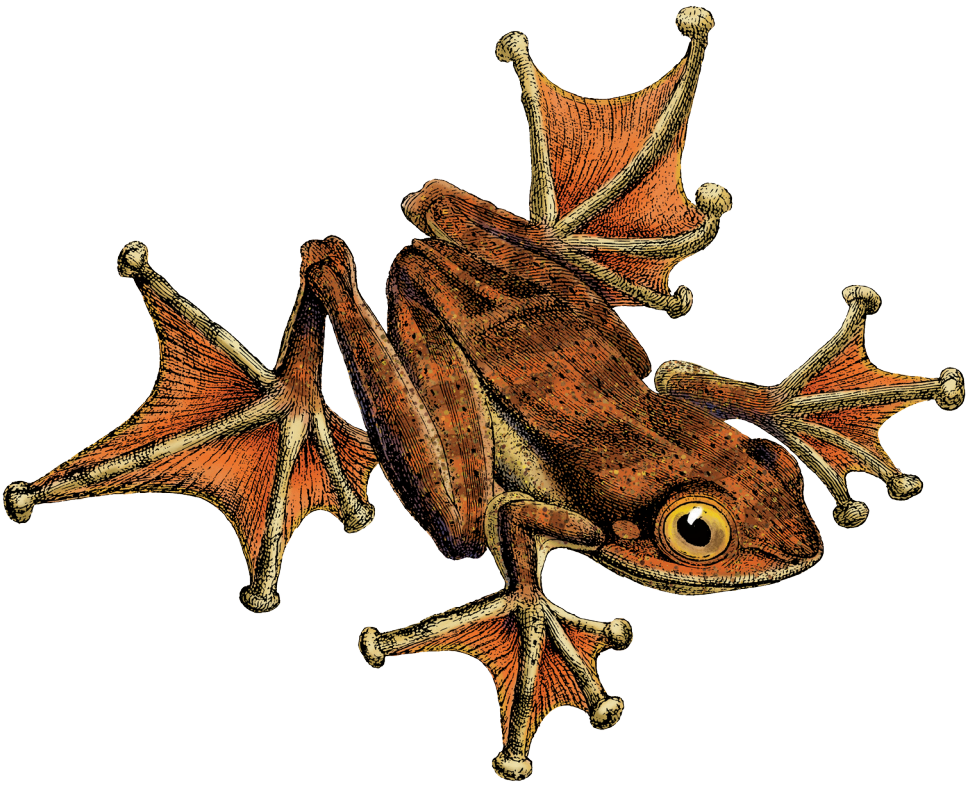
O'REILLY®

Java für die Life Sciences

Eine Einführung in die angewandte Bioinformatik



Jens Dörpinghaus,
Sebastian Schaaf & Vera Weil

Java für die Life Sciences

Eine Einführung in die angewandte Bioinformatik

Jens Dörpinghaus, Sebastian Schaaf, Vera Weil

O'REILLY®

Jens Dörpinghaus, Sebastian Schaaf, Vera Weil

Lektorat: Alexandra Follenius
Fachgutachten: Alexander Apke, Heike Kattenbusch, Christof Meigen,
Kristian Rother, Helena Schock
Korrektorat: Sibylle Feldmann, *www.richtiger-text.de*
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Karen Montgomery, Michael Oréal, *www.oreal.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-96009-125-7
PDF 978-3-96010-342-4
ePub 978-3-96010-343-1
mobi 978-3-96010-344-8

1. Auflage 2021

Wieblinger Weg 17
69123 Heidelberg

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.
O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: kommentar@oreilly.de.

5 4 3 2 1 0

Inhalt

Einleitung

Die Biologie mit all ihren vielfältigen Teilgebieten wie Botanik, Zell- und Molekularbiologie, Genetik oder Humanbiologie ist als ein reges Forschungsfeld in stetem Wandel. Sie ist gleichzeitig, mehr als andere Domänen, ständigen Wechselwirkungen mit angrenzenden Bereichen wie Chemie, Physik oder Medizin und seit einigen Jahrzehnten auch technischeren Bereichen wie der Informatik unterworfen. Gerade durch die Interaktion mit der Informatik hat sich ein eigenes Fachgebiet entwickelt, die Bioinformatik. In der Praxis führt das dazu, dass bei vielen Naturwissenschaftlern inzwischen weitreichende Programmiererfahrungen vorhanden sind, die deutlich über Tabellenkalkulationsprogramme hinausgehen und nicht nur die Thematik der Handhabung wissenschaftlicher Daten umfasst.

Zugleich hat sich in den letzten Jahren die Biologie gemeinsam mit der Medizin, der Chemie sowie den Pflege- und den Agrarwissenschaften zu einem spannenden, interdisziplinären Feld entwickelt: den Lebenswissenschaften (engl. *Life Sciences*). In diesem Gebiet fließen die Grenzen der »klassischen« Disziplinen zusehends ineinander. Ein Buch, das sich lediglich mit der Schnittmenge zwischen Biologie und Informatik beschäftigt, erscheint uns daher nicht weitreichend genug. Wir haben uns das Ziel gesteckt, den Bogen hier wesentlich weiter zu spannen.

Dementsprechend geht es in diesem Buch primär darum, angewandte Probleme der Lebenswissenschaften mithilfe der Informatik zu lösen. Es wird dabei nur so weit auf biologische oder medizinische Hintergründe eingegangen, wie es für das Verständnis des Buchs nötig ist. Das Vorgehen wird mit viel Beispielcode, teilweise im Stil eines Kochbuchs, dargestellt. Wann immer möglich, sollen Übungsaufgaben und Praxisprojekte tiefer in die Materie führen.

In diesem Buch werden vor allem folgende allgemeine Themenbereiche behandelt:

- *Big Data*: die Verarbeitung einer großen Menge sich ständig ändernder Daten.
- Das Heranziehen und Adaptieren effizienter, robuster und zuverlässiger Lösungen (insbesondere *Heuristiken* und *Algorithmen*) für bestehende Probleme.
- Informationen über *State-of-the-Art-Lösungen*, *-Bibliotheken* und *-Algorithmen*.

Somit werden alle wichtigen Grundlagen für Studierende und Praktiker in den Lebenswissenschaften, der Biologie, der Bioinformatik und anderen Fächern behandelt: Wie werden Daten und Informationen verwaltet (sogenanntes *Data Management*), wie werden sie über Algorithmen und externe Bibliotheken verarbeitet, und wie werden damit schlussendlich Probleme im Sinne der zugrunde liegenden wissenschaftlichen Fragestellung gelöst? Unser Ziel ist es, Ihnen eine Vorstellung von den verfügbaren und etablierten Softwarelösungen und Konzepten sowie von generellen Herangehensweisen bei datenbasierten Problemstellungen zu vermitteln. Nicht zuletzt legen wir Wert auf Verweise darauf, wo Sie auch über dieses Buch hinaus weitere Hilfe finden können.

Entwickelt hat sich das vorliegende Buch aus zahlreichen Lehrveranstaltungen, die die Autoren an den Universitäten Bonn, Köln und München über die vergangenen Jahre hinweg angeboten haben; hinzu kommen die eigenen Erfahrungen aus der fachlichen und beruflichen Praxis. Wir haben während der Ausarbeitung und schriftlichen Niederlegung der Kursmanuskripte von der interdisziplinären Arbeit zwischen Mathematik, Informatik, Biologie und den weiteren Lebenswissenschaften stark profitiert und hoffen, dies an Sie weitergeben zu können. Unser Dank gilt auch den Generationen von Studenten, die durch ihre Rückfragen und Rückmeldungen die Darstellung geschärft haben und dadurch helfen konnten, den Fokus dieses Buchs entsprechend auszurichten.

Bei der Themenauswahl haben wir uns von drei Faktoren leiten lassen: Welche Themen sind derzeit wichtig und emergent? Welche Themen sind bereits langfristig in Erscheinung getreten und bleiben somit auch noch für mindestens eine weitere Generation von Bedeutung? Und zu guter Letzt haben uns natürlich auch unsere persönlichen Präferenzen und Interessen geleitet. Mehr zu den einzelnen Themen berichten wir, wenn wir den Aufbau des Buchs beschreiben.

Java in den Lebenswissenschaften

Es muss auch erwähnt werden, dass die Lebenswissenschaften ein sehr agiles Feld sind und es fast täglich zu Neuentwicklungen in den verschiedenen thematischen Bereichen wie z.B. Assays, Datentypen, Dateiformaten und Software kommt. Doch bleiben grundlegende Konzepte weitestgehend davon unberührt, und auch die Trends zur Verwendung neuer Programmiersprachen verlaufen eher langsamer. Daher erscheint es uns nach wie vor sinnvoll und wichtig, Java als Grundlage für dieses Buch anzunehmen. Java erblickte 1995 das Licht der Welt und verfügt damit über eine große Anzahl von Bibliotheken und eine große Entwicklergemeinde. Darüber hinaus zeichnet sich Java durch die hohe Prozessierungsgeschwindigkeit und seine Portabilität aus; es kann auf allen gängigen Systemen ohne Probleme ausgeführt werden.

Auch wenn der größte Teil wissenschaftlicher Infrastruktur auf Unix/Linux aufbaut, können daher die Beispiele aus diesem Buch ebenso auf Mac- oder Windows-

Systemen ausgeführt und weitestgehend problemlos übernommen werden. Sollte es hier Besonderheiten geben, weisen wir an Ort und Stelle darauf hin. Die Ausnahme bildet der immer wieder vorkommende Hinweis auf BASH, insbesondere wenn es um HPC-Umgebungen (High-Performance Computing) geht. Diese Shell lässt sich unseres Wissens nur auf einer ganz kleinen Anzahl von Windows-Systemen nutzen. Auch verweisen wir bei grafischen Installationsroutinen unter Windows grundsätzlich auf andere Quellen. An manchen Stellen gehen wir also implizit von einer Unix-artigen Umgebung aus.

Kritiker mögen bemängeln, dass doch gerade in den Lebenswissenschaften Python (entstand 1991) oder Matlab (das auf die 1970er-Jahre zurückgeht) benutzt würde. Dem möchten wir allerdings entgegensetzen, dass dies schon beim Blick auf eine andere Forschungseinrichtung, in eine andere Arbeitsgruppe oder bereits bei einer konkreten Aufgabe nicht mehr der Fall sein muss. Im Gegensatz zu den oben genannten Programmiersprachen ist Java schlicht schon länger in manchen Bereichen präsent und etabliert – selbst in professionellen Umgebungen weit außerhalb der Wissenschaft. Ebenso wird es in vielen Open-Source-Projekten verwendet: Ein nicht unerheblicher Teil der verfügbaren wissenschaftlichen Software ist in Java entwickelt worden. Prominente Beispiele sind etwa das GATK-Toolkit zur Analyse von DNA-Sequenzen und die Bildverarbeitungsumgebung ImageJ. Letztlich bleibt bei dieser Fragestellung aber immer eine gewisse Ambivalenz, und deswegen blicken wir an manchen Stellen auch über Java hinaus, z. B. auf BASH.

Prinzipiell bietet sich Java auch aufgrund der Vielzahl hochqualitativer kostenloser Tutorials für einen Start in die Programmierung und speziell in die Objektorientierung an. Wir setzen daher explizit einfache Programmierkenntnisse voraus – mehr dazu im folgenden Abschnitt. Die Vorkenntnisse, sofern sie nicht in Java vorliegen, können sehr leicht auf Java übertragen werden, und auch umgekehrt profitieren Sie von den neuen Erkenntnissen aus diesem Buch, wenn Sie in anderen Sprachen programmieren. Probleme, die nur in einer speziellen Sprache gelöst werden können, gibt es heutzutage nur noch sehr selten – aber es gibt viele Probleme, die sich in Java besonders einfach lösen lassen.

Zielgruppe und Voraussetzungen dieses Buchs

Zielgruppe dieses Buchs sind Studierende, Berufstätige und Dozenten der Biologie, der Bioinformatik, der Informatik, der Life Sciences und allen verwandten Naturwissenschaften. Sie sollten Grundkenntnisse im Bereich der Biologie und Lebenswissenschaften sowie Kenntnisse mindestens einer höheren Programmiersprache mitbringen – oder zumindest die Bereitschaft, sich entsprechend einzulesen.

Wenn Sie Grundkenntnisse in der Algorithmik mitbringen, werden Sie diesem Buch schneller folgen können. Wir setzen dies aber bewusst nicht voraus, stattdessen verweisen wir stets auf die Grundlagenkapitel in diesem Buch und beschreiben – wann immer es nötig ist – die Algorithmen und Herangehensweisen. Der Leser

soll zu einem selbstständigen Herangehen motiviert werden. Wir geben allerdings zu bedenken, dass dieses Buch kein allumfassendes Kompendium zu dieser Thematik anbietet, sondern den Leser mit entsprechenden Verweisen weiterleitet.

Aufbau dieses Buchs

Die ersten drei Kapitel sind als Grundlagenkapitel zu verstehen. In Kapitel 1, *Einführung in die Arbeit mit Java*, widmen wir uns der Einrichtung der Arbeitsumgebung, der Versionsverwaltung und dem Build-Tool Maven. Hier liegt der Fokus auf der Software und den Tools. Gerade für Anwender, die nur gelegentlich mit Java arbeiten, ist dieses Kapitel dringend zu empfehlen. Dabei werden wir uns allerdings nicht mit den Tiefen von Maven beschäftigen, sondern eine praxisgerechte Schnelleinführung geben. Ziel ist es, alle Aufgaben in diesem Buch schnell und effizient lösen zu können.

Kapitel 2, *Java zum Auffrischen*, ist als Angebot an die Leser gedacht, die ihre Grundkenntnisse in der Programmierung auffrischen möchten oder Programmierkonzepte in Java gegebenenfalls nachschlagen wollen. Dazu werden einzelne Themen wie der Aufbau eines Programms, Bibliotheken, Variablen und Schleifen übersichtlich dargestellt. Beispiele und weitere Erklärungen bieten einen einfachen Einstieg.

Den Abschluss dieses Themenblocks bildet Kapitel 3, *Data Engineering mit Java*. Hier wird in die Grundprinzipien des Data Engineerings eingeführt, und mit vielen Beispielen werden Konzepte wie XML, JSON, API-Schnittstellen etc. dargestellt und ausgeführt. Hier finden sich auch erste Beispiele aus den Lebenswissenschaften. Auf diese einführenden Kapitel wird im Folgenden immer wieder verwiesen. Der erfahrene Leser kann direkt bei den späteren Kapiteln anfangen und bei Bedarf auf diese Kapitel zurückgreifen.

Sinnvoll erschien es uns, an dieser Stelle direkt das Thema *Data Mining* (Kapitel 4) anzuschließen. Hier beschreiben wir die klassischen Themen dieses Felds und insbesondere die Themen Klassifizierung und Clustering. In diesem Kapitel widmen wir uns primär den Grundlagen, da diese Methoden in fast allen darauffolgenden Kapiteln verwendet werden.

Die weiteren Kapitel behandeln konkrete Problemstellungen aus den Lebenswissenschaften. Zunächst wird auf die *Netzwerkanalyse: Graphen mit Java* in Kapitel 5 eingegangen. Dieses Thema erschien uns besonders wichtig, da Netzwerke sehr häufig verwendet, aber selten thematisch und technisch solide vorgestellt werden. Neben der technischen Einführung in die Bibliothek JGraphT geben wir hier viele Beispiele für gerichtete, ungerichtete oder auch andere Graphen und die Probleme, die damit gelöst werden können. Auch wenn wir den Versuch gewagt haben, die Themen möglichst breit aufzustellen, können wir keine allumfassende Einführung in dieses Thema geben. So können spezielle Aspekte wie phylogenetische Bäume hier nicht behandelt werden.

In Kapitel 6, *Bildverarbeitung mit Java und ImageJ,* findet sich eine umfangreiche Einführung in die Arbeit mit der Bibliothek ImageJ. Viele Probleme der Lebenswissenschaften beruhen auf Bilddaten, z.B. die Tumorerkennung oder die automatisierte Analyse von Pflanzenwuchs. Auch hier müssen wir auf viele teilweise sehr spezifische Aufgaben der Bildverarbeitung verzichten. So werden Sie vielleicht eine detaillierte Einführung in die KI-Methoden dieses Gebiets vermissen, auch wenn wir über die Verwendung von WEKA sprechen werden. Die folgenden Themen werden im Speziellen behandelt: die Partikelanalyse, die Objektklassifizierung und die Farbanalyse. Darüber hinaus runden zwei Praxisprojekte aus der Bildverarbeitung das Kapitel ab. Die Themenauswahl wurde insofern mit Bedacht gewählt, als dass dadurch alle wichtigen Methoden (und Fallstricke) von ImageJ als Bibliothek behandelt werden und der Leser sich sehr zügig weitere Felder erschließen kann.

Ein klassischer Themenbereich der Bioinformatik wird in Kapitel 7, *Sequenzanalyse mit BioJava*, aufgegriffen. Dieses Feld hat Anfang der 2000er-Jahre den Sprung zum *Next-Generation Sequencing* (NGS) gemacht und wird uns, da es die Bausteine des Lebens betrachtet, auch die nächsten Jahrzehnte beschäftigen. Es ist daher nicht nur ein wichtiger Bestandteil, sondern auch ein würdiger Abschluss unserer Themenauswahl. In der Einführung zu diesem Kapitel beschäftigen wir uns mit den programmiertechnischen Grundlagen der Sequenzanalyse mit BioJava und gehen auch auf die Datenbanksuche ein. Anschließend werden wir einzelne Themen wie »Multiple Sequence Alignment«, »BLAST« und »NGS« behandeln und in Übungsaufgaben vertiefen. Auch an dieser Stelle mussten wir die Themenauswahl sinnvoll einschränken; hierbei haben wir uns davon leiten lassen, welche Themen zügig und vor allem sinnvoll mit BioJava umgesetzt werden können. Für viele Spezialprobleme benötigen Sie dann weitere Bibliotheken oder Anwendungen. Diese können mit dem Wissen aus diesem Buch aber gut in eigene Workflows eingebettet werden.

Übungsaufgaben und Lösungen

Alle Kapitel dieses Buchs sind mit Übungsaufgaben versehen, in denen der Leser das Gelernte vertiefen und anwenden kann. Die Übungsaufgaben in den ersten zwei Kapiteln haben noch keinen Bezug zu den Lebenswissenschaften, oder der Bezug wird bestenfalls sehr künstlich hergestellt, da es sich hier um Grundlagen der Programmiersprache Java handelt. In allen folgenden Kapiteln werden vor allem Praxisprobleme aus den Lebenswissenschaften als Aufgaben verwendet. Wir stellen weiteres Material (wie Eingabedateien), Lösungsvorschläge, Lösungsanregungen und teilweise auch Lösungen in einem GitHub-Repository unter *https://github.com/jd-s/Java-fuer-die-Life-Sciences-Loesungen* zur Verfügung. Gern können Sie uns schreiben, wenn Sie eine elegantere, schnellere oder schönere Lösung zur Verfügung stellen wollen. Bei der Programmierung gibt es nicht eine einzige, sondern viele verschiedene Lösungen, die zum Ziel führen.

Weitere Informationsquellen

Dieses Buch ist selbstverständlich nicht das einzige Werk zu diesem Thema, auch wenn es unseres Wissens das einzige aktuelle deutschsprachige Werk ist. Wir haben für Sie verschiedene Literaturangaben und Verweise auf Internetressourcen zu einzelnen Themen zusammengestellt. Weitere Verweise finden Sie in den einzelnen Kapiteln in den Fußnoten.

Java im Allgemeinen:

- Christian Ullenboom: *Java ist auch eine Insel* (Rheinwerk 2020, die Ausgabe von 2017 ist auch als Internetressource verfügbar unter *http://openbook.rheinwerk-verlag.de/javainsel/*).
- Dirk Louis, Peter Müller: *Java – Der umfassende Programmierkurs* (O'Reilly 2014).
- Robert Liguori, Patricia Liguori: *Java – kurz & gut* (O'Reilly 2018).

Im Zusammenhang mit *Life Science Informatics und Java* sind uns nur zwei Veröffentlichungen bekannt:

- Peter Garst: *Mastering Java through Biology* (letzte Version von 2014; arbeitet mit Java 8).
- Harshawardhan Bal, Johnny Hujol: *Java for Bioinformatics and Biomedical Applications* (Springer 2010).

Bioinformatik und Life Science Informatics:

- Lee Harland, Mark Forster: *Open Source Software in Life Science Research: Practical Solutions to Common Challenges in the Pharmaceutical Industry and Beyond*. Woodhead Publishing Series in Biomedicine (Elsevier, 2012).
- Thomas Dandekar, Meik Kunz: *Bioinformatik: Ein einführendes Lehrbuch* (Springer 2017).
- Andrea Hansen: *Bioinformatik: Ein Leitfaden für Naturwissenschaftler* (Springer 2013).
- Martin Dugas, Karin Schmidt: *Medizinische Informatik und Bioinformatik: Ein Kompendium für Studium und Praxis* (Springer 2003).

Die in diesem Buch verwendeten Konventionen

In diesem Buch werden die folgenden typografischen Konventionen verwendet:

Kursiv
: Kennzeichnet neue Begriffe, URLs, E-Mail-Adressen, Dateinamen und Dateierweiterungen.

`Feste Zeichenbreite`
: Kennzeichnet Programmlistings sowie Programmelemente in Absätzen, wie etwa Variablen- oder Funktionsnamen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter.

`Feste Zeichenbreite fett`
: Zeigt Befehle oder anderen Text an, der vom Benutzer wortwörtlich eingegeben werden soll.

`Feste Zeichenbreite kursiv`
: Kennzeichnet Text, der durch Werte ersetzt werden soll, die der Benutzer vorgibt oder die sich aus dem Kontext ergeben.

Dieses Symbol kennzeichnet einen Tipp oder Vorschlag.

Dieses Symbol zeigt eine allgemeine Bemerkung.

Dieses Element symbolisiert einen Warnhinweis.

Verwendung von Codebeispielen

Dieses Buch ist dazu gedacht, Ihnen bei Ihren Aufgaben zu helfen. Grundsätzlich dürfen Sie die mit diesem Buch bereitgestellten Codebeispiele in Ihren Programmen und der dazugehörigen Dokumentation nutzen. Sie müssen uns dazu nicht um Erlaubnis fragen, es sei denn, Sie reproduzieren einen beträchtlichen Teil des Codes. Schreiben Sie beispielsweise ein Programm, das mehrere Codeabschnitte aus diesem Buch enthält, benötigen Sie keine Erlaubnis. Verkaufen oder verteilen Sie dagegen eine CD-ROM mit Beispielen aus Büchern von O'Reilly, brauchen Sie eine Erlaubnis. Eine Frage mit einem Zitat aus diesem Buch unter Angabe eines Codebeispiels zu beantworten, benötigt keine Erlaubnis. Wollen Sie dagegen einen wesentlichen Anteil der Codebeispiele aus diesem Buch in Ihr eigenes Buch integrieren, brauchen Sie eine Erlaubnis.

Wir freuen uns über Zitate, verlangen diese aber nicht. Ein Zitat enthält üblicherweise Titel, Autor, Verlag und ISBN, zum Beispiel: »*Java für die Life Sciences: Eine Einführung in die angewandte Bioinformatik* von Jens Dörpinghaus, Sebastian Schaaf und Vera Weil (O'Reilly 2021), ISBN 978-3-96009-125-7«.

Wenn Sie glauben, dass Ihre Nutzung von Codebeispielen über das gewöhnliche Maß hinausgeht oder außerhalb der oben vorgestellten Nutzungsbedingungen liegt, kontaktieren Sie uns bitte unter *kommentar@oreilly.de*.

Danksagung

Wir danken unseren Gutachterinnen und Gutachtern, die den Entstehungs- und Reifeprozess dieses Buchs positiv begleitet haben. Besonders umfangreich und detailliert waren die Rückmeldungen von Christof Meigen, Heike Kattenbusch und Kristian Rother. Hervorheben möchten wir auch den positiven Einfluss vieler Studierender der letzten Jahre – deren Rückfragen, Kritik und teils überraschende Leistungen in der Bearbeitung von Übungsaufgaben und Softwareprojekten stellen letztlich das Fundament dieses Buchs dar. Für das geduldige und motivierende Lektorat unseres Erstlingswerks möchten wir uns bei Alexandra Follenius herzlich bedanken.

KAPITEL 1

Einführung in die Arbeit mit Java

Dieses Kapitel behandelt zwei Themen: Zuerst besprechen wir im folgenden Abschnitt, wie Sie eine Entwicklungsumgebung auswählen und einrichten. Hier finden Sie Informationen zur Installation von JDK, Eclipse, Git und Maven, bebildert mit vielen Screenshots. Der nächste Abschnitt beschreibt die erste Java-Anwendung und damit den Weg zum ersten Programm und dessen Ausführung.

Die aktuellste Version von Java zur Drucklegung des Buchs ist Java 13. Wir verwenden allerdings keine speziellen Technologien, die eine neuere Version als Java 8 benötigen. Zudem wird Java 8 als LTS-Version (*Long-Time-Support*) immer noch unterstützt. Da alle folgenden Informationen in der Regel auch bei Verwendung einer anderen Version in gleicher Weise gelten, verweisen wir nur dort explizit auf spezifische Versionen, wo Besonderheiten zu beachten sind.

Die Umgebung einrichten

Um mit der Java-Programmierung zu beginnen, muss zunächst das *Java Development Kit* (JDK) installiert werden. Normalerweise sollte auch das JRE (*Java Runtime Environment*) ausreichen, um Java-Anwendungen auszuführen, aber wir benötigen noch einige weitere Tools wie den Compiler, die nur mit dem JDK geliefert werden. Daher installieren wir jetzt erst einmal das JDK.

JDK installieren

Das *Java Runtime Environment* beinhaltet die *Java Virtual Machine* (JVM). Dies ist der Ort, an dem Java-Anwendungen ausgeführt werden, und der Grund dafür, dass Java-Codes über Betriebssysteme hinweg portabel sind, denn nur die JVM wird spezifisch für einzelne Betriebssysteme wie Windows, Linux oder Unix entwickelt. Das JDK ist das vollständige *Software Development Kit* für Java. Es beinhaltet auch das JRE sowie den Compiler und andere Tools wie JavaDoc (Dokumentation) und den Java-Debugger (interaktive Fehlersuche). Auch wenn man die zusätzlichen

Funktionen des JDK nicht nutzen möchte, ist es auf alle Fälle sinnvoll, es trotzdem zu installieren.

Installation des JDK

Die folgende Anleitung ist sehr kurz. Wir werden nur die wichtigsten Werkzeuge besprechen und Hinweise dazu geben, wo weitere Informationen und Hilfen für die Installation dieser Werkzeuge bezogen werden können.

Das neueste JDK von Oracle findet sich unter *http://www.oracle.com/technetwork/java/javase/downloads/index.html*. Wird unter Windows oder macOS gearbeitet, lädt man das entsprechende Paket herunter und folgt den Installationsanweisungen.

Bei Verwendung von Linux oder Unix ist der beste Weg die Nutzung des Paketmanagers des Systems. Für ein Debian-basiertes System ist das beispielsweise *aptitude*:

```
apt-get install default-jdk
```

Bei Centos, Scientific Linux oder Red Hat Linux kommt dagegen *yum* zum Einsatz:

```
yum install java-1.8.0-openjdk
```

Meist gibt es für die Installation mehrere geeignete Versionen von Java, oben beispielsweise die v1.8. Aktuelle Informationen zu einem spezifischen Betriebssystem oder einer spezifischen Java-Version finden sich im Internet.

Unter Linux wird manchmal *OpenJDK* angeboten. Sowohl das OpenJDK als auch das Oracle JDK werden von der Firma Oracle erstellt und gepflegt und sind Implementierungen der gleichen Java-Spezifikation. OpenJDK ersetzt lediglich einige proprietär lizenzierte Teile, sodass praktisch nur die Lizenzierung den Unterschied zwischen beiden JDKs ausmacht. Aber auch hier können im Detail natürlich Probleme oder Unterschiede auftreten. Bei näherem Interesse finden Sie weitere Informationen unter *http://openjdk.java.net/*.

Die Installation des JDK stellt Ihnen Kommandozeilenanwendungen wie *java* und *javac* zur Verfügung. Die erste wird verwendet, um Java-Anwendungen auszuführen, während die zweite der Java-Compiler ist, der Quellcode in eben jene ausführbaren Anwendungen übersetzt. Möchte Sie eine grafische Benutzeroberfläche verwenden, bietet sich Eclipse als vollwertige Entwicklungsumgebung an: Anweisungen zu dessen Einrichtung finden Sie im nächsten Abschnitt.

Der Java-Compiler

Der Java-Compiler erzeugt *class*-Dateien, die Bytecode enthalten. Dies ist weder ein vollständig kompilierter binärer noch ein vollständig interpretierter Code. Bytecode platziert sich in seiner Art irgendwo »dazwischen«. Er ist kompiliert, wird aber von der Java-Laufzeitumgebung interpretiert und läuft in einem speziellen Container. Pakete werden in der Regel als *jar*-Dateien angeordnet, die technisch betrachtet aber eigentlich *tar*-Archive mit Bytecode sind (und ebenso manuell entpackt werden können).

Bevor wir uns nun der Arbeit mit einer GUI beschäftigen, wollen wir Ihnen zumindest ein paar einführende Beispiele dafür geben, wie Sie Java in der Kommandozeile verwenden können. Dies sind Grundlagen, um eine Anwendung zum Beispiel auf einer HPC-Umgebung (High-Performance Computing) auszuführen.

Java in der Kommandozeile

Java-Code wird normalerweise in **.java*-Dateien gespeichert, die nur Text enthalten. Diese Dateien können mit fast jedem beliebigen Texteditor bearbeitet werden. Das schließt zum Beispiel den von Ihnen bevorzugten Kommandozeilentexteditor (vim, nano, joe ...) oder nahezu jeden grafischen Texteditor (gedit, pluma, kate, notepad ...) ein.

Gute Alternativen finden sich in integrierten Entwicklungsumgebungen (IDEs) wie Eclipse, auf die später eingegangen wird. Diese Anwendungen packen alle Befehlszeilenaufrufe in übersichtliche kleine Symbole, kommunizieren direkt mit *java*/*javac*, bieten die Möglichkeit, mit einem Versions-Repository und Paketverwaltungen zu interagieren, und noch vieles mehr. Dies erspart Ihnen nach der grundlegenden Einrichtung etwas Arbeit und erleichtert letztlich den Start.

An dieser Stelle gibt es ein »Aber« – denn es ist ebenfalls eine gute Idee, mit den Kommandozeilentools vertraut zu sein. Diese funktionieren aufgrund ihrer Einfachheit auch dann, wenn die grafische Benutzeroberfläche aus diversen, vielleicht schwer zu findenden Gründen »nicht mehr will«. Und es ist tatsächlich so, dass nach ein wenig Übung die Arbeit mit Kommandozeilentools wesentlich schneller abläuft.

Im Folgenden zeigen wir Ihnen die Verwendung von Java in der Befehlszeile für Unix- und Linux-Betriebssysteme. Die Befehle funktionieren auf alternativen Betriebssystemen wie Windows in ähnlicher Weise.

Zuerst müssen Sie herausfinden, welches Java installiert ist und ob es über die `path`-Umgebung korrekt verfügbar ist:

```
$ which java
/usr/bin/java
```

Der Befehl `which` prüft, ob die Anwendung verfügbar ist, und gibt, falls ja, den Pfad zurück. Andernfalls wird er mitteilen, dass die Anwendung nicht gefunden werden kann. Das heißt nicht, dass die Anwendung nicht vorhanden ist, sondern lediglich, dass sie der Systemverwaltung nicht (korrekt) bekannt gegeben wurde. Sie können die Version jetzt überprüfen, indem Sie `java` mit einem Parameter ausführen:

```
$ java -version
openjdk version "1.8.0_242"
OpenJDK Runtime Environment (build 1.8.0_242-b08)
OpenJDK 64-Bit Server VM (build 25.242-b08, mixed mode)
```

Nehmen wir an, wir hätten eine kleine Testklasse namens `Test.java`, die die folgenden Zeilen enthält:

```
public class Test {
    public static void main(String[] args) {
        System.out.println ("Hello World");
     }
 }
```

Sie können nun diese kleine Anwendung zu Bytecode kompilieren, indem Sie `javac` aufrufen:

```
$ javac Test.java
```

Dadurch wird eine Datei namens *Test.class* erzeugt. Sie können diese Klasse mit dem Befehl `java` ausführen:

```
$ java -cp . Test
```

Was ist aber eine *jar*-Datei? Diese Dateien sind Archive (komprimiert mit `tar`), die lediglich Klassendateien enthalten. Sie werden zur Auslieferung und Verwaltung komplexer Pakete eingesetzt. Sie können mit dem Befehl `jar` eine neue *jar*-Datei erstellen, die unsere Klassendatei enthält:

```
$ jar cf TestArch.jar Test.class
```

Dieser Befehl benötigt als letzten Parameter alle Dateien, die Sie der neuen *jar*-Datei hinzufügen möchten. Es können Ordner, Ressourcen (Datendateien) und Wildcards deklariert werden.

Wenn eine Klasse innerhalb einer *jar*-Datei ausgeführt werden soll (sagen wir, eine Klasse namens `Test` ist in einer *jar*-Datei `TestArch` enthalten), ist auch das einfach möglich – es muss nichts entpackt werden:

```
$ java -cp TestArch.jar Test
```

Darüber hinaus gibt Möglichkeiten, das sogenannte *Manifest* innerhalb einer *jar*-Datei zu ändern, um es einfacher ausführbar zu machen. Darauf kommen wir in Übungsaufgabe 4.4 auf Seite 16 noch zu sprechen.

Eine IDE einrichten: Eclipse

Obwohl es auch viele andere Entwicklungsumgebungen gibt, empfehlen wir Ihnen, die Open-Source-IDE *Eclipse* zu verwenden.[1] Alle IDEs haben ihre eigenen Vor- und Nachteile, allerdings ist Eclipse am weitesten verbreitet und wird von einer großen Community unterstützt, sodass hinsichtlich Dokumentation und Fehlersuche viele Informationen online verfügbar sind. In diesem Buch werden wir daher die Arbeit mit Eclipse beschreiben. Eclipse ist unter *http://www.eclipse.org/* verfügbar.

1 Andere IDEs, die von vielen Menschen verwendet werden, sind z. B. NetBeans von Oracle (*https://netbeans.org/*) oder IntelliJ von JetBrains (*https://www.jetbrains.com/idea/*). Auch VSCode von Microsoft ist häufig im Einsatz (*https://code.visualstudio.com/*).

Es ist selbst in Java implementiert und hochgradig anpassbar, und darüber hinaus existieren Millionen von Plug-ins, die das Leben stark vereinfachen.

Die Entwicklung von Eclipse begann 2001 bei IBM, und seit 2004 ist die Umgebung Open Source. Auf der Webseite finden sich die *Eclipse Downloads*, von denen die *Eclipse IDE for Java Developers* (unter Beachtung des passenden Betriebssystems) die richtige Wahl ist. Der heruntergeladene Installer führt durch den Installationsprozess. Es ist auch möglich, ein Paket herunterzuladen, das Binärdateien enthält und lediglich auf die Festplatte extrahiert werden muss.

Die aktuellste Version von Eclipse zur Drucklegung des Buchs ist die Version 2020-03. Wie bei Java gilt auch hier, dass wir nur dort auf spezifische Versionen eingehen, wo sich die Benutzung signifikant unterscheidet.

Nach dem ersten Start sollten Sie das Fenster *Willkommen bei Eclipse* deaktivieren. Es kann notwendig sein, das richtige JDK unter *Preferences → Java → Installed JREs* einzurichten, also die eigene Java-Installation mit Eclipse bekannt zu machen. Dies mag für Windows und Mac ein wenig knifflig sein, allerdings finden sich online verschiedene Ressourcen, die Ihnen dabei helfen.

Abbildung 1-1 zeigt einen Überblick über eine laufende Eclipse-Instanz in der Java-Perspektive *(Java perspective)*. Perspektiven sind nutzungsspezifische Designoberflächen. Eclipse kann hierüber einfach per Klick auf die jeweilige Aufgabe angepasst werden.

Der Wechsel zwischen diesen Perspektiven verändert das Layout so, dass Tabs und Fenster bzw. *Panels* für einen anderen Zweck leicht erreichbar sind. Auch Schaltflächen, Styles und weitere (optische) Konfigurationen werden durch die Aktivierung einer Perspektive beeinflusst. *Team Synchronizing* und *SVN Repository Exploring* bieten so beispielsweise optimierte Werkzeuge zur Synchronisierung des Quellcodes mit einem SVN- oder Git-Repository. Es stehen diverse Perspektiven zur Verfügung, z.B. solche für Debugging, Git, XML und viele andere, die auch nachinstalliert oder angelegt werden können.

In der Java-Perspektive finden Sie auf der linken Seite ❶ eine Übersicht des gesamten *Workspace*. Dies ist ein Verzeichnis an beliebiger wiederum konfigurierbarer Stelle im Dateibaum des Systems, in dem Eclipse alle *Projekte* speichert. Projekte und ihre Inhalte werden strukturiert als Dateibaum oder in einer Package-Betrachtung angezeigt. Die allgemeine Ausgabe (*Console*, aber auch Fehler- und Debug-Ansichten) befindet sich unten in ❷. Der Quellcode selbst, Java-Dateien und weitere Inhalte werden in der Regel zentral in ❸ angezeigt. In ❹ ist der Aufriss einer geöffneten Sourcecode-Datei zu sehen, über dieses Verzeichnis kann einfach zwischen Methoden gesprungen werden.

Es gibt diverse funktionale Tabs, die auch in andere Panels, sogar neue, verschoben werden können.

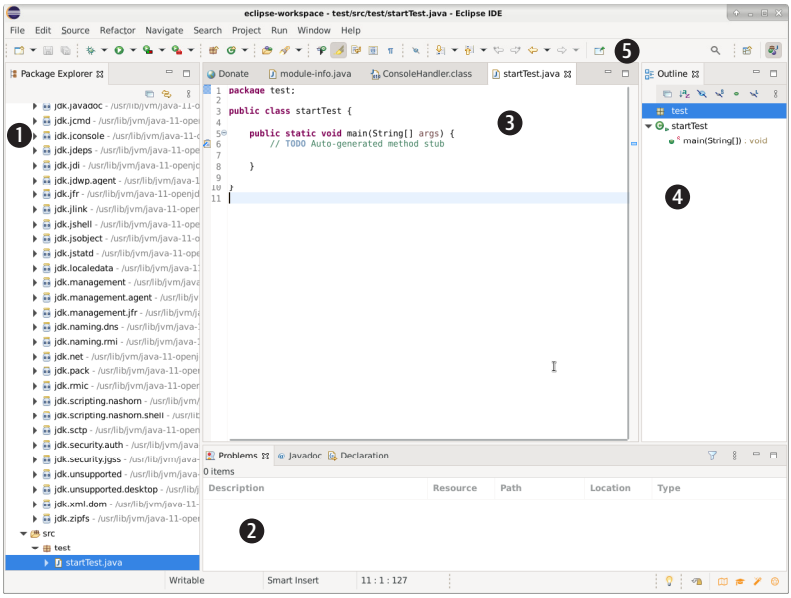


Abbildung 1-1: Eine laufende Eclipse-Instanz mit verschiedenen Bereichen: (1) Projekt-Explorer, (2) Log- und Debug-Ausgaben, (3) Quellcode, (4) Übersicht zur geöffneten Quellcodedatei und Perspektiven (5)

Die wichtigsten der angesprochenen Perspektiven finden Sie als Symbole oben rechts ❺ (Abbildung 1-1) oder im Hauptmenü *Window* (Abbildung 1-2). Für eine vollständige Auflistung öffnet sich ein separates Fenster.

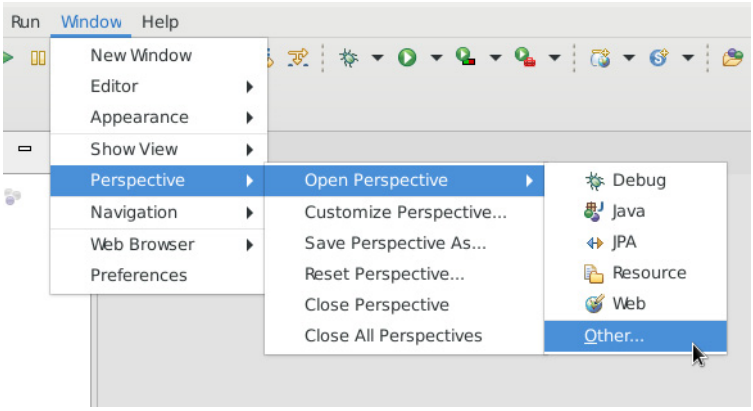


Abbildung 1-2: Das Menü, in dem verschiedene Perspektiven geöffnet werden können, sofern sie noch nicht in der Icon-Leiste rechts oben erscheinen

Grundsätzlich können Sie fast alles in Eclipse nach Ihrem Geschmack und Ihren Erfordernissen konfigurieren, wobei Sie dabei aber das Verhältnis von Aufwand

und Nutzen sowie die üblichen Risiken einer hoch angepassten Umgebung (geringe Vergleichbarkeit, gegebenenfalls Fehlerquelle) nicht außer Acht lassen sollten.

Nun können wir mit der ersten Java-Anwendung starten.

Die erste Java-Anwendung

Für eine neue Anwendung muss in Eclipse zunächst ein neues Java-Projekt erstellt werden: im Menü über *File → New → Java Project* (siehe Abbildung 1-3).

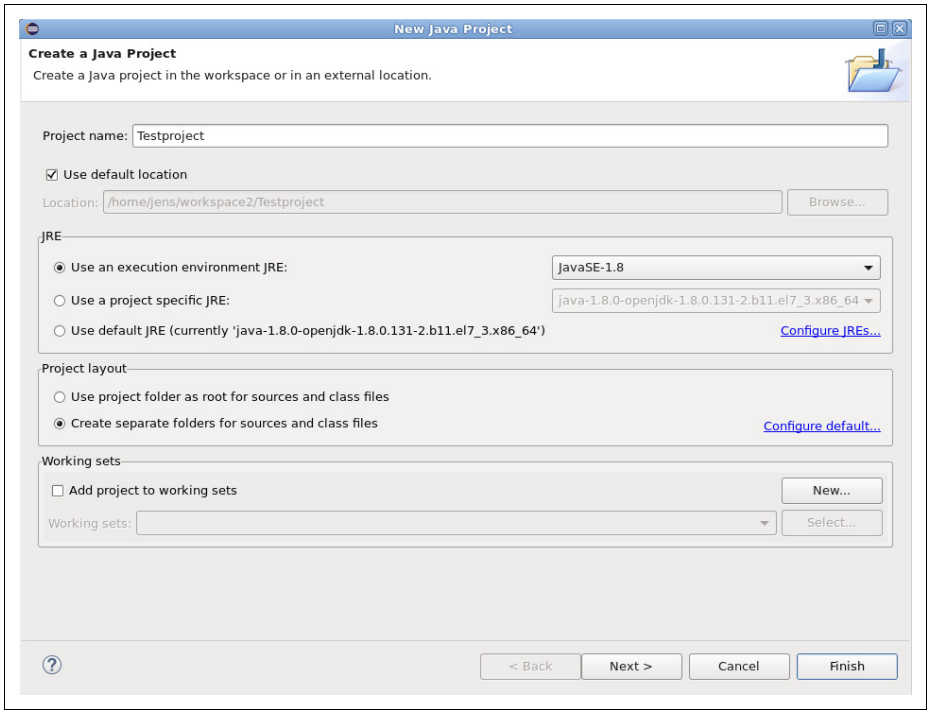


Abbildung 1-3: Ein neues Projekt erzeugen. Hier kann auch eine spezifische Java-Version ausgewählt werden (in diesem Fall wird Java 8 genutzt).

Nach der Eingabe eines Projektnamens reicht ein Klick auf *Finish*. Danach erscheint das neue Projekt bereits im *Package Explorer* auf der linken Seite. Ein Projekt bündelt eine komplette Anwendung innerhalb von Eclipse, ohne ein Projekt kann keine Anwendung ausgeführt werden. Es enthält verschiedene Ordner, zum Beispiel *src*, in dem die Quellcodes gespeichert werden. Weitere Ordner wie *resources* werden eventuell in einem späteren Bearbeitungsstand automatisch hinzugefügt.

Eine neue Java-Klasse kann auf ähnliche Weise erstellt werden: *File → New → Class*. Dann erscheint der entsprechende Dialog (siehe Abbildung 1-4).

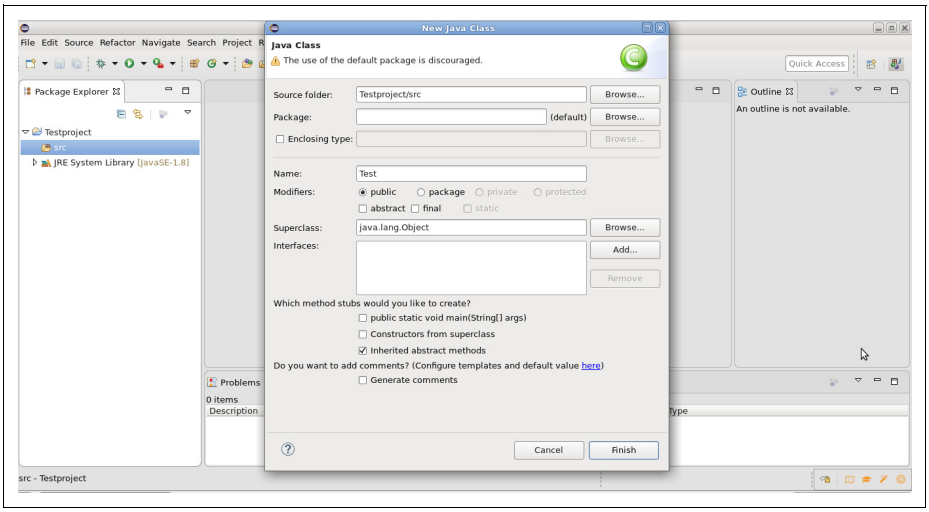


Abbildung 1-4: Eine neue Java-Klasse erstellen

Auch hier wird ein Name benötigt, und ein Klick auf *Finish* beendet den Dialog: In diesem Fall wird die neue *.class*-Datei angelegt.

Wir beginnen mit dieser Java-Klasse:

```
// Test.java
public class Test
{
    public static void main (String[] args)
    {
        System.out.println("Hello World");
    }
}
```

Die erste Zeile beginnt mit `//`, also einem (einzeiligen) Kommentar. Die beiden Schrägstriche können an einer beliebigen Stelle innerhalb einer Zeile beginnen, sodass alles in dieser Zeile rechts von `//` ein Kommentar ist (und damit bei der Ausführung des Programms ignoriert wird).

Ein Kommentar gibt Ihnen die Möglichkeit, dem Quellcode menschenlesbare Notizen und Informationen hinzuzufügen. Dies ist sehr hilfreich, um den Code lesbar zu machen und sich an eventuell getroffene Entscheidungen zu erinnern. Kommentare sind darüber hinaus überaus nützlich, wenn nicht sogar zwingend notwendig, wenn Sie mit mehreren Entwicklern gemeinsam am Code arbeiten. Außerdem können Sie Zeilen des Quellcodes oder ganze Blöcke temporär »deaktivieren«, indem Sie diese einfach in einen Kommentar umwandeln, anstatt sie zu löschen. Das kann z. B. bei der Fehlersuche sehr sinnvoll sein.

Zur blockweisen Kommentierung bietet sich der mehrzeilige Kommentar an: Er wird durch eine Einklammerung zwischen `/*` und `*/` erreicht (Slashs außen, Sterne innen). Kommentare werden in den üblichen Editoren, also auch in Eclipse, farblich abgesetzt (sogenanntes *Code Highlighting*).

Die zweite Zeile zeigt die Definition einer Klasse an. Mit `public` weisen wir darauf hin, dass die folgende Definition öffentlich ist; später werden wir die Bedeutung von `public` noch genauer diskutieren. Anschließend folgt das Schlüsselwort `class`. Schlüsselwörter sind reservierte Wörter, die nicht für andere Zwecke verwendet werden sollten und dürfen, z.B. um Variablen zu benennen. Es kommt der Name der Klasse: `Test`. Die Klassendefinition beginnt und endet mit geschweiften Klammern { }; dieser Bereich wird in der Regel als *Block* bezeichnet. Innerhalb der Klasse finden wir einen weiteren Block, definiert durch eine öffentliche und statische Methode namens `main`, der sogenannten Hauptmethode. Der Ausdruck `public static void main (String[] args)` signalisiert, dass es sich hier um die Haupteingabemethode für eine Klasse handelt. Wenn Sie das Programm ausführen, wird direkt zu dieser Methode gesprungen, und der dortige Code wird ausgeführt. Wir werden die Definition später näher erläutern. Bis jetzt ist es nur entscheidend, dass die folgende Zeile ausgeführt wird:

```
System.out.println("Hello World");
```

Wichtig: Jede Anweisung und jeder Programmbefehl endet mit dem Semikolon `;` (analog zu einigen anderen Sprachen wie SQL). Ein Textwert, eine sogenannte Zeichenkette (*String*), wird in Anführungszeichen " gesetzt.

Bei der Speicherung der Textdatei sollte sichergestellt sein, dass der Dateiname dem Klassennamen entspricht: Der Aufruf der Klasse `public class Test` in einem Stück Quellcode impliziert, dass sich diese Klasse in einer Datei mit dem Namen *Test.java* befindet. Nicht auffindbare Klassen führen dagegen zu Compilerfehlern.

Wir werden nun die Anwendung ausführen. Ein Rechtsklick auf die *Test.java*-Datei im *Package Explorer* öffnet das Kontextmenü. *Run as...* → *Java Application* startet das Programm, und die Ausgabe erscheint in der Konsole (siehe Abbildung 1-5).

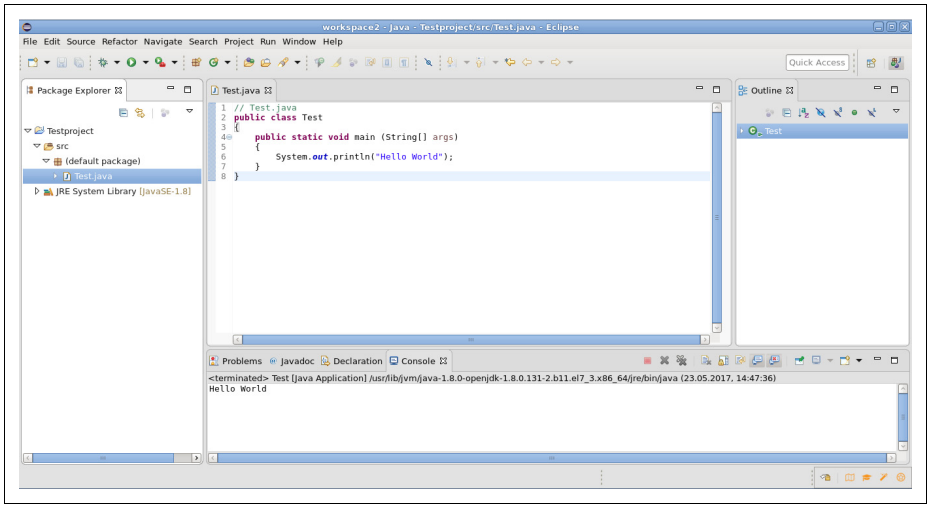


Abbildung 1-5: Die Ausgabe »Hello World«

Alternativ löst *Run → Run* aus dem Menü oder die Tastenkombination Strg+F11 das gleiche Verhalten aus.

Jetzt werden wir uns noch mit der Versionsverwaltung Git und mit Maven beschäftigen, um mit allen wesentlichen Bausteinen dieses Kapitels abzuschließen.

Versionsverwaltung Git

Im Laufe der Arbeit an einer Datei ergeben sich selbstverständlich Veränderungen. Die Eingaben aus dem Editor, die meist auch rückgängig gemacht oder wiederholt werden können, werden dabei mit jedem Speichervorgang fest übernommen. Der alte Stand der physischen Datei wird dabei mit der neuen *Version* überschrieben. In einfachen Tools wird ab hier der Weg zurück versperrt sein – die neue Version ist so die einzig erreichbare, die alte Version ist überschrieben.

Aus diversen Gründen kann es allerdings nützlich oder notwendig sein, auf ältere Entwicklungsstände einer Datei zurückzugreifen. Hier setzen Versionskontrollsysteme (*Version Control Systems,* VCS) an: Sie zeichnen Änderungen an einer Datei oder einem Satz von Dateien im Laufe der Zeit auf, sodass ältere Versionen später wieder aufgerufen werden können. Das ermöglicht es, ausgewählte Dateien oder das gesamte Projekt zurück in einen früheren Zustand der Änderungshistorie zu versetzen. Darüber hinaus können so Änderungen im Laufe der Zeit nachvollzogen, verschiedene Versionsstände verglichen und experimentelle Entwicklungszweige (*Branches*) angelegt werden, ohne einen stabilen Entwicklungsstand zu gefährden. Letztlich ist das auch der Schlüssel zur gemeinsamen Arbeit mehrerer Entwickler: Erfolgreiche, unabhängig programmierte Features oder auch Korrekturen (*Patches*) können kontrolliert und anschließend wieder in das Stammprojekt eingefügt werden (*Merging*). Konkret impliziert die Verwendung eines VCS zumeist, dass Dinge dokumentiert, ausprobiert, vermasselt und verfügbar gemacht werden können – und das mit einem sehr geringen Aufwand.

Git ist ein sehr beliebtes Tool zur Versionsverwaltung. In den meisten (Linux- oder Unix-)Betriebssystemen ist es vorinstalliert. Git hat drei Hauptzustände, in denen sich Dateien befinden können, nämlich *committed*, *modified* und *staged*:

- *Committed* bedeutet, dass der Zustand der Dateien sicher gespeichert ist.
- *Modified* bedeutet, dass eine physische Datei im Dateisystem gegenüber dem in Git abgelegten Zustand geändert wurde, also noch *committed* werden müsste.
- *Staged* bedeutet, dass eine bislang nicht verfolgte Datei neu markiert wird, um sie im nächsten Commit per Git zu sichern.

Dies führt uns zu den drei Hauptinstanzen eines (lokalen) Git-Projekts: dem Git-Verzeichnis (*.git/*), dem Arbeitsbaum und der *Staging Area*:

- Im **Git-Verzeichnis** werden die Metadaten und die Objektdatenbank für Ihr Projekt gespeichert, es ist also das Langzeitgedächtnis des VCS. Dies ist der

wichtigste Teil von Git, und es ist das, was kopiert wird, wenn Sie ein Repository von einem anderen Computer klonen.

- Der **Arbeitsbaum** ist ein einzelner *Check-out*, also eine Version des Projekts. Anders ausgedrückt, ist der Arbeitsbaum das physische Abbild einer Projektversion aus dem gesamten »Versionsraum« des Git-Repositories: Er ist Ihr Arbeitsverzeichnis. Die zugehörigen Dateien werden aus der komprimierten Datenbank im Git-Verzeichnis geladen und auf der Festplatte abgelegt, damit sie von jedem anderen Programm verwendet oder geändert werden können.
- Die **Staging Area** ist eine Datei, die sich normalerweise im Git-Verzeichnis befindet und Informationen darüber speichert, was in Ihren nächsten Commit einfließt – also eine Art Kurzzeitgedächtnis bzw. To-do-Liste. Der technische Name im Git-Sprachgebrauch ist *Index*, aber der Begriff »Staging Area« funktioniert genauso gut.

Ein Repository kann lokal auf zwei Arten angelegt werden: vollständig neu (`git init`) oder durch Kopieren einer gegebenen Quelle (*Klonen*, `git clone PATH2REPO`). Ab diesem Punkt ist das oben erwähnte Git-Verzeichnis *.git* vorhanden, das *Programm* `git` kann damit dann arbeiten.

Der grundlegende Git-Workflow für den Entwickler sieht in etwa so aus:

- Sie ändern Dateien in Ihrem Arbeitsbaum.
- Sie stellen durch *Staging* selektiv nur jene Änderungen bereit, die beim nächsten Commit berücksichtigt werden sollen (`git add DATEI`).
- Sie führen einen Commit aus, der jene Dateien übernimmt, die sich im Staging-Bereich befinden (`git commit`).

Wenn die Datei modifiziert und der Staging Area mittels `git add DATEI` hinzugefügt wurde, wird sie für den nächsten Commit bereitgestellt – dadurch werden nur definierte Änderungen im Index gelistet bzw. im Kurzzeitgedächtnis vorgemerkt. Befindet sich eine bestimmte Version einer Datei im Git-Verzeichnis, gilt sie als *committed*. Dieser sogenannte *Snapshot* wurde also dauerhaft in der Git-Versionshistorie gespeichert (im Langzeitgedächtnis abgelegt).

Im Arbeitsverzeichnis kann sich jede Datei in einem von zwei Zuständen befinden: verfolgt (*tracked*) oder nicht verfolgt (*untracked*). Verfolgte Dateien sind Dateien, die sich im letzten Snapshot befanden; sie können unveränderlich (geschützt), geändert (*modified*) und *staged* sein. Kurz gesagt: Verfolgte Dateien sind Dateien, von denen Git weiß und deren Zustand überwacht wird. Unmittelbar nach dem erstmaligen Klonen eines Repositories sind alle heruntergeladenen Dateien verfolgt (und bis zur ersten Bearbeitung unverändert). Nicht verfolgte Dateien sind alle anderen – alle Dateien im Arbeitsverzeichnis, die sich nicht im letzten Snapshot und nicht im Staging-Bereich befinden. Sie können entsprechend per Staging und Commit der Historie hinzugefügt werden, müssen es aber nicht. Das gilt z. B. für Binärdateien (Git ist primär auf Textdateien ausgelegt), Logs, Systemdateien und