hier auch die Beispiele aus ganz alten Zeiten (ab JDK 1.1) und für das JDK 1.5. Die Programme machen genau das Gleiche wie das erste Beispiel – sind nur viel mehr Code und schwerer zu verstehen.

Das Beispiel für das JDK 1.1 zeichnet sich dadurch aus, dass

- es Streams & Reader benutzt, und
- eine Checked-Exception abfangen **muss**, die von „reader.readLine()" geworfen werden könnte.

```
import java.io.InputStreamReader;
import java.io.BufferedReader;

public class Chapter0310Ex02 {

   public static void main(String[] args) {
      try {
         System.out.println("Echo-Programm - JDK 1.1");
         InputStreamReader isr = new InputStreamReader(System.in);
         BufferedReader reader = new BufferedReader(isr);
         while (true) {
            System.out.print("> ");
            String in = reader.readLine();
            if (in.length()==0) {
               break;
            }
            System.out.println(" \"" + in + "\" - " + in.length() + " Zeichen");
         }
      } catch (Exception x) {
         System.out.println("Unerwarteter Fehler");
      }
      System.out.println("Programm-Ende");
   }

}
```

Hinweis – da Einlesen schief gehen kann sind hier Exceptions möglich, und hier sind diese Checked-Exceptions und müssen daher abgefangen werden – vergleiche Kapitel 3.9. Sie können ja mal den try-catch-Block weg lassen – dann wird der Compiler einen Compiler-Fehler melden.

Die dritte Implementierung unseres Echo-Programms basiert auf dem JDK 1.5 und nutzt einen „Scanner". Da Scanner geschlossen werden müssen und unser Beispiel 100% korrekt sein soll, müssen wir auch die Fälle berücksichtigen, wo in unserem Programm etwas schief geht und dieses Problem durch eine Exception gemeldet wird (vergleiche Kapitel 3.9). Dadurch gewinnen wir wieder etwas Exception-Handling, diesmal in Form eines Try/Finally-Blocks:

```
import java.util.Scanner;

public class Chapter0310Ex03 {

   public static void main(String[] args) {
      System.out.println("Echo-Programm - JDK 1.5");
      Scanner sc = new Scanner(System.in);
      try {
         while (true) {
            System.out.print("> ");
```

```
            String in = sc.nextLine();
            if (in.length() == 0) {
               break;
            }
            System.out.println(" \"" + in + "\" - " + in.length() + " Zeichen");
         }
      } finally {
         sc.close();
      }
      System.out.println("Programm-Ende");
   }

}
```

Sie sehen, nutzen Sie moderne JDKs und bleiben beim Code des ersten Beispiels für das Einlesen von der Kommandozeile.

3.11 Konvertierungen

In den Übungs-Aufgaben werden ab und zu Konvertierungen von Strings zu Ganz- oder Gleitkomma-Zahlen benötigt. Diese Konvertierungen können über Klassen-Funktionen der Klassen „Integer“ und „Double“ vorgenommen werden. Achtung – die Konvertierungen können natürlich schief gehen – in diesem Fall wirft die Parse-Funktion eine „NumberFormatException“ Exception. Auch dies ist also ein Beispiel dafür, dass Sie sich in Java immer wieder um Exceptions kümmern müssen – auch wenn wir sie noch gar nicht richtig kennen.

```
public class Example {

   public static void main(String[] args) {
      if (args.length==0) {
         System.out.println("Kein Argument");
         return;
      }

      System.out.println("Arg: " + args[0]);
      try {
         int i = Integer.parseInt(args[0]);
         System.out.println("i: " + i);
      } catch (NumberFormatException x) {
         System.out.println(args[0] + " laesst sich nicht in int wandeln");
      }

      try {
         double d = Double.parseDouble(args[0]);
         System.out.println("d: " + d);
      } catch (NumberFormatException x) {
         System.out.println(args[0] + " laesst sich nicht in double wandeln");
      }
   }

}
```

Hinweis – die Klassen „Integer“ und „Double“ sind quasi die Objekt-Analogien zu den elementaren Typen „int“ und „double“. Sie werden z.B. als Wrapper-Klassen für Container benutzt, und sie bieten allgemeine Hilfs-Funktionen wie z.B. „parseInt“ für den jeweiligen Typ an.