Philip Ackermann

Inkl. Downloads

Mit Syntax-Highlighting!

Schrödinger programmiert Java

Das etwas andere Fachbuch

- Von den Sprachgrundlagen über Swing und JavaFX bis zur komplexen Anwendung
- Ob Module, Lambdas oder New File I/O: Nutze die Schwerter aller Versionen!
- Durchblicken, mitmachen und genießen!

DRITTE AUFLAGE

Liebe(r) Leser(in),

Sie haben gewählt:

Java

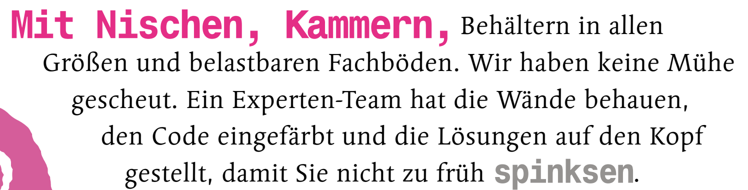
Wir gratulieren – und bedienen Sie gern. Wo soll's denn hin? **Ins Hirn?** – Gerne doch. Die Windungen sehen gut aus, da lassen sich ordentlich Fachkenntnisse unterbringen.

JA, BITTE.
AM LIEBSTEN IM SCHLAF. ODER EINFACH MIT'NEM TRICHTER...

Apropos unterbringen: Haben Sie schonmal versucht, eine Vorratskammer zu füllen, die ganz glatte Wände hat? – Geht nicht. Nicht ohne Regale jedenfalls. Bretter, Nischen, Schüsseln – damit geht's.

Was diesen **Trichter** angeht, da müssen wir leider **passen**. Aber seien Sie nicht allzu enttäuscht, denn Sie halten mehr als einen Ersatz in den Händen:

Das etwas andere Fachbuch



Mit Nischen, Kammern, Behältern in allen Größen und belastbaren Fachböden. Wir haben keine Mühe gescheut. Ein Experten-Team hat die Wände behauen, den Code eingefärbt und die Lösungen auf den Kopf gestellt, damit Sie nicht zu früh **spinksen**. Und wir bringen Sie mit Schrödinger zusammen. Nein, auch der nimmt Ihnen das Lernen nicht ab. Ehrlich gesagt, denkt er nicht einmal daran.

Gut, dass ihr das gleich klargestellt.

Das wäre auch zu schade. Sie würden nämlich fantastische Übungen verpassen, wie den Solche-Schuhe-hast-du-schon-Tester.

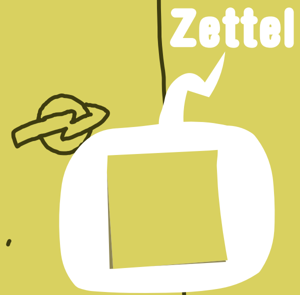
Ach, der! Als ginge es hier nur um Schuhe. Darf ich auch mal was sagen? Hier geht es um Stil, aufgepasst, Spaghetti-Programmierung ist hier nicht drin. Bis gleich, wir sehen uns in der Werkstatt …

Na dann, bitte sehr. **Es ist angerichtet.**

Viel Spaß!

Schrödingers Büro

Hintergrund info



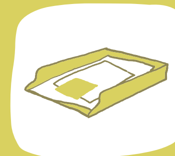
Zettel

Achtung

Belohnung/Lösung

Die nötige Theorie,
viele Hinweise und Tipps

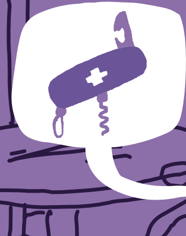
Falscher Code X



Schrödingers Werkstatt

Erledigt!/Belohnung

Fehler


Einfache Aufgabe

Schwierige Aufgabe

Unmengen von Code, der ergänzt, verbessert und repariert werden will

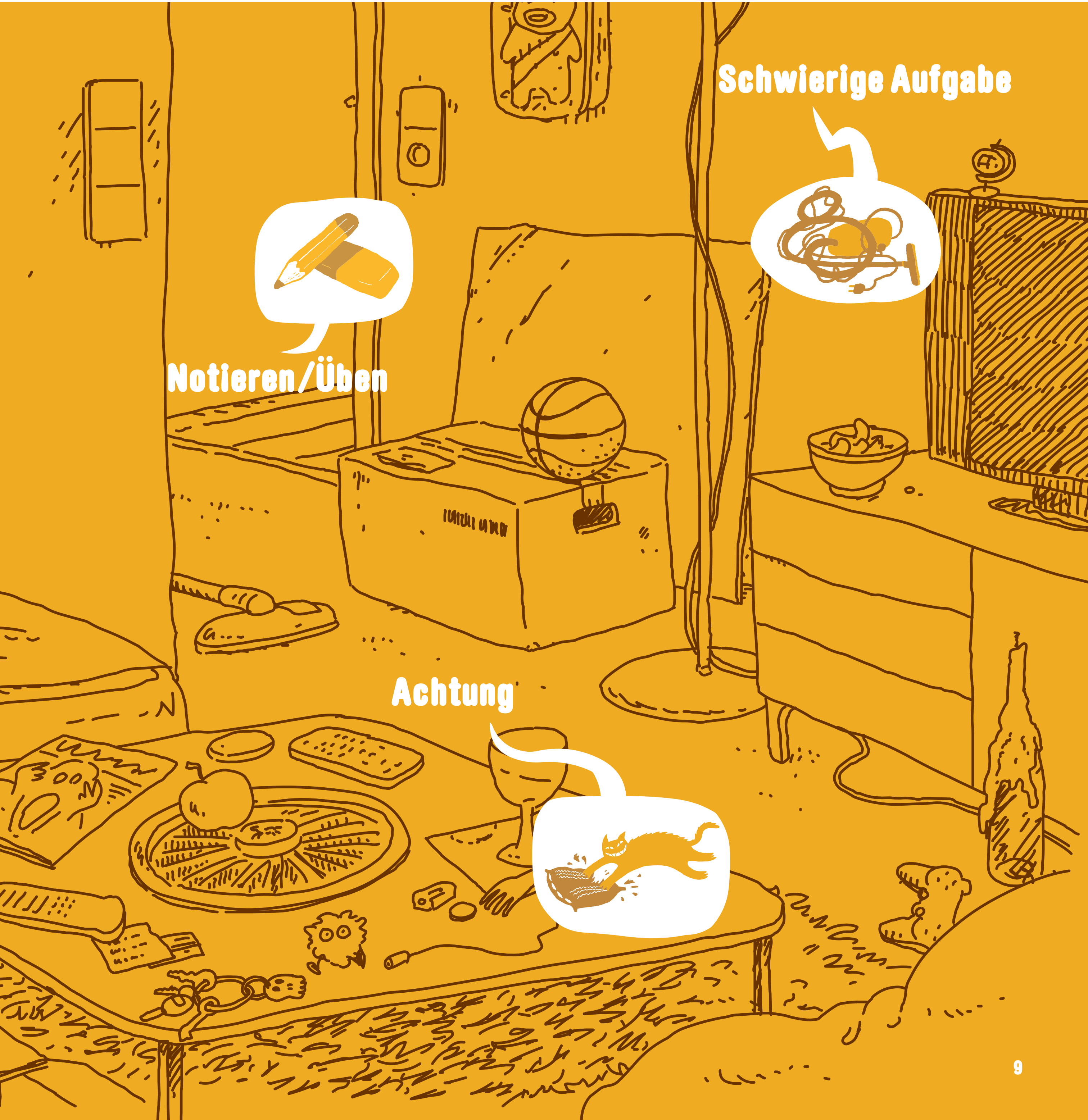
Schrödingers Wohnzimmer

Zettel

Belohnung/Lösung

Einfache Aufgabe

Viel Kaffee, Übungen und die verdienten Pausen



Schrödinger programmiert Java

Als Kanutin umschifft Almut Stromschnellen mit notfalls nur einem Paddel. Eine Fähigkeit, die sie zur Fachbuchlektorin prädestiniert.

Almut Poll, LEKTORAT

Janina „Sherlock" Brönner. Bei der Buchherstellung sind detektivische Kombinationsgabe und Finesse gefragt. Die Kollegen haben sich allerdings das Pfeiferauchen verbeten.

Janina Brönner, HERSTELLUNG

Schon zu Schulzeiten zeichnete Leo am liebsten die Bücher voll, von denen er am wenigsten kapierte.

Seit er weiß, dass man das auch gegen Bezahlung machen kann, kapiert er gar nichts mehr.

Leo Leowald lebt und arbeitet in Köln als freiberuflicher Illustrator. Er veröffentlicht unter anderem in *titanic*, *jungle world* und bei *reprodukt* und zeichnet seit 2004 den Webcomic *www.zwarwald.de*.

Andreas' zweite Leidenschaft neben der Buchgestaltung ist kochen. Wie auch immer: Hauptsache rare – VERY RARE!

Andreas Tetzlaff ist selbstständiger Buchgestalter in Köln. Er arbeitet normalerweise für Kunstbuchverlage (zusammen mit seiner Frau als *probsteibooks.de*) – dass ausgerechnet ein IT-Fachbuch ihn vor künstlerische Herausforderungen stellt, hätte er sich vorher nicht träumen lassen …

Annette ist von Haus aus Archäologin, da ist es nur ein kleiner Schritt bis zum Lektorat, und der Vorteil ist: Bei Schrödinger und Co. findet sie immer was.

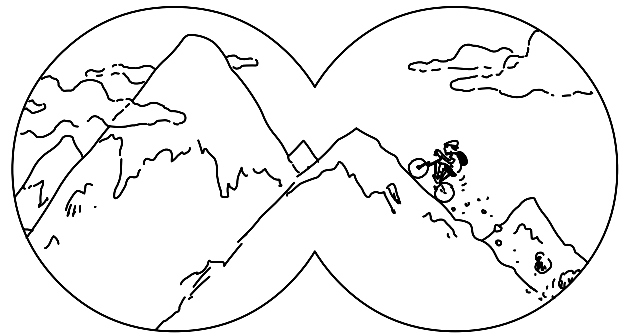
Annette Lennartz ist freiberufliche Lektorin in Bonn. Für Schrödinger hat sie immer eine offene Tür. Privat schätzt sie augenzwinkernde und gruselige Geschichten oder bastelt an filigranen Schiffsmodellen.

KORRIGIERT VON: *Annette Lennartz*

Glücklicherweise ist ein Schrödinger-Buch keine Bleiwüste, allenfalls ein Textgebirge. Trotzdem: Als Gutachter muss man da den Überblick behalten. Aber du willst nicht wirklich mit dem Fahrrad da hoch, oder, Christoph? Christoph???

Als Softwareentwickler beschäftigt sich *Christoph Höller* mit Java-Enterprise-Architekturen, modelliert Business-Prozesse und … ach was, das steht auf *www.christoph-hoeller.net*

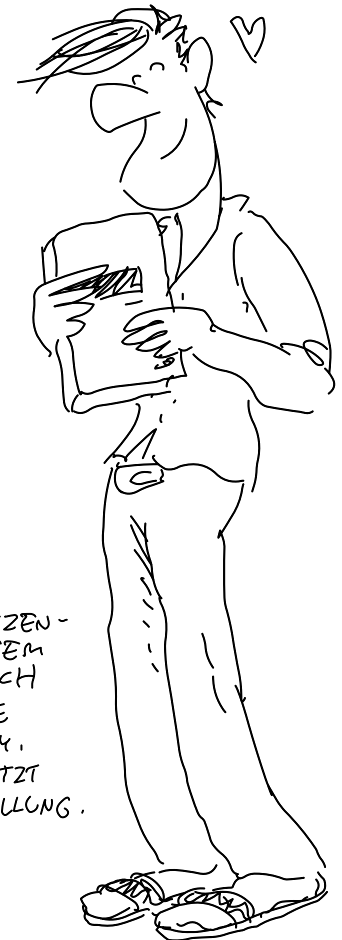
BEGUTACHTET VON: *Christoph Höller*



Ausgleichssport für Philip sind seine Inlineskates und seine Vinylplattensammlung. Je nachdem, wie gut es bei Schrödinger gerade läuft, rotiert er mit 33, 45 oder 78 Umdrehungen pro Minute.

Als Software-Entwickler wechselt *Philip* mühelos die Spur, von objektorientiert zu funktional, von Java zu Node.js und zurück. Kein Wunder, dass er auch Schrödinger mit CSS, XML, SQL und noch mehr Sprachen nicht verschont.

- *Philip Ackermann*, AUTOR



MIT TALENT, KATZENPHOBIE UND LÄSSIGEM SCHUHWERK BESTACH SCHRÖDINGER DIE RHEINWERK-JURY. FÜR IHN GEHT JETZT EIN TRAUM IN ERFÜLLUNG.

Buch- und Einbandgestaltung:
Andreas Tetzlaff und Leo Leowald

FÜR DIE, DIE ES GENAU WISSEN WOLLEN

Dieses Buch wurde gesetzt aus unzähligen Schriften (u.a. aus der WIMBY von Evert Ypma: Danke, Evert!), Tonnen an Illustrationen und anderen komischen Zeichen, die alle Beteiligten in den Wahnsinn trieben.

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 978-3-8362-7274-2

© Rheinwerk Verlag GmbH, Bonn 2020
3., aktualisierte und erweiterte Auflage 2020

Das vorliegende Werk ist in all seinen Teilen urheberrechtlich geschützt. Alle Rechte vorbehalten, insbesondere das Recht der Übersetzung, des Vortrags, der Reproduktion, der Vervielfältigung auf fotomechanischen oder anderen Wegen und der Speicherung in elektronischen Medien. Ungeachtet der Sorgfalt, die auf die Erstellung von Text, Abbildungen und Programmen verwendet wurde, können weder Verlag noch Autor, Herausgeber oder Übersetzer für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen. Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

INHALTSVERZEICHNIS

Vorwort 21

Kapitel 1: Hallo Schrödinger

Einführung und erstes Programm

Seite 23

Java überall 24
JRE, JDK, SE, EE, ME 24
Java installieren 25
Hallo Schrödinger 26
Kompilieren von Hand 27
Programm starten 30
Compiler und JVM unter der Lupe 31
Rätselstunde 32
Hexenstunde 33
Entwicklungsumgebungen 35
Eclipse herunterladen 35
Workspace und Workbench 36
Erstes Projekt 38
Pakete packen, aber ordentlich 41
Pakete anlegen leicht gemacht 43
Neue Klasse mit Eclipse 44
Miteinander reden 47
Streams 50
Let the game begin – das „World of Warcraft Textadventure" 52
Historie von Java 57

Kapitel 2: Alles eine Typfrage

Variablen und grundlegende Datentypen

Seite 61

Variablen und Datentypen 62
Arten von Datentypen 63
Können wir Zahlen, bitte? 64
Lesbarkeit von langen Zahlen 67
Zahlensuppe 68
Binär, oktal, dezimal und hexadezimal 69
Variablennamen 71
Gute Namenswahl – Java Code Conventions 73
Operationen 75
Dividieren und komplizieren 76
Genau rechnen mit BigDecimal 77
Ein bisschen mehr, nein, eher ein bisschen weniger 78
Rechenaufgaben 79
Konvertierung von Zahlen 80
Zeichen und Wunder 82
Ein Charakter namens Cäsar 84
Nichts als die Wahrheit 86
Vergleichsoperatoren 88
Sprich die Wahrheit 89

Wrapperklassen 91
Alles eine Typfrage 94
Es kommt auf den Charakter an 95
Wrap-up 96

Kapitel 3: Hier war ich doch schon mal!

Kontrollfluss

Seite 99
Kontrolle übernehmen 100
Wenn das Wörtchen „if“ nicht wär 100
Halb voll oder halb leer 101
Entweder oder 102
Aberglauben 103
Boolesche Operatoren 105
Hör auf, so zu klammern 105
Kurzschreibweise 106
Bedingungsoperator 107
Mehrere Fälle behandeln 107
Leere Lehre 109
Switch the String 110
Realitätsfern 111
Schleifen drehen 114
Schachtelung 117
Schleifen binden lernen 118
Primzahlen 118
King Schrödinger 122
Nochmal gaaanz langsam, bitte 125
Schleifen abbrechen 128
Labels 129
Kalender 130
Zusammenfassung – alles unter Kontrolle 134

Kapitel 4: Von Dinkelpfannekuchen und Buchstabensalat

Arbeiten mit Strings

Seite 135
Strings definieren 136
Auf einzelne Zeichen zugreifen 138
Strings zerlegen 139
Zeichen für Zeichen 142
Vergleichen von Strings 145
Teile vergleichen 145
Gleich oder nicht gleich, oder doch teilweise gleich 147
Suchen und ersetzen in Strings 149
Speiseplanhacker 151
Summertime 153
Noch was für Profis: Strings in den Pool werfen 157
Strings sind unveränderlich 161
Veränderbare Zeichenketten 162
Zeichen löschen, ersetzen und spiegeln 164
StringBuilder in Schleifen 166
Was du aus diesem Kapitel mitnehmen solltest 167

Kapitel 5: Objektiv betrachtet, eine ganz andere Klasse

Klassen, Objekte, Methoden

Seite 169

Der nächste Level der Wiederverwendung – Methoden ... 170
Refactoring ... 174
Sauberer Quelltext ... 178
Methodenkommentare ... 179
Kann ich das zurückgeben? ... 180
Zurück zu den Nachrichten ... 181
Auf dem Weg zum Java-Ninja – Klassen und Objekte ... 182
Kapselung ... 186
Setter und Getter ... 188
Der Fotoapparat unter der Lupe ... 191
Setter mit mehreren Parametern ... 192
Klassen vs. Objekte ... 194
Sichtbarkeit von Variablen und Methoden ... 196
Konstruktoren ... 198
Objekte in Strings umwandeln ... 202
Quelltext anhängen ... 203
Textuelle Beschreibung von Objekten ... 204
Fluent Interface ... 205
Nur für Profis – Zerlegen des Fotoapparats ... 207
Referenzdatentypen, Heap, Garbage-Collection ... 211
Wenn der Heap zu voll wird ... 213
Destruktoren … ... 214
Hab lieb den Heap ... 215
Zuschauen, wie der Heap wächst ... 216
Der Stack ... 222
Wie alles zusammenhängt ... 224
Die Katze war's ... 226
Steap oder Hack ... 228

Kapitel 6: Woher hat er das bloß?

Vererbung

Seite 231

Vererbung ... 232
Noch weniger doppelter Code dank Vererbung ... 234
Der Apfel fällt nicht weit vom Stamm ... 236
Verhalten anpassen durch Überschreiben von Methoden ... 237
Überschreiben verboten ... 239
Blümchenvererbung ... 240
Ich will aber zu Opa und Oma! Was ist mit der Großelternklasse? ... 242
Vererbung und Konstruktoren ... 244
Die Konstruktorkette wieder reparieren ... 245
Wie war das nochmal mit den Konstruktoren? ... 248
Erbt der Sohn vom Vater oder der Vater vom Sohn? ... 249
Typumwandlungen ... 252
Upcasting ... 253
Den Typ überprüfen ... 254
Welche Methode hätten's denn gerne? Methoden überladen ... 255
Abo-Service ... 256
Serviceorientiert vs. objektorientiert ... 258
Zurück zu den Objekten ... 259
Die volle Überladung ... 260
Deutschland sucht den Knetsuperstar ... 261
Zusammenfassung ... 263

Kapitel 7: Schnittstellen und andere leichte Verletzungen

Abstrakte Klassen und Interfaces

Seite 265

Abstrakte Klassen 266
Abstrakte Methoden 268
Digital oder analog? Hauptsache abstrakt! 270
Alles abstrakt, oder was? 278
Schnittstellen 279
Übungen zu Interfaces 284
Für die einen sind es Interfaces, … 286
Interfaces und Vererbung 287
Weniger ist manchmal mehr 289
Konstanten in Interfaces 290
Konstanten in Konstantenklassen 290
Statischer Import 291
Statische Blöcke 292
Konstanten über Enums 293
Interfaces – Wiederholung 294
Noch mehr Klassen 295
Statische Memberklassen 295
Nichtstatische Memberklassen 297
Lokale Klassen 298
Anonyme Klassen 299
Abstrakt und polymorph – alles klar, Schrödinger? 301

Kapitel 8: Hast du eigentlich einen Überblick über deine ganzen Schuhe?

Arrays, Collections & Maps

Seite 303

Ich will mehr davon – Beziehungen zu Katzen und andere Freundinnen 304
Objekte der Begierde 306
Ab in die nächste Dimension 307
Gib mir alles 308
Solche Schuhe hast du schon! 309
Alles muss man selber machen 310
Auf den Spuren der Objektgleichheit 312
Gleich ist gleich – oder doch nicht? 313
Arrays wachsen nicht mit 313
Schuhkollektionen 315
Solche Schuhe hast du immer noch! 317
Soll ich dir mal meine Schuhsammlung zeigen? 319
Der Iterator (hört sich krass an, oder?) 320
Link und listig 321
Einmal heißt einmal und dann keinmal mehr 322
Hash mich, ich bin der Frühling 323
Sets für die Kripo 324
Sets für den Schuhtester 327
Der richtige Hashcode 328
Bäume und Räume 331
Größer oder kleiner? 333
Sortieren ohne Bäume 336
Sortierte Schuhe 338
Mappen raus 339
So viele Schuhe hast du schon 340
Schlangen im Keller 343

Hinten anstellen! 344
Kompakt und funktional: Lambda-Ausdrücke 345
filter(), map() und reduce() 347
High Heels!!! 351
Nochmal alles ganz lambsam 353

Kapitel 9: Ausnahmsweise und um ganz sicher zu gehen

Exceptions

Seite 355
Aus Fehlern werden Ausnahmen 356
Deine erste Exception 357
Das Werfen vorbereiten 358
Fangen will gelernt sein 360
Ganz zum Schluss – der finally-Block 362
Hier werden Sie individuell behandelt 363
Vererbung von Exceptions 365
Die Ausnahme als Auskunftsobjekt – was ist eigentlich passiert? 366
Information erwünscht, Abhängigkeit unerwünscht 368
Exceptions weiterwerfen 369
Wann und wie behandeln 370
Exceptions immer weiterleiten? Nicht immer der richtige Weg 371
Muss man nicht fangen – Unchecked Exceptions 373
Defensiv programmieren gegen Runtime-Exceptions 376
Exceptions loggen 377
Lass den Türsteher mitloggen 378
Nicht mehr zu retten 380
Speicher voll 382
Automatisches Schließen von Ressourcen 383
Dateien lesen 384
Ausnahmen bestätigen die Regel 386

Kapitel 10: Ey Typ, du kummst hier nit rein!

Generics

Seite 389
Generische Typen 390
Bevor es Generics gab … 391
… und mit Generics 392
Hunde vs. Katzen 394
Katzenkorb als Unterklasse 395
Typisierte Interfaces 396
Wildcard-Typen – das Problem 397
Der nach oben beschränkte Wildcard-Typ 400
Der Haken: Schrödinger darf nicht schreiben 401
Der nach unten beschränkte Wildcard-Typ 404
Typisierte Methoden 406
Übungen zu Wildcards 407
Wiederholung 410
Kaffeesatz 413

Kapitel 11: Wilde Ströme – Eingabe und Ausgabe

Dateien, Streams und Serialisierung

Seite 415

Bossingen kommt mit Dateien **416**
Willst du mehr? Probier's binär! **417**
Binärdateien schreiben – auch mit Byte-Streams **419**
Megalangsam – Dateien kopieren mit normalen Streams **420**
Viel schneller – Dateien kopieren mit Buffered Streams **421**
Wer liest schon Bytes? Textdateien lesen mit Character-Streams **422**
Textdateien schreiben mit Character-Streams **423**
1:0 für den CSV – Textdateien umwandeln **424**
Mit Kanonen auf Verzeichnisse schießen **427**
Endlich Ordnung – Dateien und Verzeichnisse erstellen **428**
Kundendaten konvertieren am Fließband **430**
Gut gefiltert ist halb gewonnen – Verzeichnisse filtern **432**
Auf dem richtigen Pfad: Die neue File-IO-API **433**
Kundendateien konvertieren – jetzt noch einfacher **436**
Objekte speichern **438**
Geschachtelte Objekte speichern **441**
Serialisierung und Deserialisierung beeinflussen **443**
Individuelle Serialisierung mit writeObject() und readObject() **444**
Der Nusskopf und die Kopfnuss **446**
Nochmal alles zum mitstreamen **448**

Kapitel 12: Nicht den Faden verlieren

Programmierung mit Threads

Seite 449

Prozesse und Threads **450**
Der erste Thread **452**
Night of the living thread **453**
Das war gerade noch ungerade **455**
Da krieg ich Zustände **457**
Threads schlafen legen **458**
Helden, aufgepasst! **459**
Auf andere warten **462**
Synchronisierung **463**
Erst die geraden Zahlen, bitte! **466**
… Deadlocks! **467**
Deadlocks finden und umgehen **469**
Der Schlüssel zum Erfolg **471**
Livelock, Starvation, Priorisierung und ein Bier **473**
… Livelock **473**
Prioritäten setzen **476**
Warten und benachrichtigen **478**
Starvation **480**
Warten und schlafen **482**
4000 gegen 2 oder Arbeit verteilen **483**
Die Zusammenfassung, damit du nicht den Faden verlierst **487**

Kapitel 13: Das kann sich doch schon sehen lassen!

Deployment, Dokumentation und Module

Seite 489

Abgepackt ... **490**
Ich packe meine JAR-Datei … ... **493**
Java Web Start ... **495**
Bibliotheken einbinden in Eclipse ... **497**
Hallo Onkel Doc – Dokumentation mit javadoc ... **500**
Strukturiert und modularisiert ... **503**
Modular? Na klar! ... **507**
Module kompilieren ... **509**
Jetzt hast du's gepackt ... **511**

Kapitel 14: Austauschschüler – das Datenaustauschformat XML

XML

Seite 513

XML ... **514**
Russische Salami ... **520**
Musikschule mit XML ... **525**
Ist es ein Element oder ein Attribut? ... **527**
XML lesen ... **528**
Der Spürhund – Simple API for XML ... **528**
Der trainierte Spürhund oder „Don't call me, I'll call you“ – StAX ... **532**
Die Cursor-API ... **533**
Die Iterator-API ... **533**
Das Document Object Model … ... **535**
Flower Power ... **538**
XML schreiben ... **540**
1, 2 oder 3? SAX, StAX oder DOM?
Was nehme ich wann? ... **543**
Viele Wege führen nach Java ... **545**
JAXB ... **547**
Von XML nach Java und wieder zurück ... **549**
Power Flower ... **551**
Die XTra, XPlizite, Xakte, XOrbitante
Zusammenfassung ... **555**

Kapitel 15: Datenspeicherung mit JDBC

Datenbanken

Seite 557

Relationale Datenbanken ... **558**
Die erste Tabelle mit SQL ... **559**
Eine Verbindung herstellen ... **563**
Datensätze hinzufügen ... **565**
Helden und Briefmarken ... **567**
Anweisungen zusammenfassen ... **570**
Gut vorbereitet ... **571**
Daten auslesen ... **572**

Was hätten'S denn gerne? Datensätze nach Kriterien auswählen ... 574
Daten sortiert ausgeben ... 576
Wer ist der Stärkste im ganzen Land? ... 577
Informationen aus mehreren Tabellen über Joins verknüpfen ... 581
Alles wieder zurück bitte – Transaktionen ... 582
Mapping zwischen relationalen Datenbanken und Java-Objekten ... 584
Roll zurück den Troll ... 585
Wiederholung ... 588
SELECT DAS_WICHTIGSTE FROM KAPITEL_15 ... 589

Kapitel 16: Neue Tanzschritte

GUI-Programmierung mit Swing und JavaFX

Seite 591
Mit Schwung weg von der Konsole – Swing ... 592
Alles im Rahmen – wie du GUI-Komponenten erstellst ... 592
Alles in Reih und Glied – wie du GUI-Komponenten anordnen kannst ... 595
Alles im Raster mit dem Grid-Layout ... 598
Zellen verbinden mit dem Grid-Bag-Layout ... 600
Schuhe in der Box … oder doch besser im Grid? ... 604
Ordnung: gut, Verhalten: … nichts? Wie du GUI-Komponenten das Verhalten hinzufügst ... 607
Validierung von Nutzereingaben mit Swing ... 611
Schuhschiebereien ... 612
Bunter und mehr Action mit JavaFX ... 616
Auf die große Bühne – JavaFX on stage ... 619
Noch mehr Zucker mit CSS ... 624
Das Verhalten hinzufügen ... 626
Validierung von Nutzereingaben mit JavaFX ... 627
Schiebereien mit JavaFX ... 628
Für Profis – Verhalten hinzufügen ohne Listener ... 629
Schieberegler mit Verhalten ... 631
Die Lösung für den Hardcore-Profi – Binding kann auch rechnen ... 632
Das Wichtigste zur Gesellenprüfung Maler und Lackierer, Fachbereich Swing und JavaFX ... 635

Kapitel 17: Schrödinger goes international

Internationalisierung, Lokalisierung, Formatierung, Pattern Matching und reguläre Ausdrücke

Seite 637
Den Nutzer lokalisieren ... 638
Die Software internationalisieren ... 641
Finnische Schuhkartons ... 643
Speak english per favore ... 646
Klassen laden ... 647
Wein oder Bier? ... 649
Zahlen und Währungen formatieren ... 651
Datums- und Zeitangaben formatieren ... 652
Währungsrechner ... 654
Internationalisierte Textmeldungen formatieren ... 656
Das wurde aber auch Zeit – java.time ... 657
Zeitzone, Zeitlinie, Zeitpunkte und Zeitdauern ... 658

Ohne Zeitlinie – Datum, Zeit und Zeiträume 660
Zeitzonen und Zeitunterschiede 662
Alles nochmal auf einen Blick, so viel Zeit muss sein 663
Formatierung von Datums- und Zeitangaben 664
Formatieren über die Klasse String 665
Mit Format 667
Pattern Matching mit regulären Ausdrücken 668
Kein Anschluss unter dieser Nummer 673
Teile finden 673
Drin oder nicht drin? Keine Frage! 675
Auf einzelne Teile zugreifen über Gruppen 676
E-Mail-Adressen validieren 677
Tiivistelmä pääkohdista – das war Finnisch 679

Kapitel 18: Bist du dir da sicher? Unit-Testen und Java Web Start continued

Unit-Tests

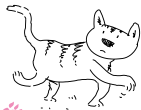
Seite 681

Deine Klassen testen 682
Test-Frameworks 683
Ich hatte aber einen Cocktail bestellt 690
Ausnahmefälle testen 694
Gar nicht eklig 696
Java Web Start continued 698
Das wichtigste unit to know 701

Index 705

Wir können ruhig du sagen – mache ich mit Schrödinger ja auch

Ich nehme mal an, du hältst dieses Buch in den Händen, weil du genau wie Schrödinger schnell und trotzdem gründlich Java lernen und deine eigenen Anwendungen schreiben möchtest. Dann stehst du wahrscheinlich genau da, wo Schrödinger vor etwa einem Jahr stand und ich vor mehr als 10 Jahren zu Beginn meiner Karriere als Software-Entwickler auch: vor einer schier unüberwindbaren Fülle von Informationen, oft dicken langweiligen Fachbüchern und dem Gefühl, nicht zu wissen, wo man anfangen soll. Tja, was soll ich sagen: Ich wurde damals mehr oder weniger ins kalte Wasser geworfen. Das tat rückblickend betrachtet eigentlich richtig gut, aber ganz so gemein war ich dann zu Schrödinger doch nicht. Denn nur für den Fall, dass du ihn noch nicht kennen solltest: Der wäre damit nämlich garantiert alles andere als einverstanden gewesen.



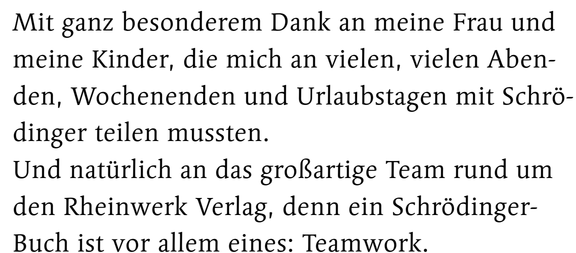
Keine Sorge also: Mit diesem Buch zeige ich dir Schritt für Schritt, wie du mit Java Software entwickeln kannst. Die Sprachmittel zu lernen ist dabei die eine Sache. Die Sprache richtig anzuwenden und zu wissen, was gut und was schlecht ist, das ist die Kunst. Unter anderem nämlich, wie du richtig objektorientiert programmierst und was der Unterschied zu serviceorientiertem oder gar funktionalem Programmieren ist, wie du deinen Code sauber hältst, wie du ihn testen kannst und vieles mehr. Natürlich reicht selbst so ein dicker Schinken wie dieses Buch nicht aus, dir alles rund um Java und die Kunst des Programmierens zu erklären. Deswegen habe ich ja auch das Wichtigste für dich herausgefiltert.

Nichtsdestotrotz: Java zu lernen, das braucht schon ein bisschen Zeit. Vor allem geht es nicht, wenn du dir nicht auch selbst die Hände schmutzig machst. Eine neue Sprache lernst du ja auch nur, indem du sie sprichst und nicht, indem du nur die Vokabeln lernst. Daher habe ich auch jede Menge Übungen und Codebeispiele vorbereitet. Und damit sich keiner die Finger wund tippen muss, gibt's alle Listings auf der Bonusseite zum Buch unter www.rheinwerk-verlag.de/4975.

Außerdem lernst du hier nicht nur Java, sondern bekommst auch einen Einblick in andere Technologien wie XML, CSS oder SQL. Ein guter Entwickler verfügt nämlich nicht nur über Tiefenwissen, sondern auch über Breitenwissen. Und dazu solltest du mehr als einmal über den Tellerrand gucken. Ich weiß, ich verlange an manchen Stellen wirklich viel und die Lernkurve ist teilweise schon etwas steiler. Aber keine Sorge, ich bin ja bei dir.

Eine Sache noch: Wenn du einmal mit Java anfängst, wirst du nicht mehr aufhören können. Das gilt auch für das Lernen. Denn auch Java entwickelt sich weiter. Während ich damals vor der ersten Auflage mit Schrödinger zusammensaß, wurde zum Beispiel bei Oracle gerade der Java-SE-8-Standard festgezurrt. Na, dachte ich mir, dann konnte ich ihm den ja auch gleich beibringen. Ich muss sagen, ich bin schon ziemlich stolz auf ihn. Die Lambda-Ausdrücke haben wir gleich auf seine Schuhkarton-Sammlung angewendet. Und für Datums- und Zeitangaben kennt er jetzt auch die neue, viel bessere Date-Time-API. Alle Themen im Buch werden übrigens für die Versionen Java 7 bis Java 12 behandelt, und wenn du eine ältere Version hast, lasse ich dich auch nicht im Stich. Aber jetzt erst mal ab ins kalte Wasser.

Widmung



Einführung und erstes Programm

— EINS —

Hallo Schrödinger

Schrödinger findet heraus,
dass Java-Programme innerhalb einer virtuellen
Maschine laufen. Um sie jedoch da hineinzubekommen,
muss er sie erstmal als Bytecode kompilieren.
Das geht für ein simples Programm auch relativ einfach,
aber Schrödinger will ja keine simplen Programme
schreiben. Deswegen hat er sich auch gleich
eine Entwicklungsumgebung installiert, mit der das
Kompilieren noch einfacher von der Hand geht.
Schrödinger schreibt damit sein erstes Java-Programm,
lernt den Aufbau einer Java-Datei und wie er Daten an
sein Programm sendet und wieder herausholt.

Java überall

Prima, Schrödinger, du willst also Java lernen?

Freut mich, dass du mich gefragt hast, dir dabei zu helfen. Tja, wo fangen wir an?

Am besten direkt loslegen. Ich bin nämlich nicht so der Typ für Smalltalk.

Okay, okay,

aber bevor wir loslegen können, musst du dir erst noch ein paar Dinge besorgen. Keine Sorge: Es ist alles kostenlos.

JRE, JDK, SE, EE, ME

Das Erste, das du brauchst, ist die Java-**Laufzeitumgebung**. Java-Programme laufen nämlich in einer sogenannten **virtuellen Maschine**, der **JVM** (Java Virtual Machine oder, wenn du ganz cool sein möchtest, „Dschej Wi Em"), und nicht wie C-Programme direkt auf dem Betriebssystem. Diese Laufzeitumgebung heißt auch **JRE** (Java Runtime Environment).

Da du aber – sonst wärst du ja sicherlich nicht hier – Java-Programme nicht nur **laufen** lassen, sondern auch **entwickeln** möchtest, benötigst du außerdem noch das sogenannte **JDK** (Java Development Kit). Sowohl JRE als auch JDK kannst du dir über die Download-Seite zum Buch herunterladen: www.rheinwerk-verlag.de/4975

[Hintergrundinfo]

Es reicht sogar, wenn du dir nur das JDK herunterlädst und installierst. Das JRE wird dann nämlich direkt mit installiert.

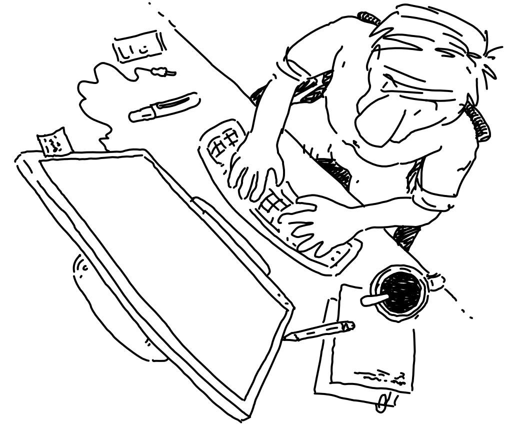
Von diesem „Dschej Di Kej" gibt's aber verschiedene Versionen: Java SE, Java EE, Java ME. Was soll das alles heißen, und welche soll ich nehmen?

In der Tat gibt es unterschiedliche Versionen des JDKs, abhängig davon, was du mit Java alles machen möchtest. **SE** ist die **Standard Edition** und enthält all das, was du eben standardmäßig für die Entwicklung brauchst. **EE** steht für **Enterprise Edition**,

und die bietet zusätzlich noch vieles, vieles mehr, das du benötigst, wenn du zum Beispiel Webanwendungen mit Java entwickeln möchtest. **ME** schließlich ist die **Mobile Edition** oder auch **Micro Edition**, mit der du, wie der Name schon sagt, Java-Anwendungen für mobile Endgeräte erstellen kannst. Uns reicht im Moment die Standard Edition.

[Achtung]

Die **Java Mobile Edition** hat übrigens nichts mit **Android** zu tun. Es handelt sich um eine komplett eigenständige Softwarebibliothek, die auch schon wesentlich älter ist als Android. Mobile Anwendungen damit sind zwar nicht immer so schick wie die mit Android entwickelten, laufen dafür aber auch auf Nicht-Android-Handys, auf denen Java installiert ist.



Java installieren

Auf eine wichtige Sache möchte ich dich noch hinweisen, bevor du dir Java installierst. Und zwar hat Oracle – die Firma, die Java mittlerweile weiterentwickelt – entschieden, dass Java ab Version 11 (sowohl JDK als auch JRE) nur noch in Entwicklungs- und Testumgebungen kostenfrei nutzbar ist.

Wie, das heißt, ich muss jetzt bezahlen, um Java verwenden zu können?

A

Naja, es kommt darauf an. Zum Lernen und für das Durcharbeiten dieses Buches kannst du Java natürlich nutzen, ohne dafür zu zahlen. Die Einschränkung betrifft eher Unternehmen oder Entwickler, die kommerzielle Software entwickeln und dazu dann Support-Verträge mit Oracle abschließen möchten.

Hört sich kompliziert an.

Ja, ist es im Detail auch. Aber das sollte dich jetzt erstmal nicht stören. Denn ich kann dich beruhigen: Oracle stellt noch das sogenannte OpenJDK zur Verfügung, und das kannst du immer kostenlos verwenden – also auch in der Produktion. Und überhaupt: Für dieses Buch kannst du beide Versionen verwenden.

[Zettel]

Eine Anleitung dazu, wie du das OpenJDK installieren kannst, findest du unter http://r-wrk.de/openjdk. Wenn du dagegen das „normale" JDK installieren möchtest, findest du die Anleitung hier: http://r-wrk.de/jdk.

Hallo Schrödinger

Bist du bereit? Dann los! Fangen wir mit einem einfachen Programm an, das den Text „Hallo Schrödinger" ausgibt.

[Notiz]
Traditionsgemäß werden Programmiersprachen in einem Beispiel vorgestellt, das einfach nur die Worte „Hello World" auf die Konsole schreibt.

Wie originell!

[Einfache Aufgabe]
Tippe den folgenden Quelltext (in einem beliebigen Texteditor) ab, und speichere ihn in der Datei **`HalloSchroedinger.java`**.

*2 **public** gibt dabei an, dass deine Klasse **öffentlich** ist. Jede Klasse, die als **public** markiert ist, muss in einer eigenen Datei gespeichert werden, die den gleichen Namen hat wie die Klasse.

*1 Klassen werden in Java mit dem Schlüsselwort **class** definiert.

*3 In unserem Fall heißt die Klasse **HalloSchroedinger**, deswegen muss die Datei **HalloSchroedinger.java** heißen.

```
public *2 class *1 HalloSchroedinger *3 { *4
  public *6 static *7 void *8 main *5(String[] args *9 ) { *10
    System.out.println("Hallo Schrödinger"); *11
  } *10
} *4
```

[Notiz]
Deine Programme werden schon bald aus mehr als nur einer Klasse bestehen. Trotzdem gibt es nur einen Einstiegspunkt für dein Programm und damit in der Regel auch nur eine **main**-Methode.

[Achtung]
Daraus lässt sich aber auch ableiten, dass eine Java-Datei nur eine einzige als **public** markierte Klasse enthalten darf.

*4 Alles, was innerhalb dieser Klammern steht, gehört zu dieser Klasse.

*5 Der **Einstiegspunkt** für jedes Java-Programm ist die **main**-Methode. Das ganze Drumherum (**public static void** und **String[] args**) musst du dir jetzt erstmal nicht merken, aber erklären will ich's dir trotzdem kurz.

*6 **public** heißt hier, dass die **main**-Methode öffentlich zugänglich ist und damit auch von außerhalb der Klasse aufgerufen werden kann. **public** ist ein sogenannter **Modifikator**, von denen du später noch weitere kennenlernen wirst. Die **main**-Methode muss immer **public** sein, damit dein Programm auch von „außen", also von dir, gestartet werden kann.

*7 **static** bedeutet, dass du **keine Objektinstanz** brauchst, um diese Methode aufzurufen. Solche Methoden werden auch **statische Methoden** oder **Klassenmethoden** genannt (keine Angst, zu all dem später mehr, und zwar in Kapitel 5, wenn es um Klassen und Objekte geht).

Und weiter? Du hast eben von einer virtuellen Maschine gesprochen, in der Java die Programme laufen lässt. Wie bekomme ich mein Programm jetzt in diese Maschine?

Ruhig, ruhig, wir sind ja noch nicht fertig mit der Aufgabe. Um dein Programm in der virtuellen Maschine laufen lassen zu können, musst du es **zuerst kompilieren**.

Kompi-was?

Kompi-lie-ren oder auch **übersetzen**, **umwandeln**. Denn im Gegensatz zu Sprachen wie PHP oder Perl, in denen zur **Laufzeit** (also erst, wenn das Programm auch läuft) der Quelltext von einem **Interpreter** interpretiert wird, musst du in Java den Quelltext kompilieren, bevor du ihn laufen lassen kannst, und zwar mit einem **Compiler**. Beide Ansätze haben ihre Vorzüge, die wir aber jetzt nicht weiter erörtern wollen. Nur so viel: Kompilierte Programme sind in der Regel **schneller**.

*8 Und `void` sagt einfach, dass diese Methode **keinen Rückgabewert** hat.

*9 `String` ist ein Datentyp für **Zeichenketten**, `String[]` ist ein Array von Strings, also so etwas wie eine Liste mehrerer Zeichenketten. Und `args` ist der Name des Parameters. Sowohl Strings als auch Arrays gucken wir uns noch ausführlich an, ist doch logisch.

*10 Alles, was innerhalb dieser Klammern steht, gehört zu der `main`-Methode.

*11 Mit dem Befehl `System.out.println()` kannst du auf die Konsole schreiben.

Kompilieren von Hand

Kompilieren tust du so: Öffne eine Konsole, gehe zu dem Ordner, in dem die Datei liegt, und gib einfach den Befehl `javac HalloSchroedinger.java` ein, **aber zuerst …**

Hab ich gemacht, und ich bekomme diesen Fehler hier.

```
Der Befehl "javac" ist entweder falsch geschrieben oder
konnte nicht gefunden werden.
```

… **zuerst müssen wir** deiner Kommandozeile diesen Befehl überhaupt beibringen. Schrödinger, Schrödinger, du bist zu ungeduldig. Lass mich doch erst ausreden.
Der Fehler kommt daher, weil wir den Pfad zu dem `javac`-Befehl noch nicht der **`Path`-Umgebungsvariablen** hinzufügt haben. Am besten fügst du sogar eine neue Umgebungsvariable hinzu, die du **`JAVA_HOME`** nennst und auf den Installationsort des JDKs zeigen lässt. Diese Variable verwendest du dann im **`Path`**. War das zu schnell?

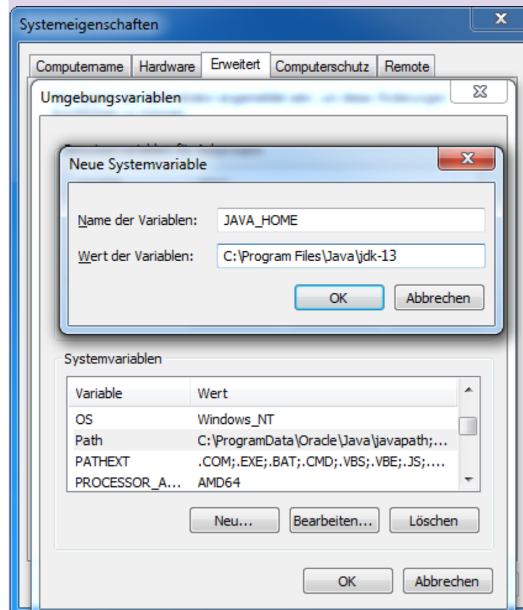
Hier nochmal ganz genau, wie du es machst:

Unter Windows 10 machst du es so:

1. Rechtsklick auf das **Windows-Logo** • „**System**“ • „**Erweiterte Systemeinstellungen**“

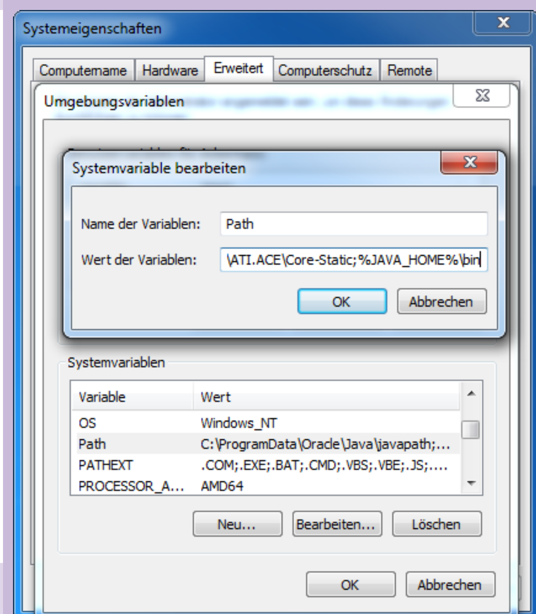
2. In dem Dialog klickst du auf „Erweitert“, anschließend auf „Umgebungsvariablen“, und dann legst du eine neue Systemvariable an wie in folgendem Screenshot:

Eine separate Umgebungsvariable **JAVA_HOME** ist die sauberste Lösung.



3. Anschließend passt du die **Path**-Variable wie folgt an:

Die Path-Variable



[Notiz]

Mit der Schreibweise **%JAVA_HOME%** kann die Variable, die du eben angelegt hast, innerhalb anderer Variablen verwendet werden.

Unter Linux und macOS machst du es so:

Am besten trägst du Folgendes in die Datei **~/.bash_profile** ein, damit der Speicherort von Java bekannt ist, wenn du eine Shell startest:

```
export JAVA_HOME =/pfad/zum/java/verzeichnis*1
export PATH =$PATH:$JAVA_HOME/bin*2
```

*1 Wieder legst du erst die Variable **JAVA_HOME** an …

*2 … und fügst sie dann mit vorangestelltem **$**-Symbol der **PATH**-Variablen hinzu.

[Erledigt!]
Über die Umgebungsvariablen findet das Betriebssystem jetzt die Java-Installation, und du kannst die verschiedenen Konsolenbefehle von Java in der Eingabeaufforderung bzw. in der Shell verwenden.

[Achtung]
Eventuell musst du die Eingabeaufforderung neu starten, damit die Umgebungsvariablen auch erkannt werden und alles funktioniert. Mit dem Konsolenbefehl **echo $JAVA_HOME** bzw. **echo $PATH** kannst du sie testen.

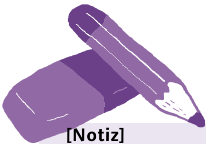
Wenn du damit fertig bist, kann's ja weitergehen: Wechsle zur Kommandozeile, und gib **jetzt** den folgenden Befehl ein (natürlich wieder in dem Ordner, in dem die Java-Datei liegt):

```
javac HalloSchroedinger.java
```

```
C:\Windows\system32\cmd.exe
D:\Work\GalileoPress\workspace>javac HalloSchroedinger.java

D:\Work\GalileoPress\workspace>
```

Standardmäßig liefert das Kompilieren keinen großen Output.



[Notiz]
Du kannst den Compiler etwas gesprächiger machen, indem du **-verbose** an den Befehl dranhängst, also so:

```
javac -verbose HalloSchroedinger.java.
```

Wenn du dir jetzt mal den Inhalt des Ordners anschaust, wird dir auffallen, dass eine neue Datei hinzugekommen ist: **HalloSchroedinger.class**. Das ist die kompilierte Datei! Herzlichen Glückwunsch, Schrödinger! du hast dein erstes Programm kompiliert. Wie sieht's aus? Kurze Pause, oder sollen wir direkt weitermachen?

Weitermachen! Ich will mein Programm laufen lassen!

Programm starten

Hab ich mir schon fast gedacht.

Das machst du mit dem Befehl `java`, und zwar so:

```
java HalloSchroedinger
```

[Achtung]

Dem `javac`-Befehl übergibst du als Parameter den **kompletten Dateinamen** (also inklusive Dateiendung). Dem `java`-Befehl übergibst du **nur den Klassennamen** des Programms, das du ausführen willst (also ohne die Dateiendung `.class`).

[Fehler]

Falls du unter Windows arbeitest, kann es sein, dass dein Programm „Hallo Schr÷dinger" liefert, den Umlaut also nicht erkennt. Das kommt daher, dass die Eingabeaufforderung von Windows standardmäßig die ASCII-Codierung verwendet, die keine Umlaute kennt. Du kannst das Encoding aber für die Ausführung deines Programms ändern, indem du dem `java`-Befehl einen Parameter mitgibst:

```
java *1-Dfile.encoding=CP850 HalloSchroedinger
```

*1 Mit dem `-D` kannst du der JVM **Parameter** übergeben. Ein Parameter besteht aus einem Namen (hier `file.encoding`) und einem Wert (hier `CP850`).

Jetzt sollte die Ausgabe auch unter Windows „Hallo Schrödinger" lauten.

[Notiz]

Es gibt übrigens noch eine ganze Reihe weiterer JVM-Parameter. Eine Liste findest du zum Beispiel hier: http://r-wrk.de/vm_options.

[Notiz]

Neben den JVM-Parametern, die du mit `-D` an den Compiler übergeben kannst, gibt es auch noch verschiedene Compiler-Optionen, zum Beispiel `-d`, um das Zielverzeichnis für die kompilierten Dateien anzugeben, oder `-source`, um die Java-Version anzugeben, für die du kompilieren möchtest. Eine Liste aller Compiler-Optionen findest du bei Oracle, für Java 13 und Windows liegt sie hier: http://r-wrk.de/java_options. Das musst du aber nicht auswendig lernen.

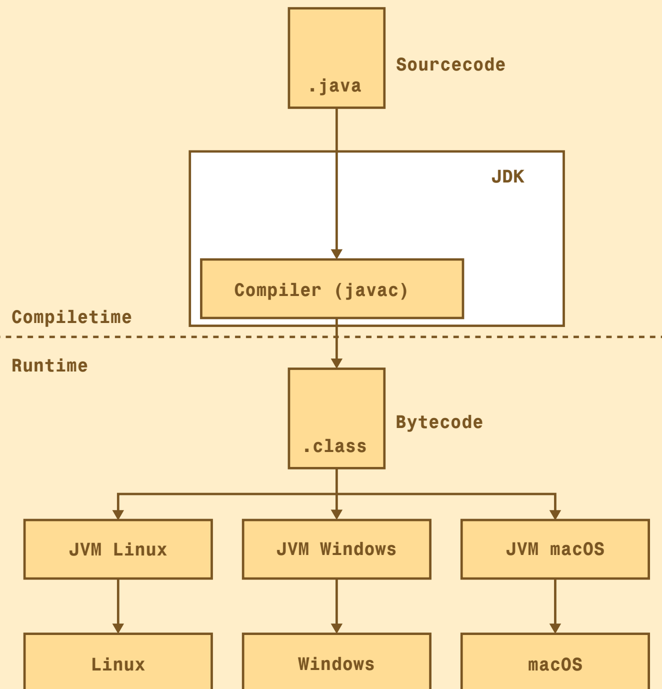
[Notiz]

Seit Java 11 hast du übrigens auch die Möglichkeit, Quelltextdateien direkt mit dem java-Befehl auszuführen:

```
java HalloSchroedinger.java
```

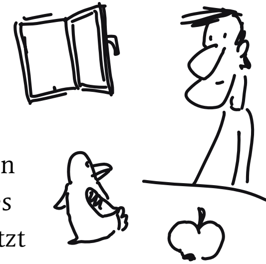
Compiler und JVM unter der Lupe

Lass uns nochmal zurückblicken auf das, was du bisher gemacht hast. Schau dir dazu folgende Übersichtsgrafik an:



Ganz am Anfang steht deine Quelltext-Datei. Der Compiler ist Bestandteil des JDKs und übersetzt den Quelltext in **Bytecode**, also in das, was die `class`-Datei enthält. Der Zeitpunkt des Kompilierens wird auch **Compiletime** oder **Übersetzungszeit** genannt.
Anschließend kann der Bytecode in jeder beliebigen JVM ausgeführt werden, vorausgesetzt natürlich, die Java-Version reicht aus. Das passiert dann zur sogenannten **Runtime** oder **Laufzeit** unter Verwendung des JREs. Die JVM interpretiert den Bytecode und wandelt ihn in speziellen, auf das jeweilige Betriebssystem zugeschnittenen Maschinencode um. Und dieser **Maschinencode** ist das einzige, was für jedes Betriebssystem unterschiedlich ist.

*Okay, verstehe. Heißt das, ich brauche mein Programm nur **einmal zu kompilieren**, und es läuft direkt überall?*

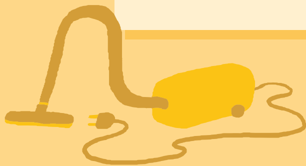


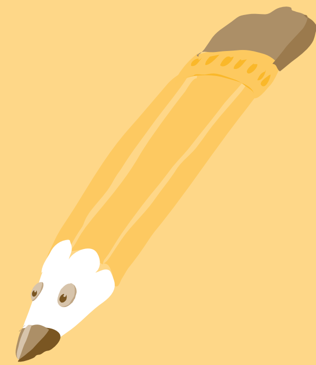
Genau, du könntest die `class`-Datei, die der Compiler eben für uns erstellt hat, direkt einem Bekannten geben, der ein anderes Betriebssystem hat als du, und es würde sofort laufen, vorausgesetzt natürlich, er hat die **JVM installiert**. Bei C++ zum Beispiel ist das anders. Da musst du als Entwickler für jedes Betriebssystem den Compiler anschmeißen und unterschiedliche Versionen deines Programms erzeugen. Allerdings muss man diesem Ansatz zugutehalten, dass die Programme dort ohne etwas Vergleichbares wie die JVM laufen.

[Zettel]
In der JVM kannst du übrigens noch weitere Sprachen ausführen, wie zum Beispiel Groovy, Scala oder Clojure, um nur einige wenige zu nennen.

Rätselstunde

[Einfache Aufgabe]
Nimm einen Bleistift und verbinde die Halbsätze.





Der Compiler ...
Die JVM ...
Der Befehl `java` ...
Der Befehl `javac` ...

... erstellt `class`-Dateien.
... macht aus Quellcode Bytecode.
... verarbeitet `class`-Dateien.
... verarbeitet `java`-Dateien.
... sorgt dafür, dass dein Programm auf verschiedenen Betriebssystemen läuft.
... führt dein Programm aus.

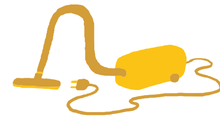
Hier die Lösung:

Der Compiler erstellt **class**-Dateien.
Der Compiler macht aus Quellcode Bytecode.
Der Compiler verarbeitet **java**-Dateien.
Die JVM verarbeitet **class**-Dateien.
Die JVM sorgt dafür, dass dein Programm auf verschiedenen Betriebssystemen läuft.
Die JVM führt dein Programm aus.
Der Befehl **java** führt dein Programm aus.
Der Befehl **javac** erstellt **class**-Dateien.
Der Befehl **javac** macht aus Quellcode Bytecode.
Der Befehl **javac** verarbeitet **java**-Dateien.

Hexenstunde

[Einfache Aufgabe]

Noch eine Aufgabe für Abergläubische, die **unbedingt ein echtes „Hallo Welt"** auf die Konsole schreiben wollen: Ändere die Ausgabe deines Programms entsprechend, und lass das Programm laufen.



Okay, hab ich geändert und lasse das Programm laufen mit dem `java`*-Befehl … Komisch, die Ausgabe ist immer noch „Hallo Schrödinger"!*

Genau, es reicht nicht, nur den Quelltext zu ändern, du musst das Programm auch wieder neu …

… kompilieren! Stimmt ja, Moment, mit dem `javac`*-Befehl. Ah, jetzt funktioniert's! Hallo Welt!*

[Einfache Aufgabe]

Erstelle mit dem Texteditor noch eine zweite Datei im gleichen Verzeichnis wie **HalloSchroedinger.java**, und zwar **Hexentexte.java** mit diesem Code:

```
public class Hexentexte {
	public static String HOKUSPOKUS = "Hokuspokus!";
	public static String PROPHEZEIUNG = "Du wirst programmieren.";
	public static String KEINE_HEXEREI = "Das ist keine Hexerei!";
}
```

Hexentexte.PROPHEZEIUNG

sieht nach Hexerei aus. Was macht denn der Punkt?

Nö, das ist wie eine Wegbeschreibung. Pass auf:

„Wo geht es hier zur Prophezeiung, bitte?“

Quelle: http://de.academic.ru/dic.nsf/dewiki/180895

PROPHEZEIUNG
Hexentexte.PROPHEZEIUNG
„Erst gehst du zur Klasse **Hexentexte**. Die findest du in der Datei **Hexentexte.class**. Dort muss eine Variable namens **PROPHEZEIUNG** definiert sein (wenn nicht, rufe: „Fehler!“)."

[Schwierige Aufgabe]
Ergänze dann in der **main**-Methode in **HalloSchroedinger.java** die Zeile

```
System.out.println(Hexentexte.PROPHEZEIUNG);
```

Kompiliere **HalloSchroedinger.java** erneut, und lass das Programm laufen.

Und? Fällt dir was auf?

Öh, nö? Die Ausgabe lautet jetzt: „Du wirst programmieren.“ Was soll mir da noch großartig auffallen?

Na ja, du hast die Hexentexte ja gar nicht kompiliert. Trotzdem findet das Programm die kompilierte Klasse. Denn merke:

[Notieren/Üben]
Wenn du von einer Java-Datei auf eine andere Java-Datei referenzierst, die noch nicht kompiliert wurde, und dann nur die erste Datei mit **javac** kompilierst, wird automatisch auch **die referenzierte Datei mit kompiliert**.

Gut zu wissen ... ähm, jetzt wo du es sagst.

[Einfache Aufgabe]
Ändere **PROPHEZEIUNG** in: „Hekate, die Göttin des Spuks, wird deine Spiele verzaubern.“
Kompiliere **HalloSchroedinger** erneut, und lass es nochmal laufen.

Das funktioniert ja wieder direkt! Moment mal, heißt das etwa, dass ...

[Notieren/Üben]
... auch referenzierte Dateien, von denen die **kompilierte Version nicht mehr aktuell ist**, automatisch mit kompiliert werden. Ja, richtig, das heißt es!

Entwicklungsumgebungen

Schrödinger hat's geschafft, der erste Code läuft, und vor seinem inneren Auge laufen schon die ersten selbst programmierten Spiele.

Wenn du wirklich ernsthaft entwickeln möchtest, wird dir ein einfacher Texteditor nicht mehr ausreichen.

Und alles händisch auf der Konsole zu kompilieren, macht auch nur begrenzt Spaß. Du brauchst also etwas Mächtigeres. Zum Glück gibt es gleich eine Reihe wirklich guter Tools zur Java-Entwicklung, sogenannte **Entwicklungsumgebungen** (oder auch IDEs – Integrated Development Environments). Solche Entwicklungsumgebungen bieten **verschiedene Features**, sie heben zum Beispiel den **Quelltext farblich hervor**, ermöglichen **automatisches Erzeugen und Vervollständigen von Quelltext** und vieles, vieles mehr. **Eclipse**, **Netbeans**, **IntelliJ** und **JBuilder** dürften die bekanntesten sein. Alle gibt es für Windows, Linux und macOS, aber nur Eclipse und Netbeans sind kostenlos. Dir bleibt natürlich selbst überlassen, welche IDE du letztendlich verwendest, wir nehmen – **ene, mene, muh – Eclipse.**

Eclipse herunterladen

Eclipse kannst du dir hier herunterladen: www.eclipse.org/downloads.

Uff, schon wieder so viele verschiedene Versionen!

Stimmt, man wird nahezu erschlagen von der Menge an möglichen Downloads. Nimm für den Anfang die **Eclipse IDE for Java Developers**. Das Gute an Eclipse ist nämlich, dass alles, was du später benötigen solltest, auch über **Plug-ins** nachinstalliert werden kann. Es gibt da eigentlich nichts, was es nicht gibt: Es gibt Plug-ins für Webentwicklung, Android-Entwicklung, für Datenbankverwaltung, für Web Services, fürs Kaffeekochen, für ALLES. Und wenn es dann doch mal für etwas noch kein Plug-in geben sollte, schreibst du einfach dein eigenes. Denn Eclipse selbst ist ebenfalls **in Java programmiert.**

[Achtung]

Apropos Plug-in: Es kann sein, dass die aktuelle Eclipse-Version noch nicht die neuesten Features von Java 12 unterstützt, wie sie in diesem Buch besprochen werden. Du kannst dir in diesem Fall aber den Java-12-Support über ein Plug-in hinzuinstallieren. Mehr dazu im Downloadbereich unter www.rheinwerk-verlag.de/4975.

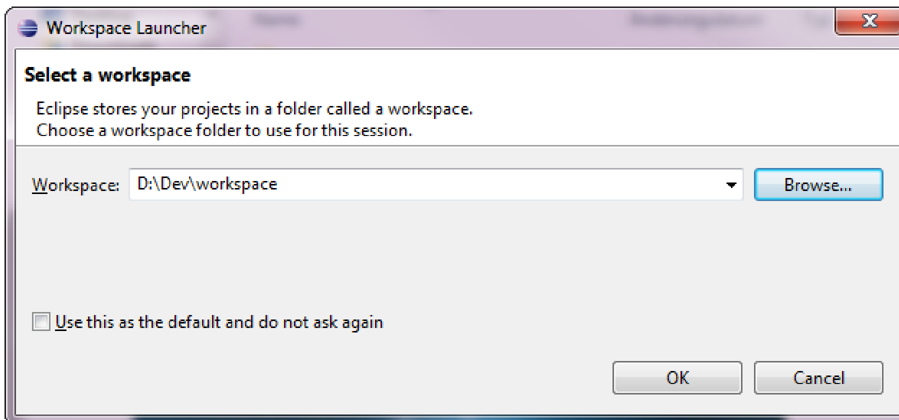
[Hintergrundinfo]

Weil Eclipse in Java programmiert ist, benötigt es auch ein installiertes JRE, um überhaupt starten zu können. Gut, dass wir das schon erledigt haben.

Die Installation von Eclipse ist denkbar einfach: Du musst nur die heruntergeladene Datei irgendwohin entpacken, das war's! Cool, oder? Im entpackten Ordner findest du dann eine ausführbare Datei, mit der du Eclipse starten kannst.

Workspace und Workbench

Beim Start fragt dich Eclipse zunächst, wo dein **Workspace** liegt. Das ist grob gesagt nichts anderes als ein Ordner, in dem Eclipse Informationen über deine Projekte verwaltet. Und fürs Erste ist das der Ordner, in dem auch dein Quelltext zu Hause ist.



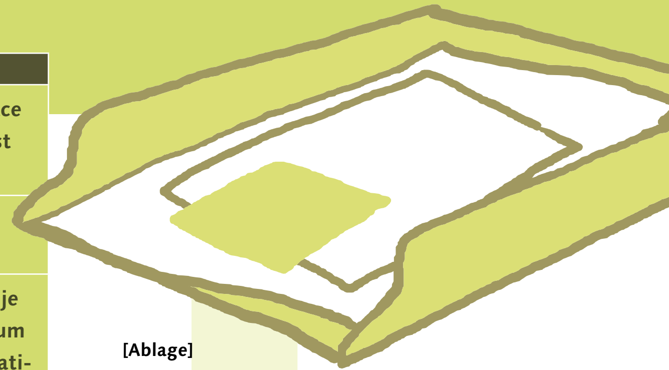
Der Auswahl-Dialog für den Workspace

Wenn du Eclipse das allererste Mal startest, bekommst du nach der Workspace-Auswahl noch einen speziellen Welcome-Screen zu sehen, von dem aus du zum Beispiel Tutorials über Eclipse aufrufen kannst. Wir halten uns jetzt aber nicht damit auf, klicke also auf den Pfeil, der dich zur sogenannten **Workbench** von Eclipse führt.

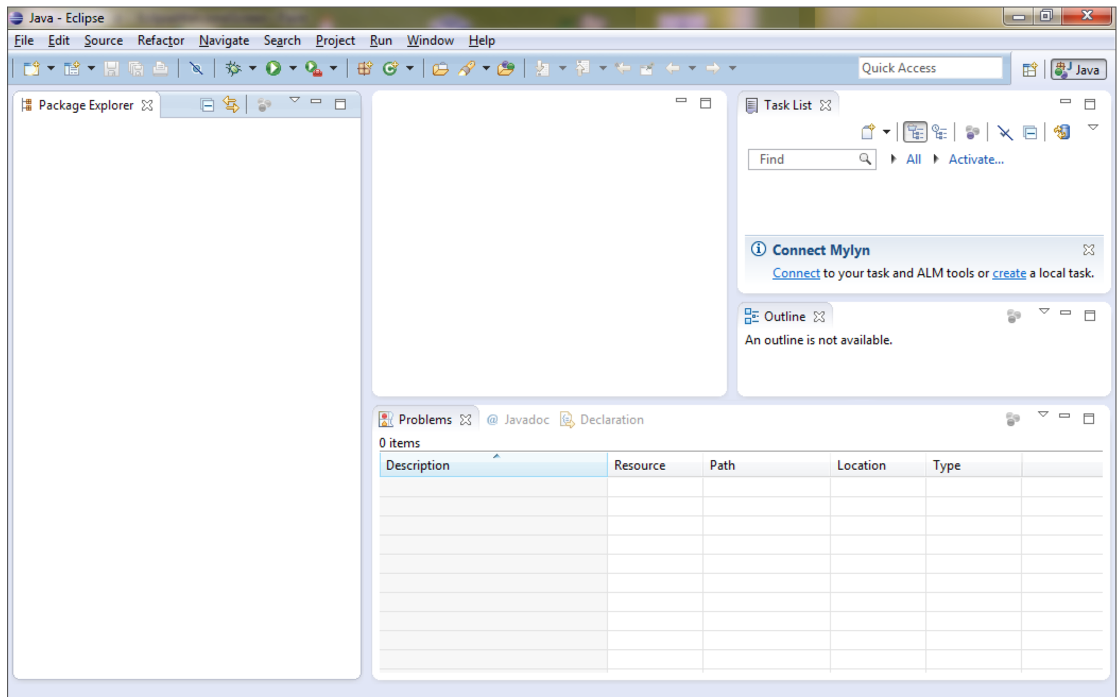
Die Workbench ist der Dreh- und Angelpunkt für die Entwicklung. Von hier aus kannst du verschiedene **Perspektiven** und **Views** für die unterschiedlichsten Anwendungszwecke öffnen. Views sind einfach kleine (oder auch größere) Fenster, die jeweils einen bestimmten Zweck erfüllen. Eine Perspektive ist eine Zusammenstellung verschiedener Views.

Hier ein paar Views der Java-Perspektive:

View-Name	Beschreibung
Package Explorer	Hier werden die aktiven Projekte aus deinem Workspace angezeigt. Im Moment ist diese View noch leer, du hast ja noch keine Projekte.
Task List	Hier kannst du Aufgaben definieren, die du erledigen musst.
Outline	In dieser View werden Detailinformationen angezeigt, je nachdem, was du gerade ausgewählt hast. Wenn du zum Beispiel eine Klasse geöffnet hast, werden hier Informationen zu der Klasse aufgelistet.
Problems	Gibt es Warnungen oder Fehler vom Compiler, werden die hier angezeigt.
Console	Wenn du ein Programm startest, das etwas ausgibt (zum Beispiel über `System.out.prinln()`), dann siehst du die Ausgabe hier. Aber auch wenn dein Programm eine Nutzereingabe erwartet, kannst du die über diese View eingeben.



[Ablage]
Die Standardinstallation von Eclipse enthält viele verschiedene Perspektiven, die du nach und nach noch entdecken wirst. Zur Perspektivenauswahl gelangst du über **Window • Open Perspective**.



Die Java-Perspektive ist für die Java-Entwicklung gedacht.

Erstes Projekt

Wollen wir also mal den Workspace füllen und dein erstes Projekt erstellen.

[Einfache Aufgabe]

Erstelle ein neues Java-Projekt mit Eclipse.

Wähle aus dem Hauptmenü **File • New • Java Project**. Damit öffnest du den Dialog zur Erstellung von Java-Projekten. Für den Moment reicht es völlig, wenn du dem Projekt einen Namen gibst und auf **Finish** klickst. Der Vollständigkeit halber zeige ich dir aber kurz, was du in diesem Dialog noch so alles konfigurieren kannst.

New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: SchroedingerProgrammiertJava

Use default location

Location: D:\Dev\workspace\SchroedingerProgrammiertJava Browse...

JRE

Use an execution environment JRE: JavaSE-1.7

Use a project specific JRE: jre7

Use default JRE (currently 'jre7') Configure JREs...

Project layout

Use project folder as root for sources and class files

Create separate folders for sources and class files Configure default...

Working sets

Add project to working sets

Working sets: Select...

< Back | Next > | Finish | Cancel

Hier gibst du dem Projekt einen Namen.

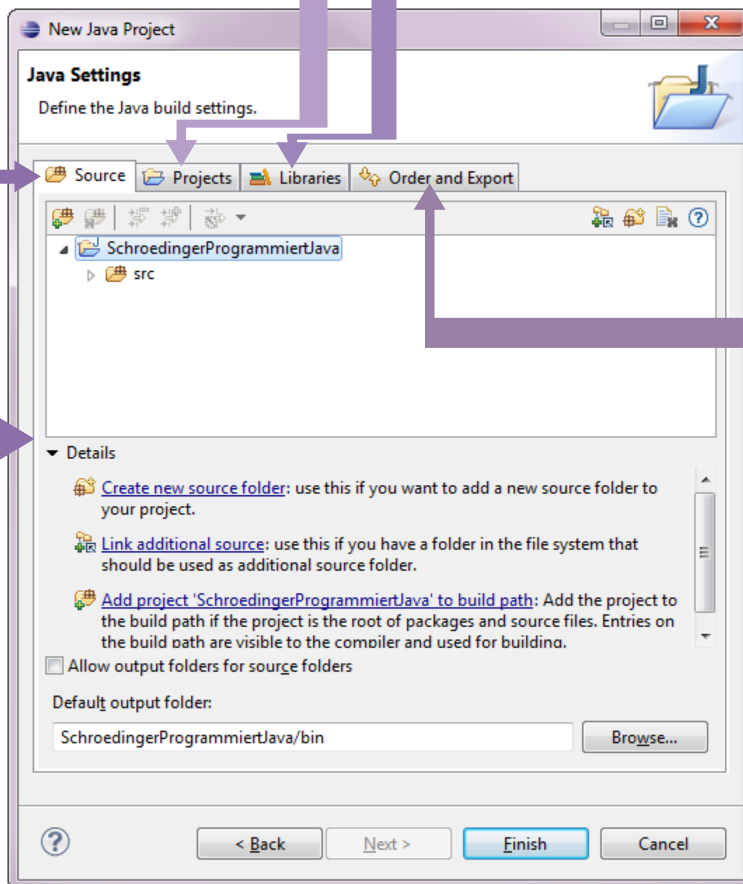
Hier kannst du die Java-Laufzeitumgebung auswählen.

Hier kannst du bestimmen, in welchen Ordner die kompilierten class-Dateien gespeichert werden. Am besten, du nimmst verschiedene Ordner für den Quelltext und die kompilierten Klassen.

Hier kannst du **andere Projekte aus deinem Workspace** als Abhängigkeit zu dem neuen Projekt hinzufügen. Für den Anfang jetzt noch uninteressant, aber später kannst du auf diese Weise gut Quelltext von einem Projekt in einem anderen Projekt wiederverwenden.

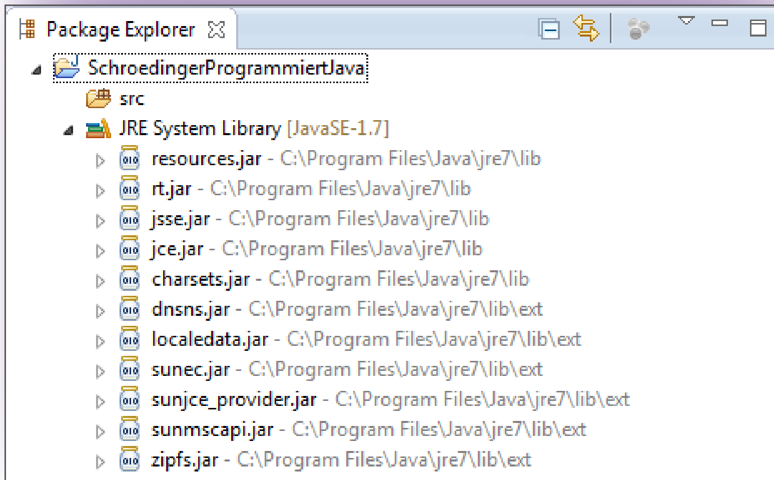
Hier kannst du festlegen, welche Ordner als Quelltext-Ordner verwendet werden. Standardmäßig werden alle Java-Dateien im Ordner src gespeichert.

Hiermit kannst du **externe Bibliotheken** als Abhängigkeit zu dem neuen Projekt hinzufügen.

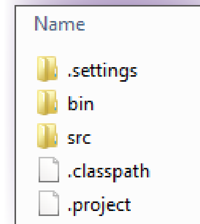


Hierüber legst du die Reihenfolge fest, in der deine Quelltext-Ordner kompiliert und exportiert werden sollen.

Nachdem du das Projekt erstellt hast, kannst du den **Inhalt des Projekts** im **Package Explorer** sehen. Bis auf den `src`-Ordner und das verlinkte JRE ist das Projekt noch ziemlich leer. Ist ja auch klar, wir haben ja noch keine Java-Dateien erstellt.



Das Projekt im Package Explorer …



… und im Datei-Explorer

Aber wenn ich den Projektordner in meinem Datei-Explorer öffne, sind da noch viel mehr Ordner und Dateien.

Das ist richtig, die verbirgt Eclipse im Package Explorer vor dir, weil du damit in der Regel nichts zu schaffen hast. Ich erklär dir natürlich trotzdem gerne, was sich dahinter verbirgt:

- Der `.settings`-Ordner und die Dateien `.classpath` und `.project` enthalten jede Menge Konfigurationen für dein Projekt, zum Beispiel die Abhängigkeiten zu externen Bibliotheken. All das kannst du aber auch irgendwie über Eclipse-Dialoge anpassen.
- In den `bin`-Ordner kommen die kompilierten Java-Dateien, also die `class`-Dateien, rein.
- Der `src`-Ordner beherbergt den Quelltext.

Pakete packen, aber ordentlich

Glaub mir, du wirst als Entwickler jede Menge Java-Dateien erstellen, und da ist es schon ganz sinnvoll, diese Dateien nach irgendeinem Schema zu **ordnen**. Alle Dateien an einem Ort bzw. in einem Verzeichnis zu speichern, rate ich dir nicht! Irgendwann verlierst du dann nämlich nicht nur die Übersicht, sondern wirst auch Probleme haben, deine Dateien eindeutig zu benennen. Java bietet zu diesem Zweck sogenannte **Packages** (oder Pakete) an.

Ein Package zu erstellen, ist einfach:

Erstelle in deinem Eclipse-Projekt ein neues Paket mit dem Namen **de.galileocomputing.schroedinger.java.kapitel01. Vergleiche** dein Projekt **nochmal** im Package-Explorer und im Datei-Explorer.

Was für ein blöder Name. Was sollen die ganzen Punkte und das „de“ am Anfang? Das sieht aus wie eine umgedrehte Internetadresse ohne www davor … äh, dahinter.

Ja, gut aufgepasst:

Das Ziel bei der Namenswahl ist, einen möglichst **eindeutigen Namen für dein Paket** zu wählen, und zwar am besten einen, der auch **weltweit eindeutig** ist! Und am besten machst du das, indem du als Basis dafür die URL einer Internetseite nimmst, die dir gehört (oder deinem Kunden oder deiner Freundin oder deiner Katze oder irgendjemand anderem, für den du das Programm entwickelst, **irgendjemandem halt und niemand anderem**), und diese URL dann umdrehst.

http://www.galileocomputing.de

gehört zum Beispiel dem Rheinwerk Verlag und wird zu

de.galileocomputing

Wenn du jetzt in diesem Paket Dateien anlegst, kannst du sicher sein, dass du niemandem auf der Welt in die Quere kommst (sofern sich alle an diese Konvention halten), außer natürlich denen, die auch bei Rheinwerk Computing Java entwickeln. Und um denen auch nicht in die Quere zu kommen, hängst du einfach noch deinen Namen an den Paketnamen dran. Zum Schluss kommen noch ein paar mehr Sachen dran, um uns später auch nicht selbst in die Quere zu kommen, daher der Name **`de.galileocomputing.schroedinger.java.kapitel01`**. Das heißt also, man wird von links nach rechts im Paketnamen immer spezieller.

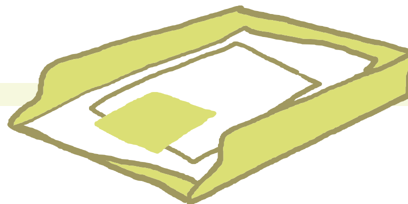
Okay, aber warum MUSS ich denn überhaupt eindeutig sein?

[Zettel]
Von Müssen ist nicht die Rede, aber du solltest es. Vielleicht wirst du ja später mal Bibliotheken von jemand anders verwenden, oder du schreibst eine so coole Bibliothek, dass sie jemand anders in seinem Programm verwenden möchte. Das funktioniert aber nur fehlerfrei, wenn es keine Konflikte gibt. Wenn es nämlich ein Paket mehrmals gibt (oder noch schlimmer, innerhalb dieser Pakete dann auch noch Dateien mit demselben Namen existieren), kommt die JVM ganz schön durcheinander.

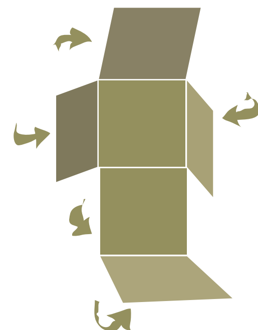
[Hintergrundinfo]
Natürlich hängst du in echten Projekten nicht **deinen eigenen Namen** an den Paketnamen. In der Regel nimmst du den Namen des Projekts, an dem du arbeitest.

[Hintergrundinfo]
Später in diesem Buch lernst du noch die sogenannten Module kennen, eines der Features, das erst mit Java 9 eingeführt wurde. Ja, eigentlich waren Module sogar **das** Feature überhaupt in Java 9. Sie erlauben, na was wohl, eine sehr viel modularere Entwicklung, aber dazu später mehr.

[Ablage]
Verwende zur Strukturierung deines Quelltextes immer Pakete mit eindeutigen Namen.

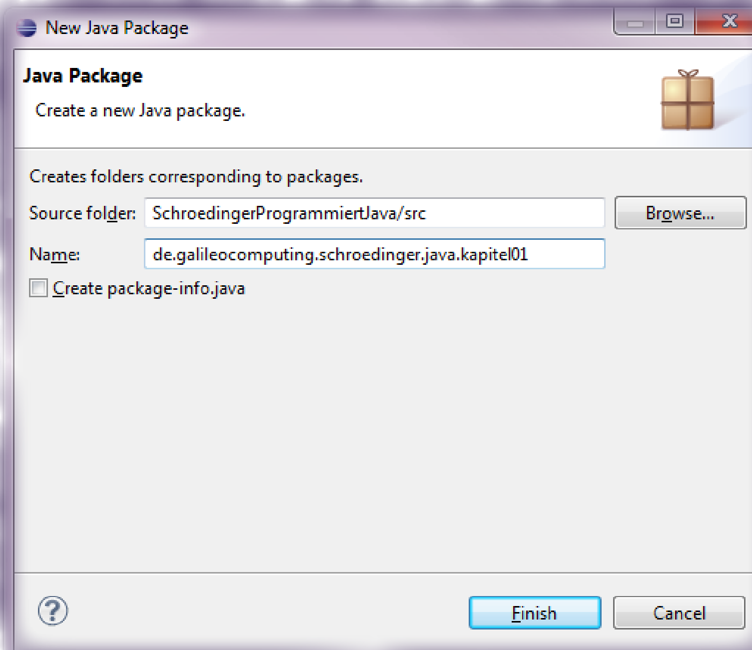


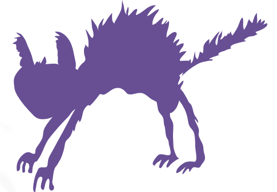
So, jetzt aber zurück zur Aufgabe: Anlegen des Pakets und Vergleichen im Package Explorer von Eclipse und im Datei-Explorer.



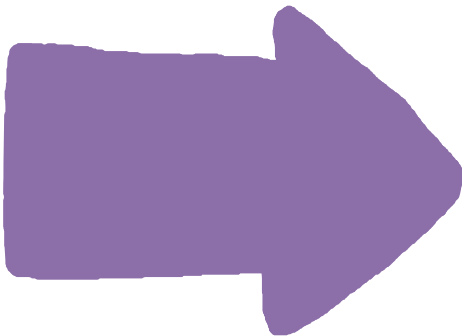
Pakete anlegen leicht gemacht

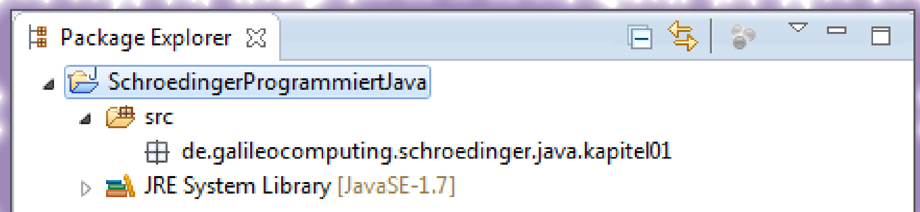
Den folgenden Dialog erreichst du mit **File • New • Package**. Dann gibst du einfach dem Package einen Namen und bestätigst mit „Finish".



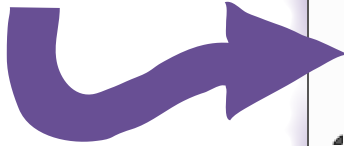


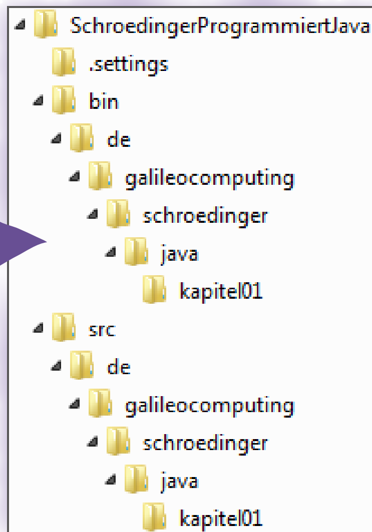
Im Package Explorer sieht das Ganze so aus, wie ich es erwartet habe: ein Paket mit dem Namen
de.galileocomputing.schroedinger.java.kapitel01.





Im Datei-Explorer sieht das bei mir aber so aus:





Und das ist alles genau richtig. Packages in Java spiegeln nämlich die **Verzeichnisstruktur** in deinem Projekt wider. Und Eclipse ist so freundlich, diese Struktur sowohl im `src`-Verzeichnis als auch im `bin`-Verzeichnis anzulegen. So kommt sich garantiert niemand in die Quere. Also, ich finde es ziemlich übersichtlich so.

Ja, hast ja recht.

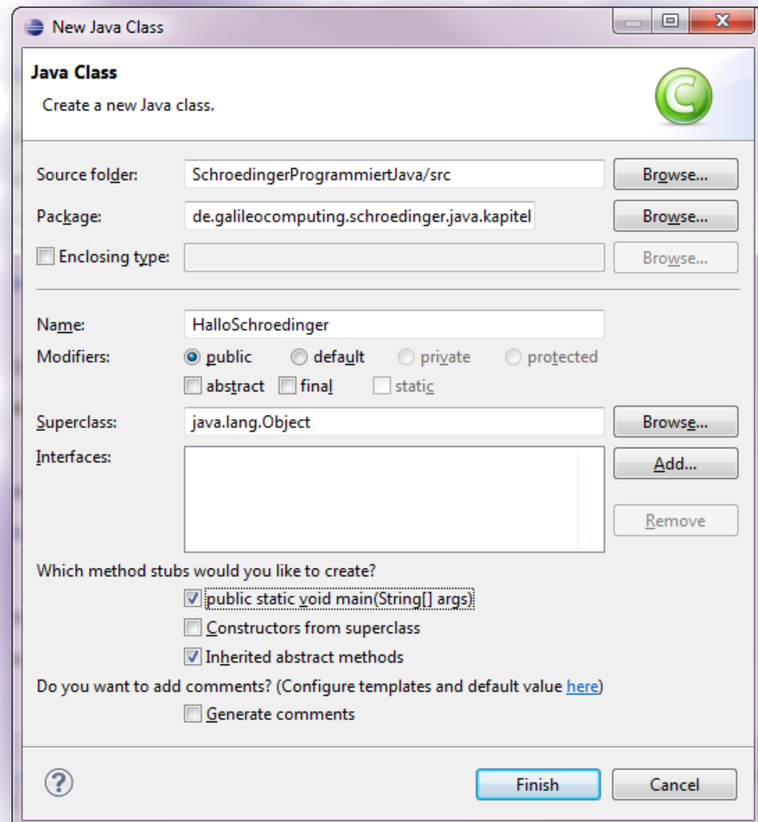
Neue Klasse mit Eclipse

Machen wir das Beispiel „Hallo Schrödinger" nochmal, diesmal aber mit Eclipse und in dem Paket, das du eben angelegt hast.

[Einfache Aufgabe]

Erstelle eine neue Klasse mit dem Namen `HalloSchroedinger`, und passe die Klasse dann so an, dass wieder „Hallo Schrödinger" ausgegeben wird.

Öffne den Dialog zur Erstellung von Klassen über **File • New • Class**. Wähle das Package, das wir eben angelegt haben, gib der neuen Klasse den Namen `HalloSchroedinger`, und wähle aus, dass die Methode `public static void main(String[] args)` direkt mit generiert wird.



Eine neue Klasse kannst du bequem über diesen Dialog anlegen.

Eclipse erstellt für dich das ganze Drumherum, so dass du nur eine Zeile einfügen musst, damit dein Programm so aussieht wie vorher:

```
HalloSchroedinger.java

package de.galileocomputing.schroedinger.java.kapitel01;

public class HalloSchroedinger {

    /**
     * @param args
     */
    public static void main(String[] args) {
        System.out.println("Hallo Schrödinger");
    }

}
```

Deine Klasse sollte jetzt so aussehen.

Hier noch der Quelltext, bis auf den Paketnamen hat sich nichts geändert:

*1 Über **package** gibst du das Package an, in dem die Klasse liegt.

*2 Dahinter kommt dann der Name des Packages, traditionell in der „gespiegelten URL-Schreibweise“.

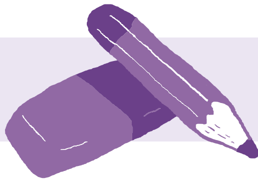
```
package*1 de.galileocomputing.schroedinger.java.kapitel01;*2

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

Das ergibt galileo.computing.de. Ich dachte, dein Verlag hat rheinwerk-verlag.de als Domäne !?!

Stimmt, gut aufgepasst. Die URL **galileocomputing.de** stammt noch aus alten Zeiten. Das ist aber kein Grund, die Paketnamen nachträglich zu ändern. Das sollte man bei einer bestehenden Codebasis vermeiden. In diesem Fall ist das kein Problem, denn Domäne gehört dem Verlag noch. Und so bleiben die Package-Namen weltweit eindeutig.

Verstehe.



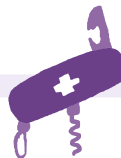
[Notiz]

Wenn du für eine Klasse kein Paket angibst, liegt sie im Default-Package. Das ist nicht weiter schlimm für kleine Programme, die du im stillen Kämmerlein für dich selbst entwickelst, für große Projekte aber definitiv ungeeignet.

Geh jetzt nochmal im Datei-Explorer in den `bin`-Ordner, und zwar runter bis in den Unterordner `kapitel01`. Dort hat Eclipse nämlich direkt auch die Klasse kompiliert. Und das Gute: Eclipse macht das immer, sobald du deine Datei änderst und speicherst. Du brauchst ab jetzt also nie mehr händisch zu kompilieren! Zu wissen, wie es funktioniert, ist aber trotzdem gut.

[Einfache Aufgabe]

Führe jetzt das Programm mit Eclipse aus.



Wähle einfach **Run • Run** oder **Run • Run As • Java Application** aus, dies öffnet eine neue Ansicht, die Konsole, auf der du die Ausgabe deines Programms siehst.

„Hallo Schrödinger". Diesmal sogar mit dem „ö"!

Genau, mit „ö"! Besser als mit der Windows-Eingabeaufforderung, oder? Prima, jetzt weißt du schon, wie man drei wichtige Dinge anlegt:

- ☑ Projekte
- ☑ Pakete
- ☑ Klassen

[Belohnung]

Jetzt kannst du dich ein bisschen auf dem Bürostuhl ausruhen, und wir gucken derweil, wie du dein Programm um ein paar einfache Ein- und Ausgaben erweiterst.

Das nennst du ausruhen?

Miteinander reden

In unserem Codebeispiel hast du ja schon gesehen, wie man etwas auf die Konsole schreiben kann: mit `System.out.println()`. Damit schreibst du ganz normal auf die Konsole. Zusätzlich gibt es noch die Variante `System.err.println()`, die dafür gedacht ist, Fehler auf die Konsole zu schreiben (in der Konsole von Eclipse zum Beispiel wird der Text dann rot ausgegeben). Probier es mal aus.

[Zettel]
Sowohl für `System.out` als auch für `System.err` gibt es jeweils noch eine andere Methode (`print()`), die das Gleiche macht wie `println()`, aber keinen Zeilenumbruch an die Ausgabe hängt.

Das ist alles prima, aber wie bekomme ich denn Daten in mein Programm?

Da gibt es verschiedene Möglichkeiten:

Möglichkeit 1: Startparameter

Wenn du dir von Anfang an sicher bist, was du deinem Programm übergeben möchtest, kannst du ihm **Startparameter** mitgeben, die dir dann in der `main`-Methode über das Stringarray `String[] args` bereitgestellt werden.

Erweitern wir doch das Beispiel. Das wirst du nämlich im jetzigen Zustand nicht besonders erfolgreich vermarkten können. Besser wäre es doch, wenn das Programm „Hallo“ zu beliebigen Personen sagen könnte. Du musst nur Folgendes in deinem Programm anpassen, und schon hast du die Zielgruppe vermilliardenfacht:

```
System.out.println("Hallo " + args[0]);
```

Wie gesagt, `args` ist ein Array von Strings, vereinfacht gesagt so etwas wie eine Liste. Die Null in den eckigen Klammern bedeutet, dass wir auf das erste Element zugreifen, also in dem Fall auf den ersten Startparameter. Fehlt jetzt nur noch die Angabe eines solchen.

Startparameter von Hand auf die Kommandozeile gebracht, sieht so aus:

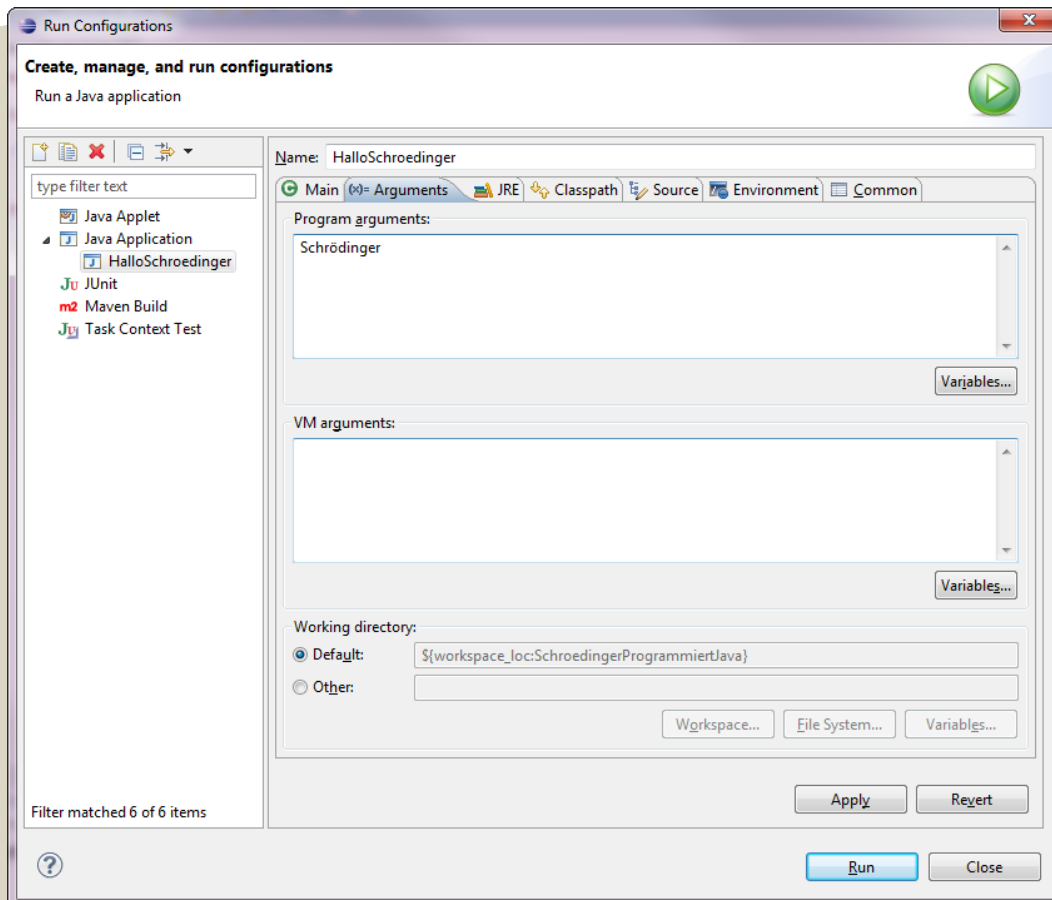
```
java de.galileocomputing.schroedinger.java.kapitel01.HalloSchroedinger *1 Schrödinger *2
```

*1 Da dein Programm jetzt in einem Paket liegt, musst du dem Compiler den vollen bzw. **qualifizierten Klassennamen** übergeben, also Paketnamen plus einfachen Klassennamen. Den Befehl führst du im `bin`-Verzeichnis aus bzw. da, wo der Ordner `de` liegt.

*2 Den oder die Startparameter hängst du dann einfach hinten dran, getrennt durch Leerzeichen.

Mit Eclipse übergibst du Startparameter so:

Eben, als du das Programm das erste Mal in Eclipse hast laufen lassen, wurde im Hintergrund eine sogenannte Run Configuration angelegt. Dort kannst du vieles einstellen, unter anderem auch die Startparameter. Klicke auf **Run • Run Configurations ...**, damit gelangst du zu einem Dialog, über den du deine „Run Configurations“ verwalten kannst.



Klick auf das Tab „Arguments“, und gib unter „Program arguments“ deinen Namen ein.

Da gibt es noch ein anderes Textfeld „VM arguments“. Kann ich da JVM-Parameter übergeben?

Ja, genau. Aber das müssen wir im Moment nicht. Klick also direkt auf „Run“, und dein Programm sollte wieder „Hallo Schrödinger“ ausgeben. Teste dein Programm jetzt mal mit anderen Namen!

Möglichkeit 2: Eingabeströme

Startparameter müssen – wie der Name schon sagt – bereits beim Start deinem Programm übergeben werden. Eine wirkliche Interaktion kriegst du so nicht hin. Dazu brauchst du etwas anderes. Die Zaubermethode heißt: **`System.in.read()`**. Damit kannst du einzelne Zeichen von der Kommandozeile einlesen. Hier ein kleines Beispiel:

```
package de.galileocomputing.schroedinger.java.kapitel01;
import java.io.IOException; *1

public static void main(String[] args) throws IOException*2 {
  System.out.println("Hallo " + args[0]);
  System.out.println("Gib einen Wert ein:");
  int wert = System.in.read(); *3
  System.out.println(wert);
}
```

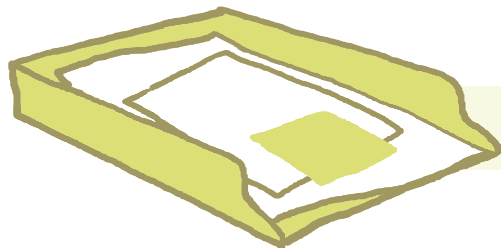
*1 Mit dem **`import`**-Schlüsselwort kannst du (unter anderem) Klassen importieren. Hier importieren wir die Klasse **`IOException`**, die grob gesagt dazu notwendig ist, um auf Fehler bei der Eingabe reagieren zu können.

*2 Das hier bedeutet: Diese Methode könnte einen Fehler vom Typ IO Exception werfen. Mehr dazu in Kapitel 9.

*3 Der Rückgabewert von **`System.in.read()`** ist immer eine ganze Zahl (das bedeutet das Schlüsselwort **`int`** für Integer), auch wenn du einen Buchstaben eingibst. Eigentlich ist es sogar nur ein Byte. Wie du daraus wieder Buchstaben machst, und mehr dazu, im nächsten Kapitel.

Streams

Die Kommunikation zwischen dir und deinem Programm wird in Java über Streams (Datenströme) gehandhabt. Das kannst du dir wie einen Kanal vorstellen, in dem einzelne Bytes hin- und hergesendet werden. Auf der einen Seite des Eingabekanals bist du mit deiner Tastatur, der die Daten eingibst, auf der anderen Seite ist dein Programm, das die Daten einliest. Umgekehrt schreibt dein Programm über den Ausgabekanal Daten auf den Bildschirm, wo du sie aus… ähm … ablesen kannst.



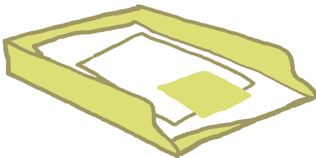
[Ablage]

Was du dir schon mal merken solltest: Es gibt genug Helferklassen, die das Arbeiten mit Streams um einiges vereinfachen und das direkte Arbeiten mit **`System.in.read()`** eigentlich überflüssig machen.

[Zettel]
out, **err** und **in** sind öffentliche Objektinstanzen innerhalb der Klasse **System**. **out** und **err** sind vom Typ **PrintStream**, und **in** ist vom Typ **InputStream**.

Uff! Objektinstanzen? Und wieso „innerhalb“ von **System**? Klingt wie ein Labor, in das man nur mit Ausweis reindarf.

Im Gegenteil. Das heißt, es ist für alle da!
Öffentliche Objektinstanz innerhalb der Klasse **System** heißt, es sind Objekte, die sowieso immer da sind und die du einfach mitbenutzen kannst. Und es heißt, dass du drankommst, indem du **System** schreibst, dann einen Punkt und dann den Namen, also zum Beispiel **out**. Dann hast du den Kanal für Ausgaben auf die Konsole gefunden.



[Ablage]
Anleitung für den Klempner:
Den Hauptabwasserkanal finden Sie unter **System.out**, die Zuleitung unter **System.in**.
Es gibt in diesem Gebäude noch einen besonderen Abwasserkanal für den Fehlerfall: **System.err**
Alle Kanäle befinden sich in der Klasse **System** und sind über diesen Klassennamen, gefolgt von einem Punkt, ansprechbar.

Objektinstanz	Objekttyp	Bedeutung
System.out	java.io.PrintStream	Standardausgabe
System.err	java.io.PrintStream	Standardfehlerausgabe
System.in	java.io.InputStream	Standardeingabe

Eine Frage hätte ich noch:
Du redest von Streams und Konsolen. Ich will doch Programme schreiben, wo ich Knöpfe drücken und Textfelder verwenden kann. Konsolenprogramme sind doch sowas von 80er!

Ja, aber auch diese grafischen Oberflächen basieren letztendlich auf den hier beschriebenen Basisklassen. Aber keine Sorge, grafische Oberflächen gucken wir uns auch noch an.

Let the game begin – das „World of Warcraft Textadventure“

So, nachdem du allen Erdbewohnern mit deinem Programm von eben „Hallo“ gesagt hast, können wir ja jetzt **richtig** anfangen: Du träumst davon, Spiele-entwickler zu werden? Dann wird es Zeit, deinen ersten Action-Kracher im World-of-Warcraft-Universum zu erstellen: ein gutes altes Textadventure.

Auf geht's!

[Notiz]

Textadventures sind Computerspiele, die du dir wie interaktive Bücher vorstellen kannst. Dem Spieler werden statt Grafiken lediglich Textabschnitte präsentiert. Über Tastatureingaben kann der Spieler Entscheidungen treffen und damit das Spielgeschehen beeinflussen. Textadventures sind absoluter Kult. Und yeah, die sind wirklich 80er!

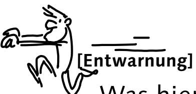
Voll 80er ist ja wohl, das Programm ABZUTIPPEN! Mein Cousin hatte diese Heftchen mit Spielen in Basic und flucht heute noch über die Schwielen an den Fingern ...

[Einfache Aufgabe]

Erstelle eine Klasse mit dem Namen **WoWTextadventure**, die den folgenden Quelltext enthält.

[Notiz]

Falls du dir Tipparbeit ersparen willst: Im Downloadbereich zum Buch findest du alle Quelltexte aus diesem Buch.



[Entwarnung]

Was hier an Quelltext folgt, musst du jetzt noch nicht bis ins kleinste Detail verstehen. Dazu haben wir ja schließlich die nächsten siebenhundertnochwas Seiten (und wahrscheinlich genauso viele Tassen Kaffee) Zeit. Apropos Kaffee. Den hab ich ja ganz vergessen. Ist wahrscheinlich schon kalt geworden.

Ich erklär dir kurz, wie ich es gemacht habe. Noch wird dir das alles wahrscheinlich wie Zauberei vorkommen, aber hey: **Auch Java-Entwickler kochen nur mit Wasser.**

```
package de.galileocomputing.schroedinger.java.kapitel01; *1

import java.io.BufferedReader; *2
import java.io.IOException; *2
import java.io.InputStreamReader; *2

public class WoWTextadventure {
  public static void main(String[] args) throws IOException {
    System.out.println("Hallo " + args[0] + ", willkommen in der World of ⮐
      Warcraft. Du befindest dich im Dorf Buxelknuxel.");
    System.out.println("Verwende die Tasten 'N', 'O', 'S' und 'W', um dich zu ⮐
      bewegen, und 'I', um einen Blick in dein Inventar zu werfen.");
    System.out.println("Mit 'Q' verlässt du das Spiel.");
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in)); *3
    String befehl;
    while ((befehl = br.readLine()*3) != null) { *4
      switch (befehl.toLowerCase()) { *5
      case "n": *6
        System.out.println("Du gehst nach Norden.");
        break;
      case "o": *6
        System.out.println("Du gehst nach Osten.");
        break;
      case "s": *6
        System.out.println("Du gehst nach Süden.");
        break;
```

*1 Hier ist wieder das Paket, in dem die Klasse liegt.

*2 Diese drei Klassen brauchst du zur Ein- und Ausgabe. `IOException` kennst du ja schon von eben. Dazu kommen noch die zwei Helferklassen `BufferedReader` und `InputStreamReader`, die dir das Arbeiten mit Eingabeströmen erleichtern. Um die drei Klassen zu importieren, könntest du auch alternativ `import java.io.*;` schreiben. Damit würden **alle** Klassen aus dem Paket `java.io` importiert.

*3 Sieht kompliziert aus, ist es aber nicht: Hier verwende ich die zwei Helferklassen, um eine „einfachere Sichtweise" auf den Eingabestrom zu bekommen. Zum Beispiel komme ich über `BufferedReader` an die Methode `readLine()`, mit der ich statt einzelner Zeichen direkt eine ganze Zeile einlesen kann und die auch nicht als Zahlenwerte zurückkommen, sondern als Strings. Also damit ist viel leichter zu arbeiten als mit dem rohen `InputStream`.

*4 `while` startet eine sogenannte Schleife, das heißt, alles, was in den nachfolgenden geschweiften Klammern steht (`{` und `}`) wird so lange ausgeführt, wie die Schleifenbedingung (in den runden Klammern) wahr ist. Die Schleifenbedingung bedeutet: so lange die Nutzereingaben lesen, wie sie irgendeinen gültigen Wert haben.

*5 Hier werte ich die Nutzereingabe aus und verwende die Fallunterscheidung (`switch`), …

```java
    case "w": *6
      System.out.println("Du gehst nach Westen.");
      break;
    case "q": *7
      System.out.println("Willst du wirklich schon aufgeben? Y/N");
      String bestaetigung = br.readLine();
      switch (bestaetigung.toLowerCase()) {
      case "y": *7
        System.out.println("Und Tschüss.");
        System.exit(0);
        break;
      case "n": *7
        System.out.println("Finde ich prima.");
        break;
      }
      break;
    case "": *6
      System.out.println("Du willst gar nichts machen? Das glaube ich nicht.");
      break;
    case "i": *6
      System.out.println("Da du noch nicht die Weisheit der Array-kundigen ⮐
        Sammler erlangt hast, befindet sich in deinem Inventar nur ein ⮐
        einziger Gegenstand: ein Holzschwert.");
          break;
    default: *8
      System.err.println("Das verstehe ich nicht."); *8
    }
  }
 }
}
```

*6 … um je nach Nutzereingabe eine entsprechende Meldung auf die Konsole auszugeben:

*7 Mit Eingabe des Buchstabens „q“ und anschließender Bestätigung mit „y“ geben wir dem Nutzer zum Beispiel die Möglichkeit, das Spiel zu verlassen. Mit „n“ bleibt er im Spiel.

[Notiz]

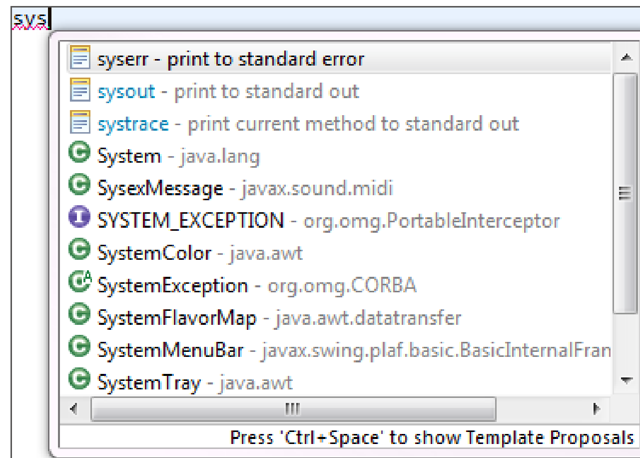
Case ist englisch und heißt „Fall“ wie in „falls“.

*8 Für alle Zeichen, die unser Spiel nicht versteht, geben wir eine Fehlermeldung aus.

Das hätte ich mir jetzt fast gedacht …

Hier noch ein paar Tipps, die dir das Abtippen des Quelltextes erleichtern:

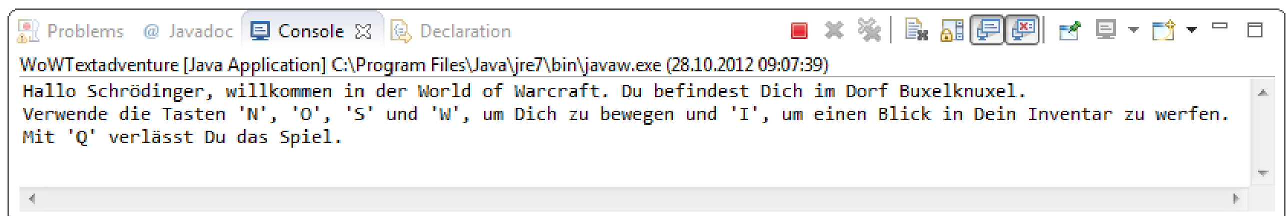
- Anstatt immer **`System.out.println()`** zu schreiben, tippe einfach nur die Zeichen „sys", und drücke dann **`Strg`** und die Leertaste. Diese Tastenkombination öffnet ein kleines Menü zur **Codevervollständigung** direkt unter dem Mauszeiger und zeigt dir die Begriffe, die mit den eingegebenen Zeichen beginnen, in diesem Fall also mit „sys". Daraus kannst du dir einfach den entsprechenden Codeschnipsel auswählen, den Eclipse für dich generieren soll.



Codevervollständigung in Eclipse

- Auf die gleiche Weise kannst du dir auch direkt das Codegerüst für die **`while`**-Schleife oder das **`switch`**-Statement generieren lassen, indem du statt „sys" einfach zuerst „while" oder „switch" eintippst. Generell kannst du hier beliebige Zeichen eingeben, und Eclipse sucht alles, was es dazu gibt.

Hey, wenn ich „syso" eintippe, wird sogar direkt **`System.out.println()`** *in den Editor geschrieben! Cool, mit deinen Tipps hat's gar nicht so lange gedauert.*



Dein Spiel in Action!

[Erledigt!]

Und jetzt viel Spaß mit dem Programm. Erkunde ein bisschen die Gegend. Falls du vergessen hast, wie es funktioniert, hier noch eine kleine Anleitung:

WoW Textadventure
Kurzanleitung

```
n: Nach Norden gehen
o: Nach Osten gehen
s: Nach Süden gehen
w: Nach Westen gehen
i: Inventar öffnen
q: Spiel beenden
```

[Notiz]

Du kannst alle Quelltexte aus dem Buch auch im Bonusbereich unter www.rheinwerk-verlag.de/4975 herunterladen.

Historie von Java

So, Schrödi, leg die Beine hoch, zum Schluss dieses Kapitels gibt's noch ein ganz klein bisschen Allgemeinbildung. Werfen wir einen kurzen Blick **auf die Historie von Java**. Ich mach's auch nicht zu lang, versprochen.

Du hast mir zwar bisher weder deinen Vornamen noch dein genaues Alter verraten, aber eins kann ich dir versichern: Viel älter als Java bist du wahrscheinlich auch nicht. Denn wenn man es genau nimmt, ist das schon Mitte zwanzig. Ja, schon so alt! Denn bereits in den frühen 90ern beschäftigten sich einige wirklich helle Köpfe der Firma **Sun Microsystems** – aus Unzufriedenheit über Sprachen wie C – mit der Entwicklung **einer neuen objektorientierten Programmiersprache** (damals noch unter dem Decknamen **Oak**).

So ziemlich alle waren schnell davon begeistert (na ja, zumindest alle, die sich damals schon mit Programmierung beschäftigt haben und die ersten Entwürfe zu Gesicht bekamen), und nach einigem Hin und Her wurde **1996 das erste JDK** als **Open-Source** veröffentlicht. Von da an hatte Java dann auch seinen richtigen Namen: Java eben, Version: Oak.

[Zettel]
James Gosling gilt als Erfinder von Java, auch wenn er nicht alleine daran gearbeitet hat.

[Zettel]
Java ist übrigens auch der Name einer Insel, das hast du bestimmt schon mal gehört. Gerüchten zufolge kam man auf den Namen Java, weil die von den Entwicklern bevorzugte Kaffee-Sorte von eben dieser Insel stammt. Entwickler sind schon ein lustiges Volk, oder?

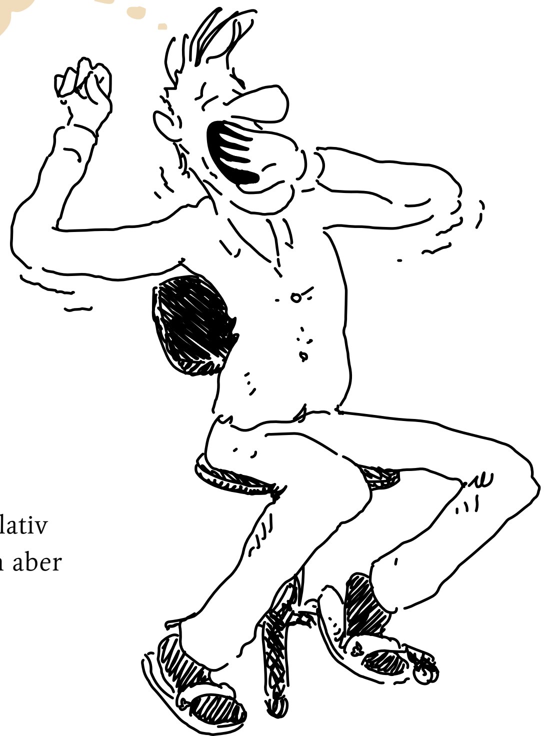
In den folgenden Jahren wurden dann weitere Versionen veröffentlicht und die Sprache nach und nach um verschiedene Features erweitert. Erst Version 1.1, dann Version 1.2, dann 1.3, 1.4 und dann ... 5.0.

Von Version 1.4 zu Version 5.0? Fehlt da nicht was?

Das hatte Marketinggründe. Man wollte damit ausdrücken, dass jede neue Version auch einen großen Schritt mit vielen neuen Features bedeutet. Und da machte es sich nun mal nicht so gut, dass man schon seit Jahren nur auf der Version 1 rumrödelte.

Was die Features angeht, wurde mit Version 5.0 auch ordentlich was draufgelegt. Und auch wenn es mir – und wahrscheinlich vielen anderen Entwicklern – wie gestern vorkommt: Das ist mittlerweile schon über 15 Jahre her.

Ach, jetzt wird es nostalgisch ...



Genauer gesagt war Ende September 2004. Knapp zwei Jahre später, im Dezember 2006, wurde dann Java 6 veröffentlicht. Dann fast fünf Jahre später die Version 7, dann knapp drei Jahre später die Version 8 und dann – mehr als drei Jahre später, im September 2017– die Version 9 ...

Ich dachte, du wolltest es nicht lang machen?!

Geduld! Worauf ich hinaus will: Du siehst, es dauerte relativ lange, bis eine neue Version veröffentlicht wurde. Das hat sich aber mit Version 10 grundlegend geändert. Java 10 wurde nämlich bereits ein halbes Jahr nach Version 9 veröffentlicht.

Also, ähm ... März 2018!

Richtig. Und das ging mit den Versionen 11, 12 und 13 so weiter. Die Version 13 wurde im September 2019 veröffentlicht.

Alle halbe Jahre? Da komme ich ja mit dem Lesen gar nicht hinterher.

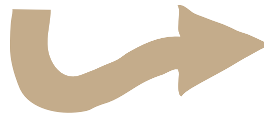
Genau das heißt es. Genauer gesagt, dass jedes halbe Jahr eine neue Version erscheint. Was das Lernen angeht, kein Grund zur Sorge: Grundlagen werden sich nicht ändern, und auch grundlegende Features, die weitreichende Auswirkungen auf Programmierstil oder Programmiertechniken haben, sind eher selten.

Puh! Aber warum denn überhaupt das Ganze mit zwei Versionen im Jahr?

Nun, bei Oracle, die 2009 Java von Sun Microsystems gekauft hatten, fand man die drei Jahre und mehr zwischen zwei Versionen zu lang und wollte einen geregelteren Versionszyklus etablieren. Das ist übrigens nichts Unübliches in der Software-Entwicklung: Der Browser Google Chrome, die Programmiersprache Go oder die JavaScript-Laufzeitumgebung Node.js haben auch alle sehr kurze Release-Zyklen. Du kannst dich also freuen, dass es auch für Java zweimal im Jahr eine neue Version gibt.

Mmh, meint es das jetzt ernst?

So, Schrödinger. Damit wären wir fürs Erste mit der Java-Historie durch. Damit du alles schön auf einen Blick hast, hier nochmal in einer Tabelle:



Version	Code-Name	Veröffentlichungsdatum
1.0	Oak	Januar 1996
1.1	–	Februar 1997
1.2	Playground	Dezember 1998
1.3	Kestrel	Mai 2000
1.4	Merlin	Februar 2002
5.0	Tiger	September 2004
6	Mustang	Dezember 2006
7	Dolphin	Juli 2011
8	–	März 2014
9	–	September 2017
10	–	März 2018
11	–	September 2018
12	–	März 2019
13	–	September 2019
...	...	...

[Zettel]
Wie du in der Tabelle auf einen Blick siehst, hat sich also in der Geschichte von Java einiges getan, sowohl bezüglich der Versionsnummern als auch bezüglich der Code-Namen (denn die gibt es mittlerweile nicht mehr) und erst recht, was die Veröffentlichungstermine angeht.

Was solltest du aus diesem Kapitel mitnehmen? Hier nochmal alles auf einen Blick:

- Damit Java-Programme **plattformunabhängig** sein können, laufen sie innerhalb einer **virtuellen Maschine**, der **Java Virtual Machine** (JVM), die in Form des **Java Runtime Environments** (JRE) kostenlos **für alle gängigen Betriebssysteme** zur Verfügung steht.
- Wenn du Java-Programme **entwickeln** möchtest, brauchst du das **Java Development Kit** (JDK).
- Das JDK gibt es direkt in drei Ausführungen:
  - als **Standard** Edition (SE)
  - als **Mobile** Edition (ME)
  - als **Enterprise** Edition (EE)
- Du kannst Java-Programme mit einem Texteditor schreiben und alles auf Kommandozeile **per Hand kompilieren**. Besser aber verwendest du dazu eine **Entwicklungsumgebung** (Integrated Development Environment – IDE), zum Beispiel Eclipse, Netbeans, IntelliJ oder JBuilder. Mit allen kannst du Java-Programme entwickeln und laufen lassen.
- Auf der Kommandozeile würdest du dafür die Befehle **`javac`** (zum Kompilieren) und **`java`** (zum Laufenlassen) verwenden.
- Analog spricht man auch von **Compiletime** (wenn der Compiler rödelt) und von **Runtime** (wenn die JVM rödelt).
- Alles, was du in Java programmierst, ist in sogenannten **Klassen** strukturiert. Der Einstiegspunkt eines Java-Programms ist eine Klasse, die über eine **`main`**-Methode verfügt.
- Um deinen Quelltext **noch besser zu strukturieren**, solltest du deine Klassen wiederum in **Pakete** (Packages) packen, wobei du hier am besten – um Konflikte zu vermeiden – einen weltweit eindeutigen Paketnamen wählst.

[Belohnung/Lösung]

Herzlichen Glückwunsch, Schrödinger! Du hast dir deinen ersten Ausrüstungsgegenstand verdient: **die Java-Landkarte**. Sie wird dir im Dschungel der verschiedenen Java-Versionen, JDKs, JREs und IDEs den Weg weisen.

— ZWEI —

Variablen und grundlegende Datentypen

Alles eine Typfrage

Schrödinger lernt verschiedene grundlegende Datentypen kennen, definiert seine ersten Variablen und sucht ihnen Namen: Erlaubt müssen sie sein, und etwas ausdrücken sollen sie auch. Dazu gibt es sogar „Best Practices". Nur gut, dass nicht auch noch das Standesamt mitredet.

Variablen und Datentypen

Damit deine Programme überhaupt irgendetwas machen können, brauchen sie Daten, auf denen sie arbeiten. Und damit du Daten überhaupt anlegen kannst, brauchst du etwas, das die Daten (zumindest temporär) speichern kann: Du brauchst **Variablen**. Und damit du außerdem festlegen kannst, welche Art von Daten das sind, ist noch etwas nötig: **Datentypen**. Wenn du in Java nämlich eine Variable anlegen (sprich deklarieren) willst, musst du auch immer den Datentyp dieser Variablen angeben.

[Hintergrundinfo]

Eine ganz wichtige Sache, die du dir gleich zu Anfang merken kannst, ist: Java ist eine sogenannte **statisch typisierte Programmiersprache**. Das bedeutet, dass **jede Variable einen festen Typ hat** (deswegen typisiert) und dieser **Typ bereits zur Übersetzungszeit bekannt** ist (deswegen statisch).
Das kannst du prima im Kollegenkreis fallen lassen oder auf der nächsten LAN-Party.

Eine Variablen-Deklaration hat folgenden Aufbau:

```
Datentyp Variablenname = Wert;
```

Hierbei musst du zwei Dinge beachten:

1. **Der Wert**, den du einer Variablen überhaupt geben kannst, **hängt von dem jeweiligen Datentyp ab**.
2. Du kannst einer Variablen **nicht beliebige Namen geben**, denn manche Namen sind Bestandteil der Sprache Java und damit reserviert.

Genauer gesagt, beachtet das sogar der Compiler für dich und warnt dich, falls du dich nicht dranhältst. Wenn du es aber selber schon beachtest, wird dich der Compiler nicht so häufig nerven.

[Zettel]

Du musst einer Variablen nicht immer zwingend direkt einen Wert geben. Wenn du es nicht machst, bekommt die Variable je nach Datentyp einen **Standardwert**.
Es gibt allerdings Stellen, an denen der Compiler verlangt, dass eine Variable explizit von dir einen Wert zugewiesen bekommt. Das ist aber erst viel später für uns interessant.

[Pingel-Modus]

Ganz, ganz streng genommen ist „warnen“ noch etwas freundlich ausgedrückt:
Der Compiler erzeugt nämlich sogar einen Fehler. Die Folge: Dein Programm lässt sich nicht mehr kompilieren.

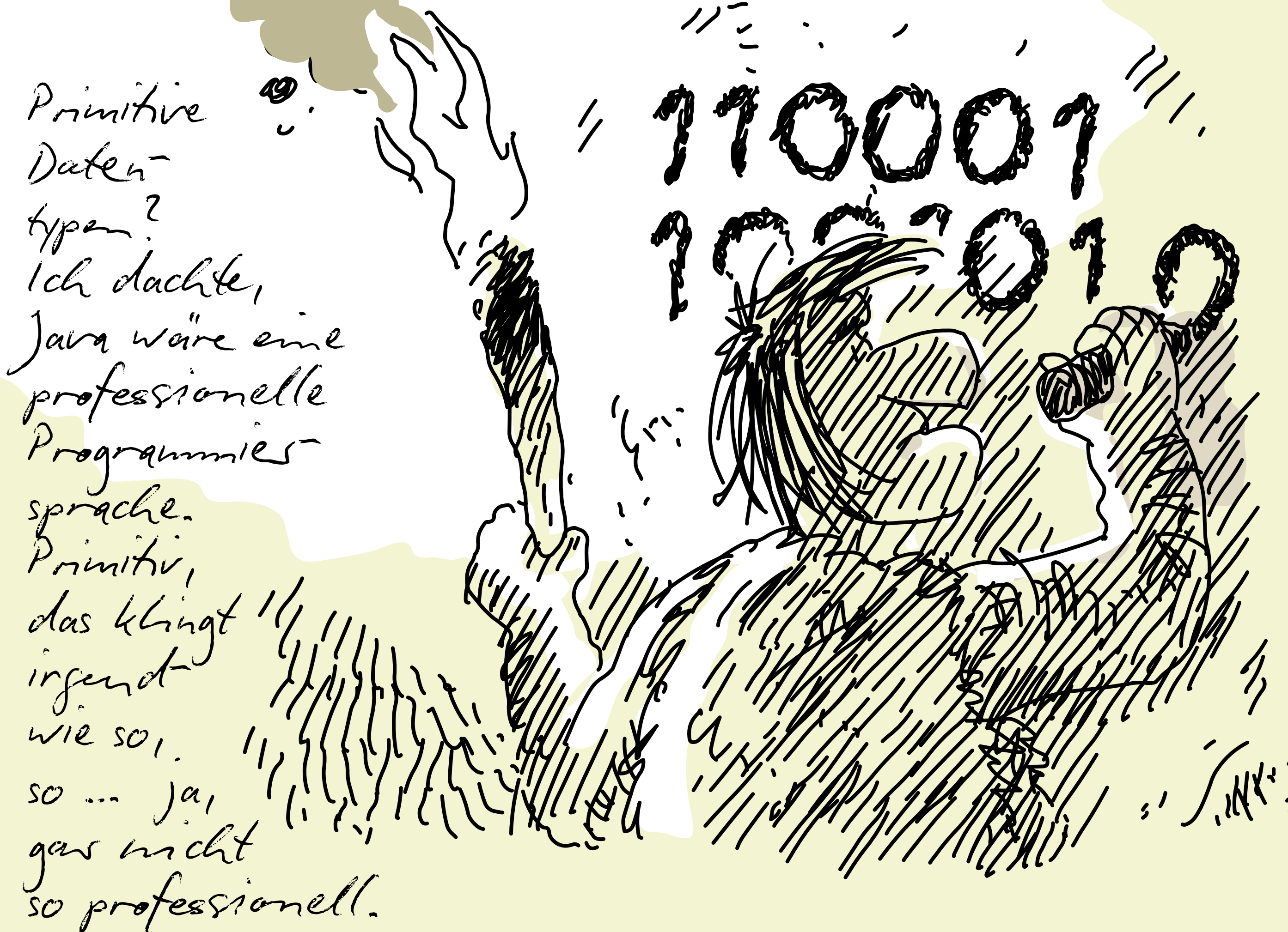
Die Typisierung hat folgenden Vorteil: Fehler, die auftreten, weil einer Variablen ein falscher Wert zugewiesen wird, werden größtenteils bereits zur **Übersetzungszeit** erkannt. In Sprachen ohne strikte Typisierung, wie zum Beispiel JavaScript, ist die

Fehlersuche wesentlich aufwendiger, da man jeder Variablen (fast) jeden beliebigen Wert zuweisen kann.

Okay, okay, ein Zwinger, aber nur zu meinem Besten, verstehe ...

Arten von Datentypen

In Java gibt es zwei Arten von Datentypen: **primitive Datentypen** und **Referenzdatentypen**. Wir fangen mit den primitiven Datentypen an und heben uns die Referenztypen für ein späteres Kapitel auf.



Der Name ist in der Tat etwas irreführend. Aber keine Sorge, Java ist eine professionelle Programmiersprache und, wie du vielleicht weißt, eine, die viel Wert auf **Objektorientierung** legt. Und aus diesem Grund sind in Java auch viele Dinge als (mehr oder weniger **komplexe**) Objekte implementiert. Aber eben manche Dinge auch nicht, wie zum Beispiel **Zahlen**, **Zeichen** und **Wahrheitswerte**. Und die nennt man dann primitive Datentypen oder auch **elementare Datentypen** oder **Basisdatentypen**. Letztendlich sind das also die Dinge, aus denen Objekte zusammengebaut werden.

Können wir Zahlen, bitte?

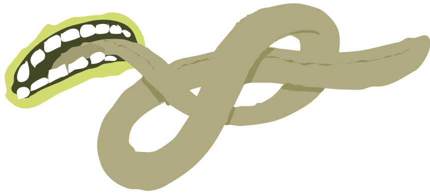
Fangen wir mit den Basisdatentypen für Zahlen an. In Java gibt es sechs verschiedene Typen dafür: **`byte`**, **`short`**, **`int`** und **`long`** für Zahlen **ohne Nachkommastellen (Ganzzahlen)**, **`float`** und **`double`** für Zahlen **mit Nachkommastellen (Fließkommazahlen)**.

Ganzzahlen und Fließkommazahlen, okay.
Aber wieso gleich so viele?

Die verschiedenen Typen geben an, **wie viel Speicherplatz** für die jeweilige Variable **bereitgestellt wird**. Dabei gilt, dass du mit einem **`byte`** weniger Zahlen darstellen kannst als mit einem **`short`**, mit einem **`short`** weniger als mit einem **`int`** und mit einem **`int`** weniger als mit einem **`long`**. Der **Wertebereich** ist also bei **`byte`** am kleinsten und bei **`long`** am größten. Und bei den Fließkommazahlen ist der Wertebereich von **`float`** kleiner als der von **`double`**.

Hier siehst du die Anzahl der Bits (und damit die Größe des Speicherplatzes), die für die einzelnen Zahlentypen vorgesehen ist, und siehst, was jeweils der Minimum-Wert und was der Maximum-Wert ist:

Datentyp	Bit	Minimum	Maximum
`byte`	8-Bit	-128	+127
`short`	16-Bit	-32768	+32767
`int`	32-Bit	-2147483648	+2147483647
`long`	64-Bit	-9223372036854775808	+9223372036854775807
`float`	32-Bit	$-3.4 * (10^{38})$	$+3.4 * (10^{38})$
`double`	64-Bit	$-1.7 * (10^{308})$	$+1.7 * (10^{308})$



Oh, Mann, das kann ich teilweise gar nicht aussprechen. Muss ich mir das jetzt alles merken?

Nein, musst du nicht. Nicht alles zumindest. Minimum und Maximum zum Beispiel nicht, da zeige ich dir nachher was Besseres. Was du dir aber merken solltest, ist die **Reihenfolge der Datentypen**. Und da ich weiß, dass deine Freundin einen Faible für Schuhe hat, hier eine wunderbare Analogie, die du dir leicht merken kannst:
Stell dir die verschiedenen Zahlentypen einfach als Schuhe vor: **`byte`** ist wie ein kleiner Halbstiefel, da passt eher wenig rein, **`short`** wie eine Stiefelette, da passt schon mehr rein, **`int`** wie ein richtiger Stiefel, in den noch mehr reinpasst, und **`long`** … na ja … du kannst es dir denken, da passt extreeeeem viel rein. **`float`** und **`double`** passen wegen der Nachkommastellen nicht so recht in diese Analogie, aber der **`double`**-Stiefel wäre auf jeden Fall größer als der **`float`**-Stiefel.

Moment, Moment, SOLCHE Schuhe hat meine Freundin aber nicht!

`byte` **`short`** **`int`** **`long`**

Okay, okay, aber du musst zugeben, DAS Beispiel vergisst du jetzt nicht so schnell.

Nee, hast recht, kann man sich gut merken:

- *erst der* `byte`*-Kurzstiefel*
- *dann die* `short`*ellete*
- *dann der W-*`int`*-erstiefel*
- *und dann der ganz extreme* `long`*ogo-Stiefel*

X Rotes „X“ = falscher Code

Schrödi, sehr gut, ich sehe, wir haben den gleichen Humor. Hier mal ein paar Beispiele, wie du Stief … Quatsch … Zahlen anlegen kannst:

```
byte b = -128;   *1
byte b2 = 128;   *2 X
short s = 6000;  *3
int i = 23456;   *4
```

*1 Hier wird ein **byte** mit dem kleinstmöglichen Wert erstellt.

*2 Achtung, das hier **kompiliert nicht**, weil (im Gegensatz zu -128) nicht 128 das Maximum für ein **byte** ist, **sondern 127**, denn ein Platz ist für die **Null** (0) reserviert.

*3 Hier wird ein **short** erzeugt …

*4 … und hier ein **int**. Alles ganz normal und im jeweils gültigen Wertebereich.

Wenn du ein **long** anlegen möchtest, musst du das explizit angeben, indem du entweder ein **kleines „l“** oder ein **großes „L“** an die Zahl hängst. Aus Gründen der Lesbarkeit empfehle ich dir aber, ein großes „L“ zu verwenden, das kleine „l“ kann doch zu schnell mit einer 1 verwechselt werden.

```
long l = 23456782345678;    *5 X
long l2 = 23456782345678L;  *6
```

*5 Hier gibt's wieder einen **Compilerfehler**, weil **der Standarddatentyp** für alle Ganzzahlen **int** ist. Der angegebene Wert passt aber nicht in den Wertebereich von **int**, sondern nur in den von **long**.

*6 Du musst also extra angeben, dass du ein **long** haben möchtest.

*Und wieso muss ich bei **byte**, **short** und **int** nicht ein „b“, „s“ oder „i“ an die Zahl hängen?*

Gute Frage, simple Antwort:

Weil es so in der Java-Spezifikation entschieden ist. Es gibt allerdings auch schon Anregungen in Foren, die entsprechende Suffixe zumindest für **byte** und **short** vorschlagen. Im Moment ist eine Umsetzung aber nicht vorgesehen.

Bei den Fließkommazahlen ist **double** der Standarddatentyp. Wenn du ein **float** haben möchtest, musst du entweder ein **kleines „f“** oder ein **großes „F“** an die Fließkommazahl hängen.

```
double d = 4.0;       *7
float f = 4.0;        *8 X
float f2 = 4.0F;      *9
double d2 = 4.0D;     *10
int a, b, c, d = 5;   *11
```

*7 Bei den Fließkommazahlen ist **standardmäßig** jede Zahl vom größtmöglichen Typ, dem Typ **double**.

*8 Das heißt, auch hier meckert der **Compiler**, da ein **double** nicht in ein **float** umgewandelt werden kann.

Lesbarkeit von langen Zahlen

Um die Lesbarkeit der Zahlen zu erhöhen, ist es seit Java 7 sogar möglich, an (fast) beliebigen Stellen innerhalb einer Zahl einen Unterstrich (_) einzufügen. Nimm als Beispiel Kreditkartennummern: Die haben 16 Stellen, und das wird schnell etwas unübersichtlich mit der normalen Schreibweise:

```
long kreditKartenNummer = 2345234523452345L;
```

Die Version mit Unterstrichen lässt sich dagegen schon viel besser lesen:

```
long kreditKartenNummer = 2345_2345_2345_2345L;
```

Super lesbar. Gut, dass das nicht meine Kartennummer ist.

Intern werden die beiden Zahlen aber exakt gleich dargestellt, der einzige Vorteil besteht darin, dass du die zweite Variante als Entwickler besser lesen kannst.

[Achtung]

An folgenden Stellen darfst du den Unterstrich nicht verwenden:

- direkt neben einem Punkt (bei **float** und **double**)
- vor dem „l“, „L“, „f“, „F“, „d“ und „D“ bei **long**, **float** und **double**
- am Anfang oder Ende einer Zahl

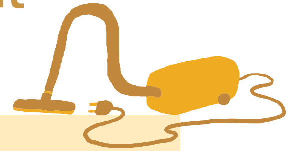
*10 Bei **double** ist ein kleines „d“ oder ein großes „D“ optional.

*9 Du musst auch hier wieder **explizit angeben**, dass die Fließkommazahl vom Typ **float** sein soll.

*11 Du kannst auch direkt **mehrere Variablen auf einmal** anlegen. Voraussetzung ist nur, dass sie den gleichen Typ haben.

Zahlensuppe

Wollen wir mal schauen, ob du gut aufgepasst hast:



[Einfache Aufgabe]

Ich habe hier ein paar Deklarationen von Zahlenvariablen vorbereitet, aber nicht alle sind erlaubt. Prüfe, was erlaubt ist, und spiele Compiler: Wenn du einen Fehler findest, springst du auf, drehst dich im Kreis, pustest die Backen auf und rufst: „Error, Error!!!!“

	Okay	Fehler
1. `short einShort = 2343434;`	☐	☐
2. `long einLong = 1234_5678_5678;`	☐	☐
3. `int einInt = 2244__________2424;`	☐	☐
4. `double einDouble = 2D;`	☐	☐
5. `double nochEinDouble = 2.0_0_D;`	☐	☐
6. `float einFloat = 2.00000000F;`	☐	☐
7. `float nochEinFloat = 02.0F;`	☐	☐

Hier die LÖSUNGEN:

1. **Fehler!** Der Wert „2343434“ liegt **außerhalb des Wertebereichs** von **`short`**.
2. **Fehler!** Standardmäßig ist der Wert vom Typ **`int`**. Damit er ein **`long`** wird, musst du ein kleines „l“ oder ein großes „L“ anhängen.
3. **Alles prima!** Es können **beliebig viele Unterstriche** hintereinander verwendet werden. Beachte: Der Wert, der in der Variablen gespeichert wird, enthält keine Unterstriche und lautet „22442424“.
4. **Okay!** Fließkommazahlen **müssen keinen Punkt** enthalten.
5. **Fehler!** Der Unterstrich darf nicht vor dem „D“ vorkommen.
6. **Alles prima!** Das ist ein ganz normaler **`float`**-Wert.
7. **Erlaubt!** Keine der Nullen am Anfang einer Fließkommazahl werden berücksichtigt.

Binär, oktal, dezimal und hexadezimal

Schrödinger hat die Katze eingesperrt, weil er nicht rechnen kann, wenn jemand dazwischenmiaut. Jetzt hat er es eilig.

Wir können loslegen. Was gibt's?

365 Tage gibt's dieses Jahr. Jetzt, wo du Ruhe hast, kannst du die mal in binär, oktal und in hexadezimal umrechnen.

Sehe ich aus, als spräche ich fließend Hex? Sonst noch was?

Keine Sorge, die Umrechnung musst du als Java-Entwickler nicht im Kopf beherrschen. Es gibt im Internet genug Umrechnungstools. Allerdings hätte ich gerne, dass du mir die Werte dann korrekt in Java-Notation aufschreibst.

Du hast mir aber noch gar nicht verraten, wie man das macht.

Oh, pardon. Das geht mit einem Präfix:

Präfix	Stellenwertsystem	Basis	Beispiele
0b / 0B	Binärsystem	2	0B00011000
0	Oktalsystem	8	030
	Dezimalsystem	10	24
0x / 0X	Hexadezimalsystem	16	0X18

Null oder Oh?

Null.

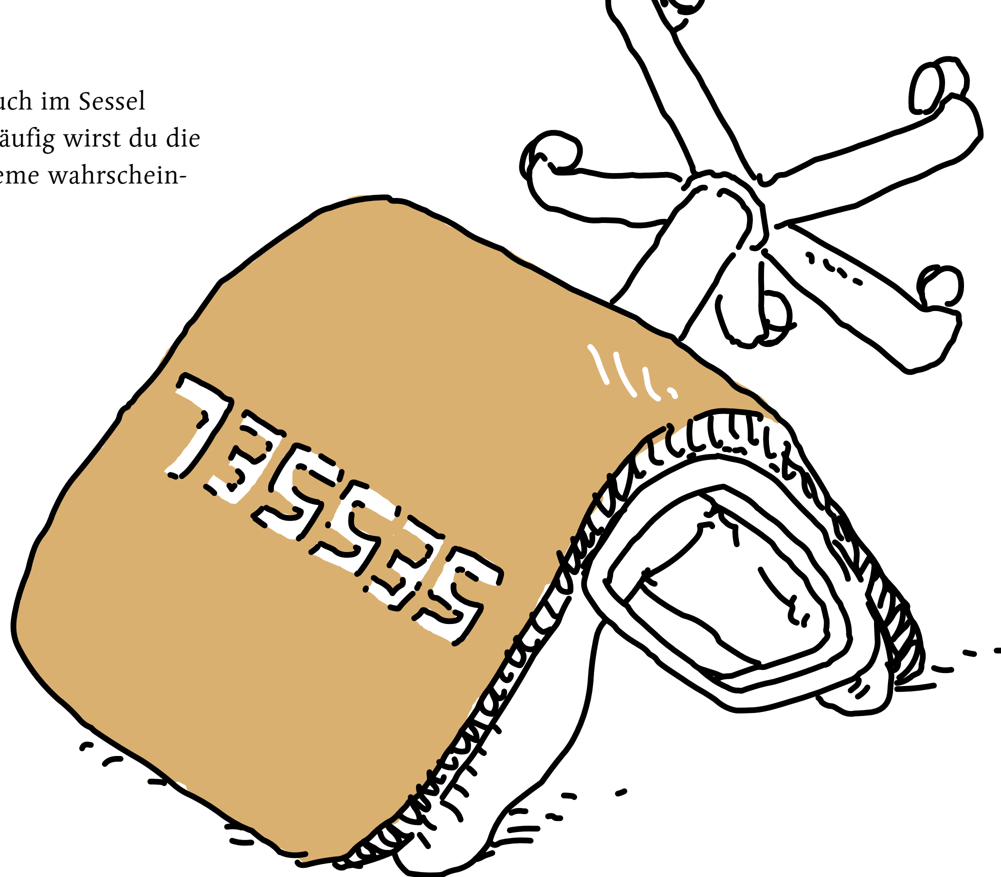
Über das Präfix bestimmst du, welches System verwendet werden soll: Zum Beispiel stellst du beim Hexadezimalsystem der Zahl ein **0x** oder **0X** davor. Binär geht das übrigens erst seit Java 7.

Okay, kapiert. Also hier jetzt deine dreihundertfünfundsechzig, weil du's bist:

```
int oktalZahl = 0555;
int binaerZahl = 0B101101101;
int dezimalZahl = 365;
int hexZahl = 0X16D;
```

Prima, das war ja auch nicht so schwer. Wenn du noch Lust hast und die Katze noch nicht aus dem Sack ist, kannst du ja mal eine Variable für die Zahl 735535 in den verschiedenen Systemen deklarieren. Wenn nicht, auch

nicht schlimm, kannst auch im Sessel sitzenbleiben, denn so häufig wirst du die anderen Stellenwertsysteme wahrscheinlich eh nicht brauchen.



[Zettel]

Ein Anwendungsfall, bei dem zum Beispiel **Binärzahlen** von Vorteil sind, ist die sogenannte **Bitmaskierung**. Jetzt und hier habe ich leider nicht die Zeit und den Platz, dir das zu erklären. Zum Glück, denn wirklich spannend ist das nun wirklich nicht.

Danke, ich bleibe im Sessel. Na, dann kann ich die Katze ja wieder rauslassen.

Lösung:

0b10110011100100101111, 02634457, 735535, 0xB392F

Variablennamen

Am Beispiel von Zahlen hast du jetzt gesehen, dass der Wert, den du einer Variablen geben kannst, vom jeweiligen Datentyp abhängt. Bevor wir uns noch weitere Datentypen für Zeichen und Wahrheitswerte angucken, noch schnell ein paar Worte zu gültigen Variablennamen. Denn erstens sind **nicht alle Zeichen** innerhalb eines Variablennamens **erlaubt**, und zweitens gehört es zum guten Ton unter Entwicklern, **gewisse Regeln bei der Namensgebung** einzuhalten.

Variablennamen können Buchstaben, Zahlen, das Dollar-Zeichen (**$**) oder den Unterstrich (**_**) enthalten und mit allem davon – mit Ausnahme von Zahlen – auch beginnen. Du solltest allerdings auch den **guten Stil** eines wahren Java-Entwicklers wahren und versuchen, **aussagekräftige Variablennamen** zu wählen und diese **immer** mit einem Buchstaben anfangen zu lassen. Außerdem ist es durchaus guter Stil, die sogenannte **Camel-Case-Schreibweise** zu verwenden: Besteht ein Variablenname aus mehreren Wörtern, werden dabei die Anfangsbuchstaben der einzelnen Wörter (außer der des ersten Wortes) großgeschrieben.

Hier ein paar Beispiele:

```
int zahl = 4; *1
long ganzLangeZahl = 234567; *2
int 5tesBeispiel = 5; *3 X
char buchstabe-b = 'b'; *4 X
```

*1 Besteht ein Variablenname nur aus einem Wort, ist es guter Stil, den Namen kleinzuschreiben.

*2 Bei Variablennamen mit mehr als einem Wort werden **alle weiteren Worte** großgeschrieben.

*3 Variablennamen können **nicht mit einer Zahl** beginnen.

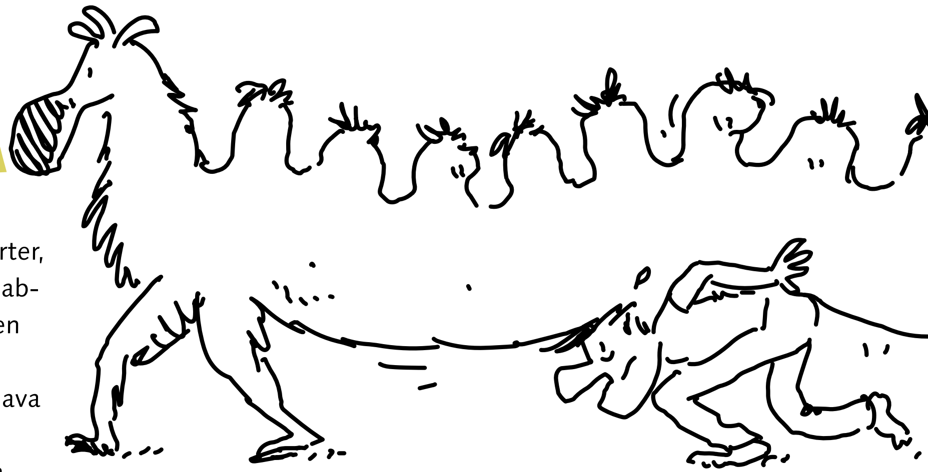
*4 Variablennamen dürfen auch **keine Bindestriche** enthalten.

Moment, Moment, was soll denn dieses **char** *da?*

Ups, das ist mir da wohl so reingerutscht. Mit **char** definierst du Zeichen und Buchstaben. Das gucken wir uns gleich an. An dem Variablennamen ändert das aber nichts, der wäre auch für **int**, **short** und die anderen nicht gültig.

[Zettel]
Ausnahmen bestätigen auch hier die Regel: Es gibt natürlich Fälle, in denen es auch mal reicht, eine Variable nur **i** oder **j** zu benennen (Zählervariablen in Schleifen zum Beispiel, die du im nächsten Kapitel kennenlernst).

[Zettel]
Camel-Case hat übrigens nichts mit Zigaretten zu tun. Gemeint ist vielmehr, **dassWörterInCamelCaseDieForm-VonSehrLangenKamelenHaben**.



[Achtung]
Es gibt ein paar Wörter, die du nicht als Variablennamen verwenden darfst, da sie zum Sprachumfang von Java gehören. Die Rede ist von sogenannten **Schlüsselwörtern**.

Hier eine Liste dieser **Schlüsselwörter** in Java:

A	abstract assert	E	else enum exports* extends	L	long	S	short static strictfp super switch synchronized
B	boolean break byte	F	final finally float for	M	module*	T	this throw throws to* transient try
C	case catch char class const continue	G	goto	N	native new	U	uses*
D	default do double	I	if implements import instanceof int interface	P	package private protected provides* public	V	void volatile
				R	requires* return	W	while with*

* in Modul-Konfigurationen

Das Schlüsselwort-ABC