edv-buchversand.de

Let's Craft Code!

Wie du deine Minecraft-Welt mit Plugins erweiterst

» Hier geht's direkt zum Buch

DIE LESEPROBE

Diese Leseprobe wird Ihnen von www.edv-buchversand.de zur Verfügung gestellt.

Kapitel 2

Aller Anfang ist leicht: Die ersten Schritte zur Plugin-Entwicklung

Dreimal auf Holz geklopft

Nun kann es also losgehen: Du weißt, was dich in diesem Buch erwartet, und bist sicher schon gespannt, deine ersten Plugins zu programmieren. Dieses Kapitel legt dafür die wichtigsten Grundlagen, was leider auch ein wenig Theorie mit sich bringt. Ich kann dir dafür versprechen: So theoretisch wie in diesem Kapitel wird es danach nicht mehr.

Gleichzeitig warten auch coole praktische Dinge auf dich: Schon in diesem Kapitel wirst du dein erstes Plugin erstellen, das von deinem Server erkannt wird. Davor müssen wir allerdings noch ein paar Dinge erledigen. Zunächst müssen wir einiges auf deinem Computer einrichten: Wir benötigen eine Umgebung für die Programmiersprache, mit der wir im Buch arbeiten, und installieren ein Programm, das uns beim Programmieren unterstützt. Außerdem erstellen wir das erste Plugin und setzen einen Test-Server auf, der direkt auf deinem Rechner läuft und dem du dann im Mehrspielermodus beitreten kannst. Zu dem neu angelegten Plugin erkläre ich dir im Detail, welche Bedeutung verschiedene Dinge im Code haben und welche wichtigen Dateien wir für ein Plugin erstellen müssen. So verstehst du von Anfang an, wie du dich im Code zurechtfindest, und kannst im Laufe des Buches viel einfacher deine eigenen Ideen entwickeln oder neue Plugins erstellen.

2.1 Was ist Java, und warum nutzen wir diese Programmiersprache?

Wenn du noch nie zuvor programmiert hast, kennst du sicher viele Mythen und Legenden rund um Programmiersprachen und Quellcode: Kryptisch aussehender Text, den nur absolute Genies schreiben und verstehen können und der von der einen auf die andere Sekunde über den Bildschirm flackert.

Glücklicherweise ist es viel leichter als das, und wie du schon in diesem Kapitel sehen wirst, kannst auch du Quellcode leichter verstehen und sogar selbst schreiben, als du es dir im Moment noch zutraust. All das ermöglicht uns eine *Programmiersprache*. In den letzten Jahrzehnten und sogar noch im letzten Jahrhundert sind zahlreiche Programmiersprachen entstanden, die alle ein gemeinsames Ziel eint: Uns das Leben beim Entwickeln von Programmen oder auch Minecraft-Plugins leichter zu machen. Dafür schreiben wir den schon oft erwähnten *Quellcode* (kurz *Code* genannt): Das ist Text, der fest vorgeschriebenen Regeln folgt und den wir mit ein wenig Hintergrundwissen und Erfahrung gut lesen und schreiben können. Dabei werden viele Sonderzeichen verwendet, die du in normalen Texten nur selten findest. Im Quellcode haben diese Zeichen aber meist eine besondere Bedeutung, sodass schon das Hinzufügen oder Entfernen nur eines Zeichens ungeahnte Folgen haben kann.

Ein Beispiel für die schnelle Wirkung von Änderungen im Quellcode

Um ein Beispiel für solche Folgen zu nennen: Stell dir vor, dass du mit deinem Plugin die Geschwindigkeit eines Monsters verändern willst. Fügst du aus Versehen (oder auch absichtlich) ein Minuszeichen (–) vor die Zahl der Geschwindigkeit, bekäme die Zahl plötzlich eine völlig andere Bedeutung. Statt einer hohen Zahl wäre der Wert nun auf einmal kleiner als 0, was kein gültiger Geschwindigkeitswert mehr wäre. Solche Fehler schleichen sich schnell ein, hier musst du also aufmerksam sein.

Wenn du Code geschrieben hast, folgt ein wichtiger Prozess: Nun muss der Text, den wir als Menschen verstehen könnten, für den Computer lesbar gemacht werden. Die technischen Details sind an dieser Stelle gar nicht so wichtig, doch stell es dir so vor, dass dein PC am Ende nur mit den Zahlen 0 und 1 arbeitet. Denken wir am besten gar nicht daran, Abfolgen aus 0 und 1 von Hand zu schreiben, sondern verwenden wir den magischen Vorgang des *Kompilierens*. Im Englischen wird dieser Prozess auch *Compiling* genannt, das passende Programm dazu nennt man einen *Compiler*. Wenn der Compiler seine Arbeit getan hat, haben wir ein fertiges Programm, das von unserem PC gelesen werden kann, zusammengebaut.

Nicht jede Sprache hat einen Compiler

Es gibt Programmiersprachen, die andere Technologien als einen Compiler verwenden. Besonders bekannt sind hier sogenannte *Interpreter-Sprachen*, die einen passenden *Interpreter* verwenden. Dieser wandelt den Quellcode nicht in maschinenlesbaren Code um, sondern stellt eine eigene Software bereit, die den Programmcode »live« verstehen kann, während das Programm läuft.

In diesem Buch arbeiten wir mit der Programmiersprache *Java*. Vielleicht hast du schon einmal von Java gehört, denn es ist auch heute, mehr als 30 Jahre nach der Erfindung der Sprache, eine der bekanntesten und meistgenutzten Programmiersprachen der Welt. Viele bekannte Programme wurden in Java entwickelt, so zum Beispiel unser Lieblingsspiel Minecraft (das sagt ja auch der Name *Java* Edition).

Java ist eine besondere Programmiersprache, da sie aus mehreren Teilen besteht, die du beim Entwickeln von Programmen grundlegend verstehen musst.

- *Java Development Kit* (*JDK*): Das JDK ist das Herzstück, um eigene Java-Programme zu entwickeln. Mithilfe des JDK startest du nämlich das Kompilieren des Quellcodes in eine maschinenlesbare Form. Java-Programme werden dabei in eine besondere Maschinenform übersetzt, den sogenannten *Bytecode*. Der Bytecode ist speziell für Java entwickelt worden.
- *Java Runtime Environment* (*JRE*): Das JRE ist laut der deutschen Übersetzung die »Laufzeitumgebung« für Java-Programme. Es enthält verschiedene Java-Dateien, mit denen in Java entwickelte Programme verstanden und ausgeführt werden können. Doch damit das gut funktioniert, ist ein weiterer wichtiger Bestandteil der Java-Familie notwendig.
- *Java Virtual Machine* (*JVM*): Die JVM ist die eigentliche Umgebung, die ein Java-Programm ausführt. Sie versteht den Bytecode, den das JDK erzeugt hat, und wandelt die Bytecode-Befehle so um, dass genau das passiert, was wir programmiert haben. Die JVM auf dem Server würde also dafür sorgen, dass dein entwickeltes Plugin genau so funktioniert, wie du es ursprünglich im Code erdacht hast.

Wie du es dir vielleicht schon denken kannst, benötigen wir für die Entwicklung von Plugins alle drei genannten Bestandteile: Mithilfe des JDK schreiben wir zunächst Plugin-Quellcode, den wir im Anschluss in Bytecode umwandeln. Das JRE hilft uns im Anschluss direkt auf den Server, den Bytecode zu verstehen und so das Plugin testen zu können, indem der Bytecode innerhalb der JVM ausgewertet wird. Glücklicherweise müssen wir nicht alle drei Programme einzeln installieren. Setzen wir für die Ausführung von Java-Programmen das JRE auf, so enthält dieses automatisch die JVM, um Bytecode verstehen zu können. Wenn wir also nur Java-Programme ausführen wollen, dann würde es ausreichen, lediglich das JRE zu installieren. Wir wollen allerdings entwickeln, doch auch hier hilft uns Java: Das JDK enthält nicht nur die notwendigen Entwicklungswerkzeuge wie den Bytecode-Compiler, sondern zugleich das JRE (und dieses enthält wiederum das JDK). Vereinfacht gesagt: Wenn wir das JDK installieren, dann installieren wir das JRE und die JVM automatisch mit. Abbildung 2.1 zeigt grafisch, wie die drei vorgestellten Werkzeuge zusammenhängen.

Warum der Bytecode und die JVM so praktisch für uns sind

Java-Bytecode ist eine wirkliche Besonderheit, da er für unterschiedliche Betriebssysteme gleich ist. Während andere Programme also mehrmals für verschiedenste Plattformen kompiliert werden müssen, werden Java-Programme nur einmalig in Bytecode umgewandelt.

Der wahre Trick liegt hier in der JVM: Sie weiß auf dem jeweiligen Betriebssystem, wie der Bytecode gelesen werden muss, damit das Programm funktioniert. Während der Bytecode also immer gleich bleibt, hat jede Plattform eine eigene JVM, die »weiß«, wie der Bytecode auf dem jeweiligen Betriebssystem gelesen werden muss. Das macht Java zu einer *plattformunabhängigen* Sprache. Das heißt, dass der gleiche Quellcode für verschiedene Plattformen verwendet werden kann. Das siehst du auch gut an Minecraft: Die ursprüngliche Version wurde in Java entwickelt und konnte deshalb direkt an allen PCs gespielt werden, egal, ob Windows, Mac oder Linux als Betriebssystem verwendet wurde.

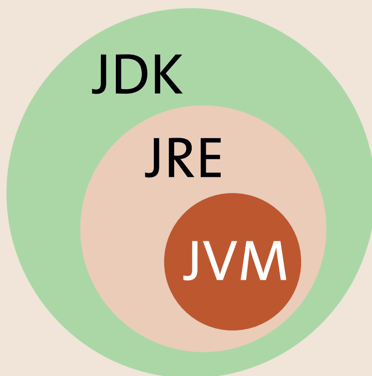

Abbildung 2.1 JDK, JRE und JVM hängen eng zusammen und sind bei der Installation »ineinander verschachtelt«.

Nun, da du weißt, wie JDK, JRE und JVM zusammenhängen, können wir erst einmal das JDK installieren, mit dem wir später Plugins kompilieren und auf unserem Server ausführen können.

2.1.1 Das JDK installieren

Ohne weiter in die technischen Details der Sprache Java zu gehen, starten wir nun mit der ersten Installation: Wir installieren das soeben erklärte JDK auf unserem PC, damit wir im Anschluss Plugins mit Java programmieren und ausführen können.

Wenn du Java schon für Minecraft installiert hast

Wie es der Name verrät, ist für die Minecraft *Java* Edition sowieso eine Java-Installation notwendig. Dennoch schadet es nicht, wenn du die Installation so nachmachst, wie ich es im Buch zeige. So stellst du sicher, dass alle Entwicklungswerkzeuge, die wir für die Plugin-Entwicklung benötigen, tatsächlich auf deinem PC vorhanden sind.

Wie jede (gute) Software ist auch Java bereits durch verschiedene Versionen gegangen, in denen neue Funktionen zur Sprache hinzugefügt oder vorhandene Features verändert wurden. Für dieses Buch verwenden wir die Java-Version 21, die zum Zeitpunkt der Veröffentlichung die neueste Version mit einer Langzeitunterstützung ist. Für die Installation von Java 21 verwenden wir die Plattform *Adoptium*, die das JDK ganz von Zauberhand für dich auf dem PC einrichtet. Die Downloads für die Installationsprogramme findest du unter *https://adoptium.net/de/temurin/releases/?version=21* (siehe Abbildung 2.2).

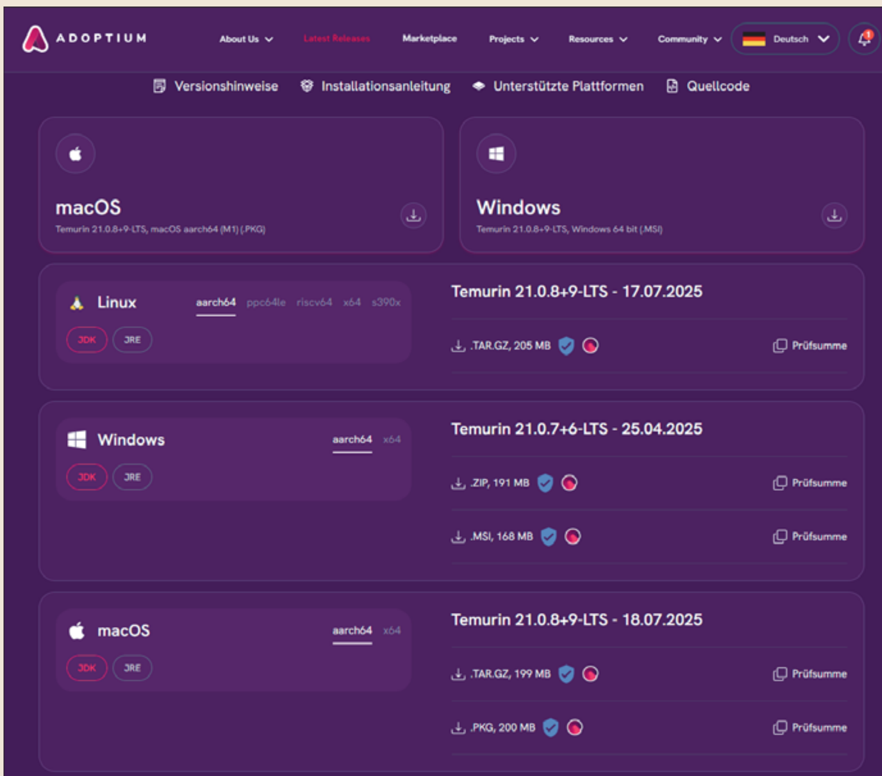

Abbildung 2.2 Bei Adoptium findest du Java-Installationsprogramme für die Betriebssysteme Windows, macOS und Linux.

Neuere Java-Version und Aktualisierung der Adoptium-Website

Wenn du dieses Buch liest, sind vielleicht schon neuere Java-Versionen erhältlich. Wichtig ist, dass du mindestens die Version 21 installierst, da sie die Mindestvoraussetzung ist für die Plugin-Basis, mit der wir im Laufe des Buches arbeiten. Das Verwenden einer höheren Version ist ebenfalls kein Problem und könnte für neuere Minecraft-Versionen sogar notwendig sein.

Außerdem könnte die Adoptium-Website mittlerweile anders aussehen als in Abbildung 2.2. Halte Ausschau nach den Installations-Buttons für dein Betriebssystem.

Auf der linken Seite musst du unter deinem Betriebssystem die Bezeichnung JDK wählen, immerhin wollen wir das gesamte Java-Entwicklungspaket herunterladen und nicht nur die JRE-Laufzeitumgebung. Die Wahl der Systemarchitektur links daneben hängt von dem Prozessor deines Systems ab. Unter Windows ist das normalerweise X64 für herkömmliche Prozessoren von Intel oder AMD. Verwende außerdem am besten den .MSI-Download, der ein Installationsprogramm bereitstellt, das du nur noch ausführen musst. Unter Mac-Rechnern musst du X64 wählen, wenn der Rechner Intel-basiert ist. Solltest du einen Apple-Silicon mit einem M-Chip verwenden, fällt die Wahl stattdessen auf AARCH64. Verwende außerdem den Installationstyp .PKG, der alle notwendigen Einrichtungsschritte für das JDK vornimmt.

Unter Linux ist die Installation ein wenig komplizierter als unter Windows und Mac. Die Wahl fällt normalerweise auf X64, wenn du einen Intel- oder AMD-Prozessor nutzt. Solltest du einen anderen Hersteller verwenden, musst du dich vor der Wahl des Programms über die Systemarchitektur informieren. Das heruntergeladene .TAR.GZ-Verzeichnis kannst du an einem Ort deiner Wahl entpacken. Den Unterordner */bin/* des entpackten Ordners musst du abschließend manuell zu den Umgebungsvariablen des Betriebssystems hinzufügen.

Im Folgenden zeige ich die Installation des JDK beispielhaft mit dem Windows-Installationsprogramm. Unter macOS sieht die Installation allerdings ähnlich aus, sodass du die Schritte parallel verfolgen kannst. Führe dafür zunächst das Installationsprogramm aus. Nach einer kurzen Begrüßung musst du zunächst die Lizenzbedingungen aufmerksam lesen und mit einem Haken bestätigen. Anschließend kannst du auswählen, für welche Accounts deines Rechners die Installation vorgenommen werden soll. Diese drei Schritte siehst du in Abbildung 2.3.

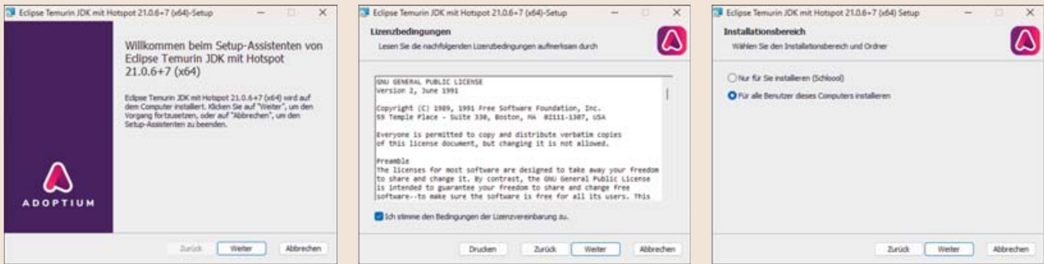

Abbildung 2.3 Die ersten Installationsschritte unter Windows: Zunächst musst du die Lizenzbedingungen akzeptieren und festlegen, für welche Benutzer des Computers die Installation vorgenommen werden soll.

Besonders wichtig ist, dass im System verschiedene Daten gespeichert werden, mit denen genau abgerufen werden kann, wo deine Java-Installation liegt. Dafür ist der nächste Bildschirm da: Hier solltest du neben den drei ersten Einträgen unter JDK MIT HOTSPOT das Icon auf den grauen Kasten ändern, so wie in Abbildung 2.4 gezeigt. Wenn du auf das Symbol klickst, entspricht das der Auswahloption WIRD AUF DER LOKALEN FESTPLATTE INSTALLIERT. Indem du diese Optionen aktivierst, kann dein System Java später bei bestimmten Prozessen sofort finden. Außerdem werden Dateien mit der Endung *.jar* automatisch mit einer Ausführung durch Java verknüpft.

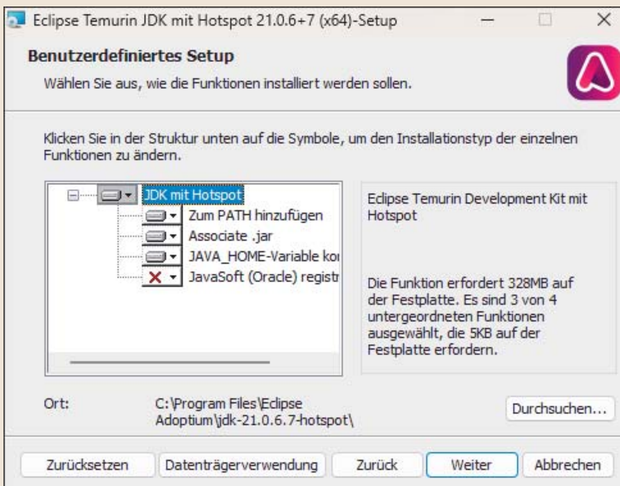

Abbildung 2.4 In der Auswahl sollten die Optionen Zum PATH hinzufügen, Associate .jar und JAVA_HOME-Variable konfigurieren für die Installation aktiviert werden.

Nach dieser Konfiguration ist es geschafft: Die Installation startet, und schon nach wenigen Minuten ist das JDK fertig auf deinem PC installiert (siehe Abbildung 2.5).

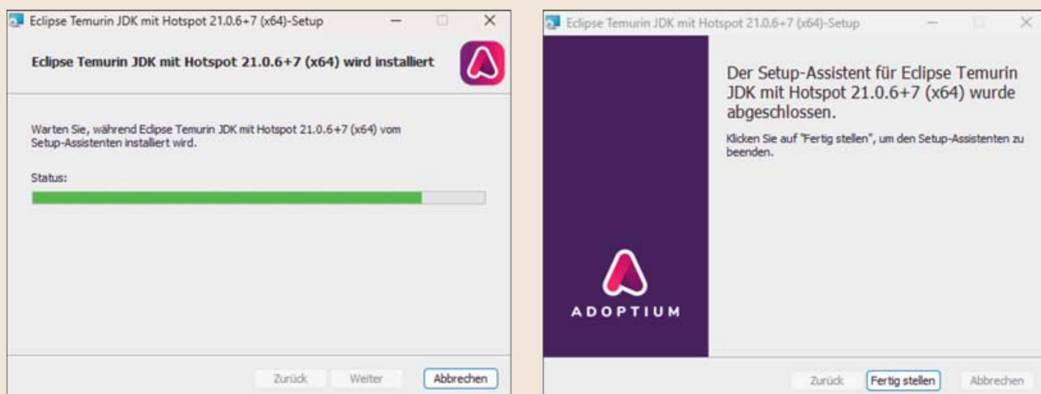

Abbildung 2.5 Nachdem die Installation durchlaufen wurde, ist die Einrichtung des JDK abgeschlossen.

Schon haben wir das JDK und damit auch JRE und JVM auf dem PC installiert. Doch bevor wir mit den eigentlichen Themen rund um die Plugin-Entwicklung beginnen, werden wir ein weiteres wichtiges Programm installieren, das uns das Leben während der Entwicklung deutlich vereinfacht.

2.1.2 IntelliJ: Wie eine Entwicklungsumgebung uns das Leben leichter macht

Du hast bereits gelesen, dass Quellcode streng vorgeschriebenen Regeln folgt und anfällig für kleine Schusselfehler sein kein. Nichts wäre ärgerlicher, als wenn dein Plugin durch kleine Unaufmerksamkeiten nicht so funktioniert, wie du es dir vorgestellt hast! Deshalb sollten wir uns darüber Gedanken machen, wie wir Quellcode möglichst so schreiben können, dass wir schnell sehen, wenn uns ein Fehler unterläuft. Gleichzeitig wollen wir Code nicht nur schnell und sicher analysieren, sondern auch schreiben können. Ein normaler Texteditor wäre hier ungeeignet: Immerhin können wir hier jedes Zeichen tippen, ohne dass der Editor weiß, ob es sich um korrekten Java-Code handelt. Fehler würden uns dann erst auffallen, wenn wir versuchen, den Quellcode zu kompilieren, und der Compiler unseres JDK meckert. Die Lösung für all die genannten Probleme liegt in der Verwendung einer *Integrated Development Environment* (kurz *IDE*), zu Deutsch einer *Entwicklungsumgebung*.

Eine IDE ist ein Programm, das dafür entwickelt wurde, uns beim Programmieren zu unterstützen. Auf dem Markt gibt es die verschiedensten IDEs, die meist für bestimmte Sprachen oder Anwendungsbereiche entwickelt wurden. Während sich die genaue Funktionsweise und auch der Umfang einzelner Entwicklungsumgebungen unterschei-

det, haben sie alle ähnliche Werkzeuge, die uns bei der Programmierung unterstützen sollen. Die wichtigsten Vorteile sind:

- Die IDE hebt deinen Quellcode in bestimmten Farben und mit anderen visuellen Elementen hervor. So kannst du den Code auch nachträglich besser lesen und verstehen. Außerdem werden Fehler sofort angezeigt, wenn sie auftreten. Dadurch siehst du direkt, wenn du dich mal vertippt hast und der Quellcode nicht mehr richtig vom Compiler gelesen werden kann. Doch nicht nur das: Du kannst die IDE zum Live-Test mit deinem Server verbinden und dich in die Funktionen deines Plugins »hacken«. Auf diese Weise kannst du schnell Fehlerquellen finden, wenn irgendetwas im Quellcode nicht stimmt.
- Die Entwicklungsumgebung enthält zahlreiche Werkzeuge zum schnellen Schreiben von bestimmten Codeabschnitten, die immer wieder auftreten. Nervige Arbeiten kannst du also dem Programm überlassen, während du dich auf den spannenden Programmierteil konzentrierst. Sogar während des Tippens denkt die IDE für dich mit und versucht dir vorzuschlagen, welcher Quellcode zu einer bestimmten Situation passen könnte. Auch diese Eigenschaft hilft dir, Quellcode sicherer und schneller zu schreiben.
- In deiner IDE ist ein Dateiexplorer eingebaut, mit dem du verschiedene Dateien verwalten kannst, die du für das Projekt benötigst. Du wirst am Beispiel deines ersten Plugins sehen, dass noch viele Dateien über den Java-Quellcode hinaus eine wichtige Rolle spielen. Im Programm behältst du aber die volle Übersicht und kannst schnell zwischen Dateien wechseln oder neue Dateien anlegen.
- Außerdem ist ein Java-Compiler direkt verbaut, sodass du den geschriebenen Quellcode schnell in Java-Bytecode und damit zu einem fertigen Plugin weiterverarbeiten kannst. Dieses kannst du dann auf deinen Server laden, um alles zu testen.

Neben diesen wichtigen Vorteilen gibt es viele weitere praktische Funktionen, die von der Entwicklungsumgebung angeboten werden. Damit du in den Genuss all dieser Funktionen kommst, wollen wir uns neben dem JDK eine IDE installieren, die uns beim Programmieren »über die Schultern schaut« und beispielsweise hilft, wenn wir neuen Quellcode schreiben oder Fehler in vorhandenem Code finden müssen. Dafür verwenden wir das Programm *IntelliJ IDEA* (kurz *IntelliJ*) der Firma JetBrains. IntelliJ ist eine der bekanntesten und beliebtesten Entwicklungsumgebungen für Java und ist in der Grundversion, die den Namen *IntelliJ Community Edition* trägt, vollkommen kostenlos. Daneben gibt es die kostenpflichtige Version *IntelliJ Ultimate*, die Zusatzfunktionen bietet, die wir im Rahmen des Buchs allerdings nicht benötigen.

Vereinheitlichung der Community und Ultimate Edition

JetBrains hat angekündigt, dass ab der Version 2025.3 die Community Edition und die Ultimate Edition zu einer Distribution vereinigt werden. Das bedeutet, dass du in der Zukunft nur noch ein Installationsprogramm auf der Seite findest, das für eine einheitliche Installation von IntelliJ genutzt wird.

Lade dir zunächst das Installationsprogramm von der offiziellen Website der IDE herunter: *https://www.jetbrains.com/idea/download*. Die Download-Ansicht ist in Abbildung 2.6 dargestellt. Achte dabei darauf, dass du die kostenfreie Community Edition, für die du ein Stück nach unten scrollen musst, auswählst.

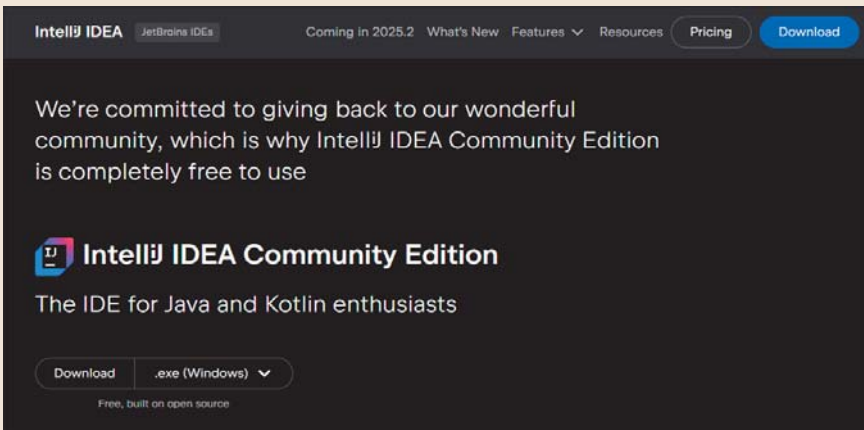

Abbildung 2.6 Auf der IntelliJ-Website gibt es Installationsprogramme der Community Edition für verschiedene Betriebssysteme, so auch für Windows.

Das heruntergeladene Installationsprogramm kannst du sofort starten, damit IntelliJ auf deinem Rechner installiert wird. Die Schritte dafür sind einfach: Nach einer Begrüßung wählst du den Installationsordner aus. Anschließend kannst du verschiedene optionale Häkchen setzen (beispielsweise, um eine Verknüpfung zum Programm auf dem Desktop zu erstellen), du musst hier nichts anklicken. Im nächsten Menü kannst du den Startmenüordner manuell setzen, auch diesen Schritt kannst du überspringen, damit die Installation startet. Nach einer kurzen Weile ist IntelliJ fertig aufgesetzt, wir lassen die IDE aber noch ein wenig verschnaufen, bis sie zum Einsatz kommt. Die einzelnen Installationsschritte sind zusammengefasst in Abbildung 2.7 zu sehen.

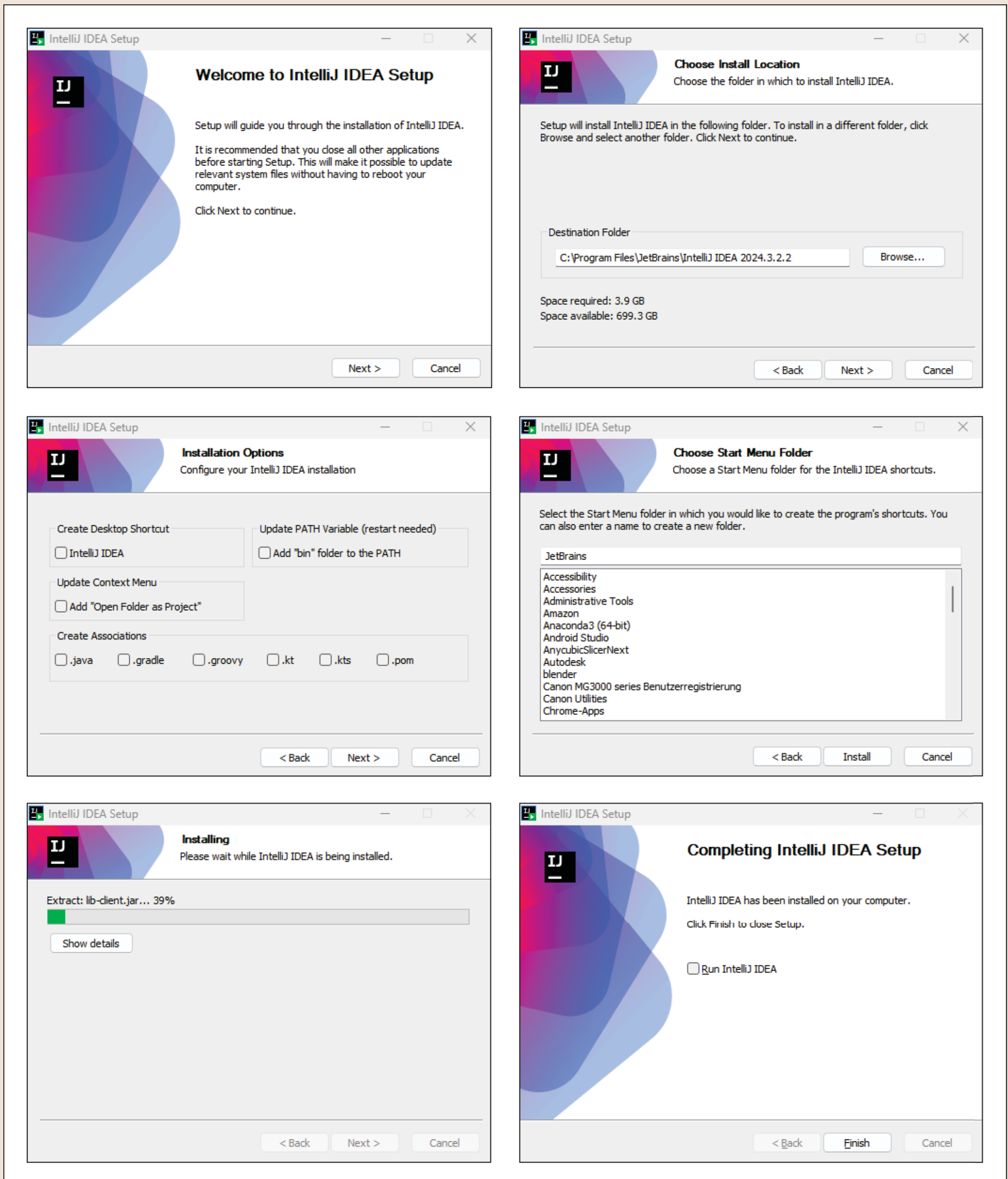

Abbildung 2.7 Während der Installation kannst du dich einfach durch die einzelnen Schritte klicken.

2.1.3 Basics zu Plugin-Frameworks und Einführung in Paper

Nun, da wir alle wichtigen Programme haben, fehlt nur noch der Star in unserer Show: Ein Server, der unsere frischgebackenen Plugins erhält, damit wir sie auf Herz und Nieren testen können. Mit der Wahl der Server-Software legen wir uns gleichzeitig fest, welche Art von Plugins (oder Mods) unterstützt wird. Ich möchte dir einen kurzen Abriss geben, wie sich verschiedene bekannte Anbieter für genau diese Software über die letzten Jahre entwickelt haben.

Der erste bekannte Vertreter, den seit 2010 viele Server verwendeten, hieß *Bukkit*. Da Minecraft selbst keine Unterstützung für Plugins anbot, wurde dieser Server-Ersatz von vielen engagierten Spielerinnen und Spielern gebastelt, damit Plugins auf dem Server lauffähig sind. Bukkit wurde 2014 aus rechtlichen Gründen eingestellt, das Tor für eine riesige Plugin-Community und weitere Server-Software war damit dennoch geöffnet. Die nächste große Entwicklung trägt den Namen *Spigot*, sie wurde 2012 ins Leben gerufen, baute direkt auf Bukkit auf und verbesserte viele Punkte in der Leistungsfähigkeit. Noch heute ist Spigot eine der beliebtesten Server-Umgebungen, die von vielen Servern weltweit genutzt wird.

Neben zahlreichen weiteren Entwicklungen, die in der Zwischenzeit entstanden, sticht eine besonders hervor: Sie trägt den Namen *PaperMC* (kurz *Paper*) und baute anfangs wiederum auf Spigot auf. Paper enthält viele weitere Optimierungen, die dafür sorgen, dass der Server noch flüssiger läuft und auch unter großen Lasten leistungsfähig bleibt. Da Paper ursprünglich von Spigot abstammt, funktionieren auch derzeit Spigot-Plugins noch zu einem Großteil auf Paper-Servern, weshalb wir Paper für unseren Test-Server verwenden wollen. In Zukunft können sich jedoch einige Dinge bei der Kompatibilität der verschiedenen Plugin-Arten ändern.

Doch nicht nur für Betreiber von Servern ist Paper wahnsinnig interessant, es gibt auch viele Optimierungen für uns Plugin-Entwickler und -Entwicklerinnen! Der wohl wichtigste Nutzen sind verbesserte Aufrufe im Quellcode, die im Detail erlauben, verschiedene Aspekte des Spiels anzusteuern. Wie genau sich das beim Programmieren gestaltet, wirst du schon bald sehen. Außerdem profitierst du beim Entwickeln von den verschiedenen Optimierungen, die Paper-Server bereitstellen: Selbst das Erschaffen riesiger Block-Schlösser durch Code oder das Teleportieren eines Spielers durch die ganze Welt wird so zum Kinderspiel, das der Server gut verarbeiten kann. Zuletzt gibt es viele Stellen, die uns bei Fehlern behilflich sein können: Neben detaillierten Fehler-Logs, die unser Server aufzeichnet, hat Paper eine gewaltige Dokumentation, die jedes kleinste Detail der Programmierschnittstelle erklärt. Nicht zu unterschätzen ist außerdem die große Entwicklungsgemeinschaft, die Paper verwendet: Du wirst im Handumdrehen viele andere Entwicklerinnen und Entwickler finden, die dir bei Problemen helfen können oder die sich mit dir über deine neuesten Ideen für Plugins austauschen.

In der Vergangenheit wurden Server meist über eine Java-Datei und ein kleines Script, das den Server automatisch startet, in Betrieb genommen. Wir machen es uns mit einer praktischen Erweiterung für das Plugin-Projekt aber deutlich einfacher und lassen den Server direkt über das Projekt in IntelliJ starten. So ist es nicht nur deutlich leichter, den Server in Betrieb zu nehmen, sondern wir können bei Änderungen das Plugin sofort auf dem Server aktualisieren und den Server anschließend neu starten. Das Aufsetzen dieses Servers nehmen wir somit bei der Einrichtung des Projektes vor, mit der wir nun beginnen wollen.

2.2 Das Plugin-Projekt erstellen

Nun, da wir alle notwendigen Programme haben und wissen, welchen Server wir später installieren, kann es endlich mit dem eigentlichen Inhalt des Buches losgehen: Wir programmieren unser erstes Plugin! Jedes Plugin, das du entwickelst, kommt mit einigen Dateien daher. Damit wir die Übersicht behalten, wird in IntelliJ eine solche Sammlung verschiedener Entwicklungsdateien als ein *Projekt* abgelegt. Unter der Motorhaube ist ein solches Projekt eigentlich nur ein Ordner, allerdings können wir in der IDE einen guten Überblick über die Projektdateien behalten und zusätzlich schnell zwischen verschiedenen Projekten hin und her wechseln.

Wir wollen ein solches Projekt für das Plugin, das wir im Laufe des Buches entwickeln, erstellen. Es ist also an der Zeit, dass du die Entwicklungsumgebung IntelliJ zum ersten Mal startest. Dabei gelangst du zunächst in ein Willkommens-Fenster (siehe Abbildung 2.8), über das du ein neues Projekt erstellen oder ein vorhandenes Projekt öffnen kannst.

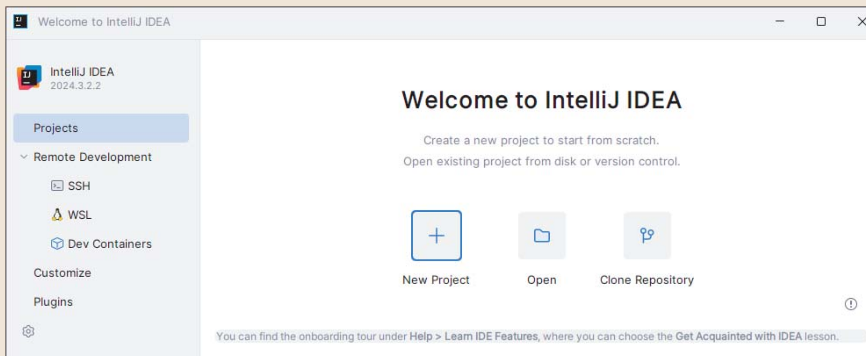

Abbildung 2.8 Beim ersten Start heißt IntelliJ dich herzlich willkommen.

Klicke im Hauptmenü auf den Knopf NEW PROJECT. Es öffnet sich ein neues Fenster, in dem du auswählen kannst, mit welcher Vorlage das neue Projekt generiert werden

sollte. Die Ansicht sollte zunächst so ähnlich aussehen wie in Abbildung 2.9, wenn du im linken Teil Java wählst.

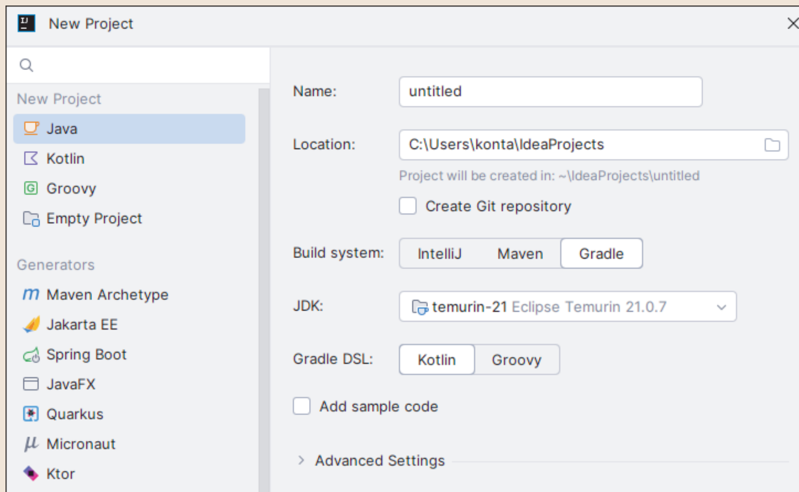

Abbildung 2.9 In dem Erstellungsfenster für ein neues Java-Projekt kannst du verschiedene Einstellungen für das neue Projekt vornehmen.

Das Projekt ohne eine erweiterte Vorlage erstellen

Es gibt eigene Plugins für IntelliJ (ja, nicht nur Minecraft hat Plugins!), mit denen Projektvorlagen für Paper-Plugins verfügbar sind. Das Problem bei vieler dieser Vorlagen ist allerdings, dass bestimmte Details nicht mehr aktuell sind. Wir bauen das Projekt deshalb »von Grund auf« auf. Das beugt nicht nur Fehlern vor, sondern du lernst gleichzeitig, aus welchen wichtigen Bestandteilen ein Paper-Plugin bestehen muss.

Für das neue Projekt wollen wir zunächst einige Daten konfigurieren. Sehen wir uns dafür die wichtigsten Einstellungen an, die wir für das Paper-Plugin einrichten wollen:

- Name: Der Name, unter dem das neue Projekt gespeichert wird. Das ist gleichzeitig der Name des Ordners, in dem alle Projektdateien liegen. Hier wähle ich »bookplugin«, immerhin wird es ein Plugin für dieses Buch.
- Location: Der Ordner, in dem das Projekt gespeichert wird. Über das Ordner-Symbol an der rechten Seite kannst du ihn direkt auswählen. Beachte, dass innerhalb dieses Ordners ein eigener Ordner für das Projekt erstellt wird. Ich habe einen eigenen Ordner für all meine Minecraft-Plugins erstellt, den ich auswähle.
- Build System: Das Build System ist ein nützlicher Teil des Projektes, mit dem wir verschiedene Prozesse beim Programmieren automatisieren. Die Hintergründe zum System behandele ich in Abschnitt 2.2.3. Wähle für den Moment die Option Gradle.

- JDK: Deine JDK-Installation. Wenn die heruntergeladene Temurin-Version 21 von Adoptium nicht automatisch in der Liste auftaucht, kannst du sie über die Option ADD JDK FROM DISK... über ihren Installationsordner hinzufügen.
- GRADLE DSL: Die Basis, auf der dein gewähltes Build System läuft. Wähle hier KOTLIN.
- ADD SAMPLE CODE: Wenn er gesetzt ist, entferne für diese Option den Haken, damit kein Beispielcode für dich generiert wird.

Unter ADVANCED SETTINGS interessieren uns lediglich die Einträge GROUPID und ARTIFACTID. Die GROUPID ist eine Art übergeordnete Zuordnung des Projektes. Standardmäßig wird dafür eine Art »umgekehrte Website« vergeben: Ich habe beispielsweise die Website *schlosser.dev*, weshalb ich »dev.schlosser« eintrage. Wenn du keine Website verwendest, kannst du einfach die Endung für dein Land (beispielsweise *.de* für Deutschland) und deinen Minecraft-Namen oder deinen Nachnamen verwenden. So könntest du als »Max Mustermann« zum Beispiel »de.mustermann« für die GROUPID eintragen. Die ARTIFACTID ist der Name für das konkrete Projekt, sodass ich hier den gleichen Namen wie das Projekt verwende, nämlich »book-plugin«.

Eine Übersicht über alle Änderungen, die ich bei der Projekterstellung vorgenommen habe, siehst du in Abbildung 2.10.

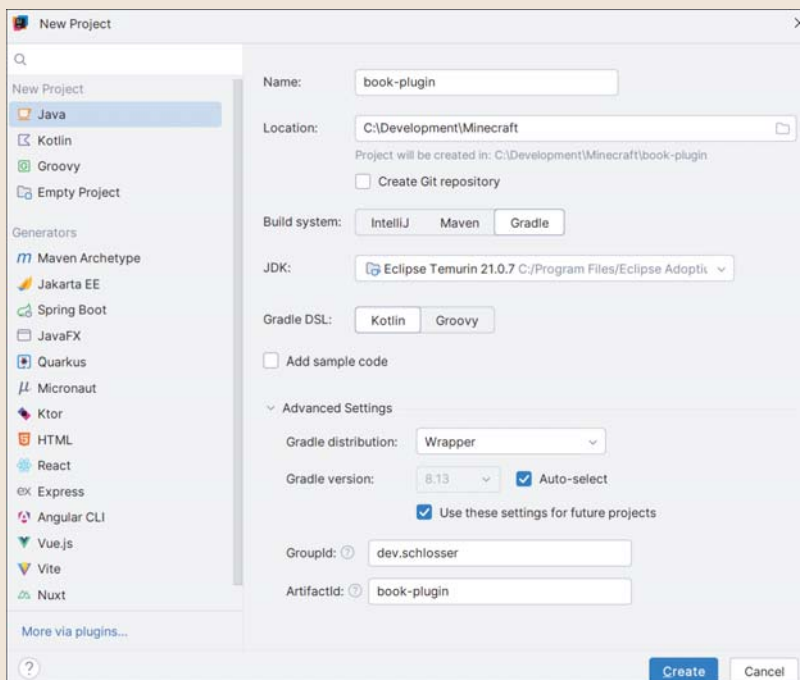

Abbildung 2.10 Für das neue Plugin zum Buch ändern wir einige Details.

Damit haben wir es geschafft: Das Plugin-Projekt ist aufgesetzt! Allerdings haben wir uns für ein leeres Projekt entschieden, das im Moment zu jedem erdenklichen Java-Programm werden könnte. Wir müssen uns im nächsten Schritt also darum kümmern, dass das Projekt als Paper-Plugin erkannt wird. Außerdem richten wir, wie bereits angesprochen, direkt im Projekt einen Test-Server ein, der auf deinem PC läuft und mit dem du das Plugin jederzeit testen kannst.

2.2.1 Die Hauptklasse für das Plugin erstellen

Wir beginnen, indem wir wichtigen Quellcode schreiben, mit dem das Java-Projekt als Plugin erkannt werden kann. Wie du sicher weißt, ist Ordnung das halbe Leben, und so ist auch unser IntelliJ-Projekt in ziemlich viele Ordner aufgeteilt. Für Code interessiert uns primär der Ordner *src*. Darin findest du einen Unterordner mit dem Namen *main*, in dem wiederum der Ordner *java* untergebracht ist. Die einzelnen Ordner kannst du in IntelliJ über den kleinen Pfeil an der linken Seite aus- oder einklappen (siehe Abbildung 2.11).

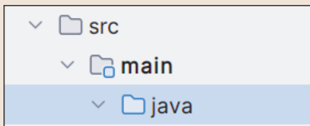

Abbildung 2.11 Alle Quellcodedateien werden im Unterordner »java« organisiert.

Doch hier hört die Ordnung noch nicht auf: Wir können innerhalb des Ordners Code in sogenannten *Packages* organisieren. So ein Package kannst du dir wie einen weiteren Ordner vorstellen, der verschiedene Quellcodedateien enthalten kann. Später werden wir mehrerer solcher Packages in unserem Projekt anlegen, sodass die Java-Dateien aus verschiedenen Packages miteinander interagieren. Wir wollen zunächst ein Package für den ersten Code, den wir schreiben, anlegen. Klicke dafür mit der rechten Maustaste auf den Ordner *java* und wähle, wie in Abbildung 2.12 gezeigt, die Option NEW • PACKAGE.

Jedes Package benötigt einen Namen, der so wie die GROUPID beim Erstellen des Projekts standardmäßig mit dem umgekehrten Websitenamen angegeben wird. Darum starte ich den Package-Namen mit `dev.schlosser`. Damit besser erkennbar wird, dass das Package zum Plugin-Projekt gehört, füge ich getrennt durch einen Punkt schließlich noch den Projektnamen an, nämlich `bookplugin`. Den vollen Namen, `dev.schlosser.bookplugin`, trage ich in dem Dialog ein (siehe Abbildung 2.13).

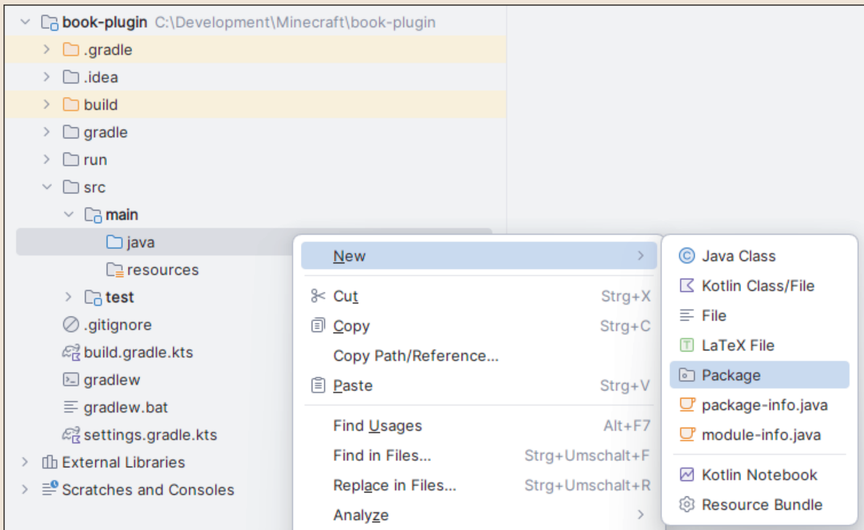

Abbildung 2.12 Mit der gezeigten Option legst du dein erstes Package an, in dem du deinen Code organisieren kannst.

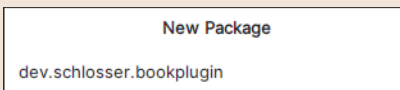

Abbildung 2.13 Für den Package-Namen wird eine durch Punkte getrennte Schreibweise verwendet.

Die Erstellung bestätigst du mit dem Drücken der Taste ↵. Sofort danach ist das neue Package im Ordner *java* deines Projektes sichtbar.

Der Package-Name in seinen Einzelteilen

Die Punkte haben nicht irgendeine Bedeutung: Wenn du den *java*-Ordner auf deinem PC öffnest, kannst du sehen, dass jeder Punkt zu einem eigenen Unterordner wird. Es gibt also nicht einen Ordner mit dem Namen *dev.schlosser.bookplugin*, sondern den Oberordner *dev*, darin einen weiteren Ordner *schlosser* und darin wiederum den letzten Ordner *bookplugin*. Die Ansicht dieses Ordners unter Windows siehst du in Abbildung 2.14, wobei oben der letzte Teil des Dateipfads angegeben ist.

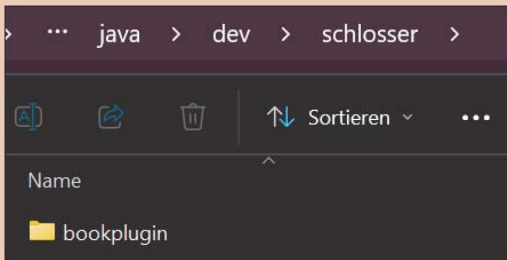

Abbildung 2.14 Jeder Punkt im Package-Namen sorgt im Hintergrund dafür, dass ein neuer Unterordner angelegt wird.

Als Nächstes benötigen wir eine zentrale *Klasse*, die das Herzstück unseres Plugins darstellt. Eine Klasse ist vereinfacht gesagt ein Behälter für Quellcode. Dieser Code kann dann zusammengefasst aus dieser Klasse verwendet werden. Im Laufe des Buches wirst du weitere Details dazu erfahren, welche Bedeutung Klassen haben. Für den Moment reicht es, darüber hinaus zu wissen, dass Klassen einen Namen tragen.

Deine erste Klasse soll eine besondere Bedeutung für das Projekt haben, denn sie soll als *Hauptklasse* für das Plugin dienen. Das heißt, dass die Klasse der Ausgangspunkt für verschiedene wichtige Funktionen deines Plugins sein wird. Die Hauptklasse des Plugins legen wir im gerade erstellten Package an. Mache dafür einen Rechtsklick auf den Package-Namen und wähle die Option NEW • JAVA CLASS (siehe Abbildung 2.15).

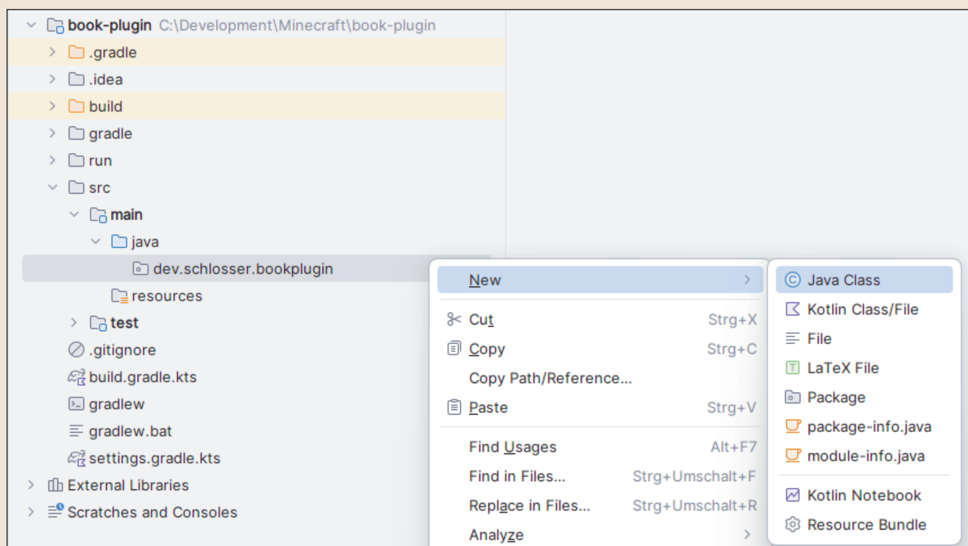

Abbildung 2.15 Innerhalb des Packages wird eine neue Klasse erstellt.

Nach dem Wählen der Option JAVA CLASS erscheint wie beim Package in Abbildung 2.13 ein Dialog, in dem der Name für die Klasse vergeben wird. Wichtig ist, dass der Klassenname keine Leerzeichen enthalten darf. Ich möchte dennoch den Namen des Projektes, also »Book Plugin«, darstellen. Dafür schreibe ich die beiden Wörter einfach zusammen, sodass ich `BookPlugin` wähle. Nach dem Bestätigen des Klassennamens mit ↵ wird die neue Klasse innerhalb des Packages angezeigt. Wie du in Abbildung 2.16 siehst, füllt sich gleichzeitig das erste Mal das große Fenster in der Mitte deiner IDE mit Leben: Der Code für deine erste Klasse wird angezeigt!

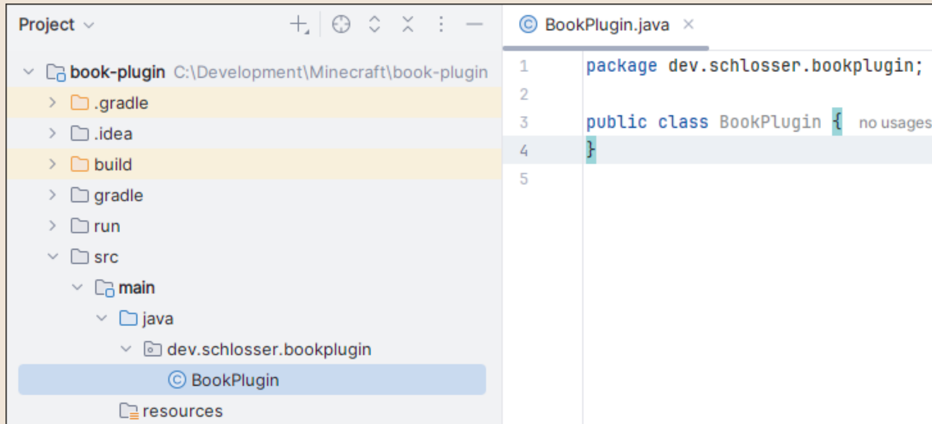

Abbildung 2.16 Der erste Code! Die neu generierte Klasse wird automatisch von IntelliJ mit Leben befüllt.

An dieser Stelle verstehst du wahrscheinlich zum Großteil Bahnhof, immerhin hast du diesen Code nicht selbst geschrieben. Deshalb wollen wir uns den automatisch generierten Inhalt der Klasse zunächst näher anschauen.

In English, please

Zunächst ein wichtiger Hinweis, der sich durch das gesamte restliche Buch ziehen wird: Englisch hat sich nicht nur außerhalb der Programmierwelt als Weltsprache etabliert: Wenn es um Quellcode und auch um zugehörige Programme wie IntelliJ geht, sind viele Befehle und Menüpunkte »von Haus aus« in englischer Sprache verfasst. Daran solltest du dich auch beim Schreiben von eigenem Quellcode gewöhnen, da das den großen Vorteil hat, dass andere Menschen aus aller Welt deinen Code sofort lesen und besser verstehen können. Zwar könntest du bestimmte Quellcodeabschnitte auch auf Deutsch verfassen, allerdings würden die deutschen Wörter dann wirr zwischen all den englischen Bezeichnungen, die Java und Paper mit sich bringt, stehen. Das wäre ein ziemliches Chaos!

> Aus diesem Grund werde ich alle Beispiele, die sich um Quellcode drehen, ebenfalls in Englisch verfassen. Viele Begriffe werden dir bekannt vorkommen, wenn du die Sprache beispielsweise in der Schule schon einmal gelernt hast oder dich mit Minecraft auseinandergesetzt hast. Du brauchst dir keine Sorgen zu machen, dass es an diesem Punkt scheitert. Neue wichtige englische Befehle und Bezeichnungen werde ich, wenn ich sie zum ersten Mal verwende, erläutern. Wenn dir Wörter, die ich nicht erkläre, unbekannt vorkommen, dann kannst du jederzeit in ein Wörterbuch schauen.

Starten wir mit der ersten Zeile, die ganz oben in der Datei steht:

```
package dev.schlosser.bookplugin;
```

Diese Zeile sieht bei dir wahrscheinlich anders aus, denn hier taucht der Name des Packages auf, das du vor deiner Klasse erstellt hast. Dafür wird die Zeile durch das Wort `package` eingeleitet. Abgeschlossen wird die Zeile durch ein Sonderzeichen, das du davor wahrscheinlich noch nicht so häufig verwendet hast, das *Semikolon* (`;`). Während diese lustige Kombination aus Punkt und Komma beim Schreiben von Texten seltener zum Einsatz kommt, kommt ihr beim Programmieren mit Java eine besondere Bedeutung zu. Vereinfacht kannst du dir zunächst merken, dass jede Zeile, die einen Befehl darstellt, mit einem solchen Semikolon abgeschlossen wird. Das passt in unserem Fall sehr gut: Wir befehlen mit unserem Code also, dass die Datei dem Package *dev.schlosser.bookplugin* zugeordnet wird.

Spannender wird es mit dem Teil der Klasse, der ab der dritten Zeile beginnt:

```
public class BookPlugin
```

Die Bedeutung der Wörter musst du noch nicht im Detail verstehen, allerdings ist das grobe Verständnis wichtig:

- **`class BookPlugin`**: Dieser Teil der Zeile ist dafür zuständig, dass die Datei als Klasse (also als Behälter für Code) verstanden wird. Während das Wort `class` den Code als Klasse ausweist, ist `BookPlugin` der *Name* der zugehörigen Klasse. Wie du siehst, hat IntelliJ diesen Klassennamen automatisch nach der Eingabe im Dialogfenster eingetragen, sodass Datei und Klasse den gleichen Namen haben.
- **`public`**: Dieses Wort, das mit »öffentlich« übersetzt werden kann, beschreibt, wo im Quellcode auf diese Klasse zugegriffen werden kann. Im Falle von `public` dürfen alle anderen Klassen auf die `BookPlugin`-Klasse zugreifen.

Die Einleitung dieser Klasse kommt mit einer weiteren Besonderheit: Die Zeile wird nicht mit einem Semikolon abgeschlossen, sondern mit einer offenen geschweiften

Klammer ({). Geschweifte Klammern haben, so wie das Semikolon, eine besondere Bedeutung: Sie grenzen bestimmte *Bereiche* im Quellcode ab. Darum gehört zu jeder offenen geschweiften Klammer immer eine geschlossene (}). So ist es auch im Falle unserer Klasse: Die betreffende Klammer befindet sich in der letzten Zeile.

Damit die Klasse als Hauptklasse für ein Paper-Plugin erkannt wird, müssen wir noch eigenen Code schreiben. Bevor es in Abschnitt 2.2.4 so weit ist, müssen wir allerdings einige weitere Vorbereitungen treffen.

2.2.2 Die »paper-plugin.yml«-Datei anlegen

Der nächste wichtige Schritt ist das Anlegen einer *paper-plugin.yml*-Datei. Diese Datei enthält verschiedene Details zu deinem Plugin, die dein Server zum Laden des Plugins oder für die Ausgabe von Informationen benötigt.

»plugin.yml« als ältere Version der »paper-plugin.yml«

Unter Spigot wurde für diese Datei der Name *plugin.yml* verwendet. Auch dieser Dateiname wird unter Paper unterstützt, auch der Aufbau beider Dateien ist größtenteils gleich. Dennoch signalisieren wir durch das Verwenden von *paper-plugin.yml*, dass wir tatsächlich ein Plugin für Paper entwickeln, und können in der Datei Daten hinterlegen, die bei Spigot nicht unterstützt werden.

Lege zunächst die *paper-plugin.yml*-Datei im Ordner *resources*, der sich neben deinem Codeordner *java* unter *main* befindet, an. Mache dafür einen Rechtsklick auf *resources* und wähle diesmal die Option New • File (siehe Abbildung 2.17). Tippe als Dateinamen dann »paper-plugin.yml« ein (exakt so geschrieben!) und bestätige die Erstellung über die Taste [↵].

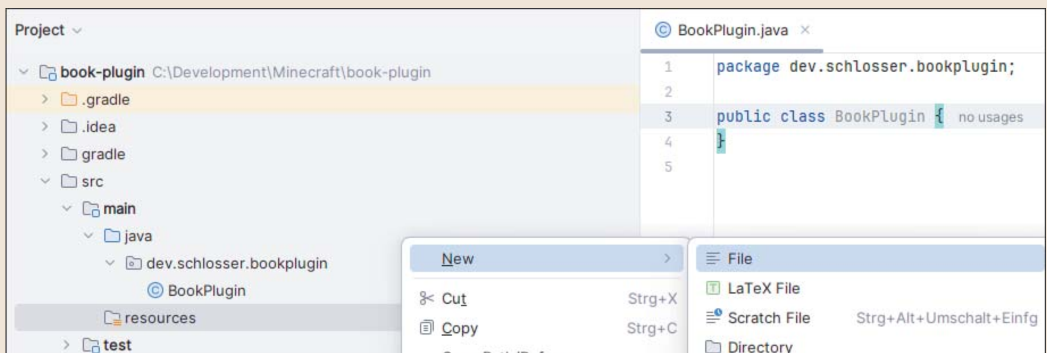

Abbildung 2.17 Die »paper-plugin.yml«-Datei muss im Ordner »resources« erstellt werden.

Nachdem du die Datei erstellt hast, öffnet sie sich in einem neuen Tab im Codefenster. Natürlich ist die Datei im Moment noch leer. Wir müssen also einige Einstellungen festlegen. Insgesamt sind gerade einmal drei solcher Einstellungen notwendig, damit die *paper-plugin.yml*-Datei für ein Paper-Plugin gültig ist:

- **`name`**: Der Name, unter dem der Server das Plugin erkennt. Es ist wichtig, dass dieser Name keine Leer- und Sonderzeichen enthält.
- **`version`**: Die Versionsnummer, unter der der Server das Plugin ausliest. Du kannst die Version später anpassen, wenn du Updates für dein Plugin veröffentlichst. Vielleicht hast du schon einmal die Schreibweise von Versionen gesehen: Dabei bedeuten Zahlen, die weiter links stehen, größere Versionssprünge. So könnte auf Version 1.0 beispielsweise Version 1.1 folgen, dann 1.2 und so weiter. Bei größeren Sprüngen könnte dann Version 2.0 eingeleitet werden. Genau nach diesem Schema vergibt auch Minecraft seine Versionsnamen.
- **`main`**: Hiermit wird der Pfad zur Hauptklasse des Plugins angegeben. Hier wollen wir die gerade erstellte `BookPlugin`-Klasse hinterlegen. Der Pfad setzt sich aus der Bezeichnung des Packages (bei mir `dev.schlosser.bookplugin`) und, nach einer weiteren Trennung durch einen Punkt, dem Namen der Klasse (bei mir `BookPlugin`) zusammen. Daran, dass die Klasse in der *paper-plugin.yml*-Datei hinterlegt ist, wird deutlich, wie wichtig die Hauptklasse in den nächsten Schritten wird.

Eine Eigenschaft wird angelegt, indem du den Namen schreibst und nach einem Doppelpunkt den jeweiligen Wert einsetzt. Auf diese Weise sieht meine *paper-plugin.yml*-Datei zunächst so aus:

```
name: BookPlugin
version: 1.0.0
main: dev.schlosser.bookplugin.BookPlugin
```

Achte darauf, dass du unter `main` deinen Package-Namen einträgst und du den Klassennamen anpasst, falls du nicht ebenfalls `BookPlugin` verwendet hast.

Neben diesen unverzichtbaren Eigenschaften gibt es zahlreiche weitere Eigenschaften, die in der Datei hinterlegt werden können (aber nicht müssen). Die wichtigsten dieser Eigenschaften sind:

- **`api-version`**: Die Paper-Server-Version, auf die das Plugin ausgerichtet ist. Diese Einstellung ist besonders wichtig, damit veraltete Funktionen älterer Versionen möglichst fehlerfrei funktionieren. Für diese Eigenschaft kannst du die passende Minecraft-Versionsnummer angeben, wobei die niedrigste mögliche Version `1.13.` ist. Die Angabe ist ab der Version 1.20.5 mit nur zwei Abstufungen (also `1.21` statt `1.21.7`) möglich, bei einer älteren Version muss die volle Version angegeben werden.

- `description`: Ein kurzer Beschreibungstext für das Plugin.
- `author`: Hier kannst du dich verewigen, denn unter dieser Eigenschaft kannst du den Namen des Plugin-Erstellers aufführen.
- `authors`: Solltest du nicht allein am Plugin gearbeitet haben, können auch mehr Namen hinterlegt werden. Die Namen müssen in eckige Klammern geschrieben und durch Kommas voneinander abgetrennt werden, beispielsweise so:

  `authors: [Name1, Name2, Name3]`
- `website`: Ein Website-Link für das Plugin. Du kannst beispielsweise eine Seite verlinken, unter der du den Quellcode hochgeladen hast, oder, wenn du das Plugin groß aufziehst, eine von dir bereitgestellte Internetseite, auf der du alle Funktionen des Plugins genau erläuterst.
- `prefix`: Der Name, der vor Nachrichten deines Plugins in die Konsole gestellt wird. Standardmäßig wird hier der Plugin-Titel von der Eigenschaft `name` verwendet, doch wenn du einen anderen vorangestellten Namen wünschst, kannst du ihn hier setzen.
- `depend`: Über diese Eigenschaft kannst du eine Liste von Plugins angeben, die vor deinem Plugin geladen werden soll. Das ist besonders praktisch, wenn dein Plugin auf anderen Plugins aufbaut und diese Plugins somit vor deinem geladen werden müssen. Die einzelnen benötigten Plugins werden über ihren Namen (also den Namen, den diese Plugins in ihrer *paper-plugin.yml*-Datei festlegen) angegeben. Die Liste wird wie bei `authors` über kommagetrennte Werte in eckigen Klammern angelegt, beispielsweise so:

  `depend: [plugin1, plugin2, plugin3]`
- `loadbefore`: Die umgekehrte Eigenschaft zu `depend`, die angibt, dass dein Plugin *vor* anderen Plugins geladen werden soll. Das kann praktisch sein, wenn dein Plugin diese anderen Plugins manipulieren soll.

Für eine detaillierte Liste aller verfügbaren Dinge, die du über die *paper-plugin.yml*-Datei einstellen kannst, wirf einen Blick in die Paper-Dokumentation: *https://docs.papermc.io/paper/dev/plugin-yml*.

Neben den zuvor gesetzten Einstellungen füge ich `api-version`, `author` und `description` zur *paper-plugin.yml* hinzu (siehe Listing 2.1). Unter `author` solltest du dabei natürlich deinen eigenen Namen einfügen.

```
name: BookPlugin
version: 1.0.0
main: dev.schlosser.bookplugin.BookPlugin
api-version: 1.21
author: Schlosser
description: Mein Buch-Plugin
```

Listing 2.1 Die fertige »paper-plugin.yml«-Datei mit erweiterten Einstellungen

Mit der Fertigstellung der *paper-plugin.yml*-Datei ist der nächste Schritt zum ersten Plugin erledigt!

> **Falls du Probleme mit der Erstellung der »paper-plugin.yml«-Datei hast**
>
> Du kannst dir die hier gezeigte Version der *paper-plugin.yml*-Datei von der Website zum Buch (*https://www.rheinwerk-verlag.de/6140*) herunterladen. Verschiebe die Datei dann einfach in den Ordner *resources* und passe die Eigenschaft `main` an, sodass dein Package- und Klassenname hinterlegt ist.

2.2.3 Paper und den Server mithilfe des Build Tools einrichten

Im nächsten Schritt lernst du einen sehr wichtigen und praktischen Unterstützer kennen, der ebenfalls in der Projektvorlage enthalten war. Dabei handelt es sich um ein sogenanntes *Build Tool* (deutsch »Bauwerkzeug«). Der Name ist hier Programm, denn das Build Tool unterstützt dich bei allen Prozessen rund um das Bauen einer fertigen Java-Datei aus deinem Quellcode, also dem Kompilieren. Doch nicht nur das: Mit dem Build Tool können wir Programmierbibliotheken für die Entwicklung direkt im Projekt herunterladen und davon ausgehend Prozesse aus dem Projekt starten. Wie du gleich sehen wirst, ermöglicht uns diese Eigenschaft, alle Funktionen der Paper-Entwicklung in unserem Projekt zu verwenden und gleichzeitig einen Minecraft-Server zum Testen unseres Plugins direkt aus IntelliJ zu starten.

In unserem Projekt haben wir uns in Abschnitt 2.2 für das Build Tool *Gradle* entschieden. Gradle ist heute eines der bekanntesten Build Tools für Java und wird besonders für seine sehr gut anpassbare Struktur verwendet. Ein anderes bekanntes Build Tool heißt *Maven*, dieses Werkzeug folgt bei der Einrichtung allerdings teils strikteren Strukturen. In diesem Buch werden wir ausschließlich Gradle verwenden, um unsere Plugins zu bauen. Auch wenn wir kaum selbst Dinge in Gradle anpassen müssen, möchte ich dir einen kurzen Überblick über die wichtigsten Dinge der neuen Gradle-Einrichtung geben. Direkt im Anschluss nutzen wir Gradle, um unseren aktuellen Quellcode zu einem funktionierenden Plugin zusammenzubauen.

Mit dem Erstellen des Projektes wurden verschiedene Gradle-Dateien angelegt. Die meisten dieser Dateien interessieren uns zunächst nicht, da wir den Blick auf die für uns wichtigste Datei werfen: *build.gradle.kts*. Du kannst diese Datei mit einem Klick auf BUILD.GRADLE.KTS aus der Projektübersicht an der linken Seite deiner IDE öffnen. Zunächst: Bekomm keinen Schreck! In der Datei steht eine Menge an neuem Text, der dich vielleicht zunächst an Code erinnert. Es handelt sich dabei allerdings nicht um Java-

Code, sondern um eine andere Sprache, die von Gradle verwendet wird. Diese Sprache musst du nicht kennen und verstehen, um Gradle einzurichten, da wir viele Dinge direkt aus dieser Datei herauslesen können. Vielleicht kommen dir ein paar Sonderzeichen sogar von deiner Hauptklasse bekannt vor. Die generierte Datei sieht zunächst so aus:

```
plugins {
    id("java")
}

group = "dev.schlosser"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

dependencies {
    testImplementation(platform("org.junit:junit-bom:5.10.0"))
    testImplementation("org.junit.jupiter:junit-jupiter")
}

tasks.test {
    useJUnitPlatform()
}
```

Listing 2.2 Die von IntelliJ generierte Version der »build.gradle.kts«-Datei

Werfen wir zunächst einen Blick auf ein paar der wichtigsten Abschnitte, die für das Plugin-Projekt relevant sind:

- **`plugins`**: Wie beim Bereich unserer Hauptklasse kommt hier ein Block geschweifter Klammern zum Einsatz, der verschiedene Dinge zusammenfasst, die zur »Obergruppe« `plugins` gehören. Mit der darin enthaltenen Zeile `id("java")` weiß Gradle, dass es sich um ein Java-Projekt handelt. Damit kann es beim Kompilieren zuordnen, dass ein Java-Projekt zu einer fertigen Datei gebaut werden soll.
- **`group = "dev.schlosser"`**: Dies ist der Paketname des Projektes, den wir beim Erstellen in IntelliJ eingegeben haben. Er orientiert sich, wie das Java-Package, an der Schreibweise der »umgekehrten Domain«. Mit dem Gruppennamen kann dein Plugin später besser identifiziert werden, wenn du aktiv mit Build Tools arbeitest.

- **version = "1.0-SNAPSHOT"**: Der Bezeichner `1.0-SNAPSHOT` spiegelt den Versionsnamen deines Plugins wider. Der Anhang `-SNAPSHOT` bedeutet lediglich, dass es sich um eine Entwicklungsversion für das Plugin handelt.
- Der Bereich **repositories**: Hinter `repositories` wird ebenfalls ein Block geschweifter Klammern angebracht, der verschiedene Quellen aufführt, von denen Gradle Bibliotheken laden kann. Dabei ist standardmäßig der Unterpunkt `mavenCentral()` enthalten. Das ist das größte öffentliche Verzeichnis für das Veröffentlichen von Bibliotheken, das von Maven bereitgestellt wird, aber auch von Gradle genutzt werden kann.
- **dependencies:** In diesem Block wird konkret angegeben, welche Bibliotheken aus den `repositories` für das Projekt verwendet werden. In unserem Fall wurden zwei Bibliotheken eingebunden, die wir allerdings nicht für die Entwicklung unseres Plugins benutzen werden.

Den Bereich `tasks.test` lassen wir zunächst außen vor. Diese Standard-Gradle-Datei ist für uns nicht ausreichend: Wir wollen immerhin, dass die Paper-Bibliotheken direkt im Projekt verfügbar sind und einen Test-Server zur Verfügung stellen. Dafür benötigt es eine Anpassung der *build.gradle.kts*, die du zunächst aus Listing 2.3 direkt in dein Projekt übernehmen kannst. Neu hinzugefügte Zeilen habe ich fett formatiert.

```
plugins {
    id("java")
    id("xyz.jpenilla.run-paper") version "2.3.1"
}

group = "dev.schlosser"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
    maven("https://repo.papermc.io/repository/maven-public/")
}

dependencies {
    compileOnly("io.papermc.paper:paper-api:1.21.7-R0.1-SNAPSHOT")
}

java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(21))
```

```
    }
}

tasks {
    compileJava {
        options.encoding = "UTF-8"
    }

    processResources {
        from(sourceSets.main.get().resources.srcDirs) {
            filesMatching("paper-plugin.yml") {
                expand("version" to project.version)
            }
            duplicatesStrategy = DuplicatesStrategy.INCLUDE
        }
    }

    runServer {
        minecraftVersion("1.21.7")
    }
}
```

Listing 2.3 Die erweiterte »build.gradle.kts« mit Download der Paper-Bibliotheken und zur Bereitstellung eines Test-Servers

> **Die »build.gradle.kts«-Vorlage**
>
> Du musst diesen Code nicht abtippen: Die angepasste *build.gradle.kts*-Datei aus Listing 2.3 steht über die Website zum Buch (*https://www.rheinwerk-verlag.de/6140*) direkt zum Download zur Verfügung. Denk allerdings auch hier daran, die Eigenschaft `group` für dein Projekt anzupassen.

In diesem Moment siehst du wahrscheinlich Fehler in der Datei, was du beispielsweise daran erkennst, dass der Name des Tabs in IntelliJ rot unterstrichen ist. Das liegt daran, dass wir die verschiedenen Bibliotheken, die nun über Gradle eingebunden werden sollen, zunächst im Projekt herunterladen müssen. Glücklicherweise gibt es in IntelliJ ein Fenster, das uns bei dieser Aufgabe hilft. Du erreichst es über das Gradle-Elefantensymbol an der rechten Seite des Bildschirms. Klicke nach dem Öffnen auf das Aktualisieren-Symbol, wie in Abbildung 2.18 gezeigt. Der Synchronisierungsprozess startet dann und kann eine kurze Zeit in Anspruch nehmen.

Kapitel 9

Ausrüstung leicht gemacht: Items und Inventare

Schmieden mit Stil

In diesem Kapitel widmen wir uns einem besonders spannenden Thema, das in der Entwicklung von Plugins für verschiedenste Bereiche Anwendung findet: Items und Inventare. Ein *Item* ist jeder Block, jedes Werkzeug oder jedes Rüstungsteil, das du während des Spielens abbauen, sammeln oder craften kannst. All deine Items werden in deinem *Inventar* abgelegt oder zum Beispiel als Rüstung getragen, wodurch du beeinflusst wirst. Es gibt darüber hinaus andere Formen von Inventaren, beispielsweise Truhen, Öfen oder streng genommen sogar Werkbänke.

Mithilfe deines Plugins kannst du Items künstlich generieren und in den verschiedensten Situationen einem bestimmten Inventar hinzufügen, sie einem Spieler anlegen oder sogar auf natürliche Art und Weise in der Welt fallen lassen. Dabei kannst du nicht nur spezielle Effekte wie einen Namen oder Verzauberungen anwenden, sondern ganz neue Funktionen programmieren. Egal, ob ein Zauberstab, der Feuerbälle schießt, ein Block, der beim Platzieren eine gigantische Explosion erzeugt, oder eine Schneekanone, die Schneebälle verschießt, die die Landschaft in ein Winter-Wunderland verwandeln: Du kannst deiner Kreativität freien Lauf lassen, damit deine Items das Spielen auf deinem Server zu einer einzigartigen Erfahrung machen.

Doch auch Inventare bieten im Zusammenhang mit der Plugin-Entwicklung spannende Möglichkeiten. Du kannst »virtuelle« Inventare vor den Augen eines Spielers öffnen und darin beliebige Items anzeigen, die man sich einfach ins eigene Inventar ziehen kann. Das ist beispielsweise für einen Überlebenskampf, bei dem in Truhen zufällige Items erscheinen sollen, eine praktische Technik. Doch nicht nur das: Durch Inventare kannst du Menüs simulieren. So kannst du beispielsweise verschiedene Items für besondere Orte auf deinem Server hinterlegen, und beim Anklicken eines bestimmten Symbols wird der Spieler sofort zur passenden Position teleportiert.

Du lernst im Folgenden die wichtigsten Ansätze, mit denen du Ideen wie die genannten Beispiele umsetzen kannst.

9.1 Items erstellen und nutzen

Wir starten, indem du deine ersten Items erstellst und sie auf bestimmten Wegen verteilst, beispielsweise durch das Platzieren in das Inventar des Spielers, das Fallenlassen in der Welt oder das Anlegen als Rüstungsteil. Anschließend tasten wir uns in Abschnitt 9.1.1 und Abschnitt 9.1.3 an erweiterte Möglichkeiten heran, mit denen du Items im vollen Umfang »bearbeiten« und für eigene Ideen nutzen kannst.

Starten wir mit den Grundlagen, indem du einen neuen Befehl mit dem Namen `/itemtest` erstellst. Baue dafür in einer neuen Klasse `ItemTestCommand` das Grundgerüst für einen neuen Befehl auf und sorge dafür, dass nur Spieler den Befehl ausführen können (immerhin wollen wir, dass der Absender als Bezug für das Hinzufügen und Fallenlassen der neuen Items verwendet wird). Jedes Item, das du im Rahmen deines Plugins verwenden möchtest, wird in der Programmierung durch den Datentyp *ItemStack* dargestellt. Ein neues Item kannst du am einfachsten über die Methode `ItemStack.of` generieren, wobei du ein Material übergeben musst, das den Typ des neuen Items festlegt. Im folgenden Beispiel erstelle ich einen `ItemStack`, der ein Steinschwert (`Material.STONE_SWORD`) enthält, und speichere ihn in einer Variable ab:

```
ItemStack sword = ItemStack.of(Material.STONE_SWORD);
```

Nun muss das Item nur noch in das Inventar des Absenders. Die einfachste Möglichkeit ist, das Item so hinzuzufügen, als wäre es gerade vom Boden aufgesammelt worden. Dabei wird der erste freie Slot, der im Inventar gefunden wird, verwendet. Dafür musst du das Inventar des Spielers zunächst über die Methode `getInventory` abrufen und den `ItemStack`, der hinzugefügt werden soll, als Parameter an die Methode `addItem` übergeben. Der volle Aufruf für den Beispielbefehl sieht so aus:

```
player.getInventory().addItem(sword);
```

Items an bestimmte Positionen setzen

Im Rahmen von Inventaren lernst du in Abschnitt 9.2, wie du Items an ganz bestimmte Positionen platzieren kannst. So ist es ebenfalls möglich, Items im Inventar eines Spielers zu überschreiben.

Damit ist das erste Beispiel geschafft: Du hast ein Item erstellt und fügst es dem Inventar des Absenders automatisch an einer passenden Stelle hinzu. Bevor wir den Befehl testen, bauen wir zwei weitere Tests für Items hinzu. Wir wollen nämlich außerdem ein paar Goldbarren vor dem Absender in der Welt fallen lassen. Für den `ItemStack` ist dabei keine große Änderung notwendig, du kannst weiterhin den Aufruf `ItemStack.of` verwenden, diesmal mit dem passenden `Material.GOLD_INGOT`. Der wichtige Unterschied: Wir wollen diesmal nicht ein Item generieren, sondern einen Stapel aus 10 Barren. Dafür kannst du der Methode einfach einen zweiten `int`-Parameter übergeben, der für die Anzahl der Items steht. Der Aufruf für diesen Test sieht so aus:

```
ItemStack gold = ItemStack.of(Material.GOLD_INGOT, 10);
```

Den neuen `ItemStack` können wir nun direkt in der Welt, in der sich der Spieler befindet, fallenlassen. Wir wollen das Item direkt vor dem Spieler fallen lassen, sodass wir die Blickrichtung einbeziehen. Dafür fügen wir der Position des Spielers wie bereits häufiger verwendet über `add` einen Abstand hinzu. Die Blickrichtung wird über die Methode `getDirection` der `Location` abgerufen. An den Abruf der Richtung ketten wir drei weitere Methodenaufrufe an:

- **`setY(0)`** sorgt dafür, dass das Blicken nach oben oder unten keinen Einfluss auf die Richtung hat, sondern lediglich die Ausrichtung nach vorn und zur Seite.
- **`normalize()`** führt eine sogenannte *Normalisierung* aus, was dafür sorgt, dass die Richtung nach ihrer Veränderung auf eine standardisierte Länge von einem Block gebracht wird.
- **`multiply(2)`** verdoppelt die Richtung. Da die Richtung zuvor auf die Länge von einem Block normalisiert wurde, steht der Ausdruck dafür, dass der finale Abstand zwei Blöcke beträgt.

Die fertige Berechnung der Drop-Position, die sich zwei Blöcke vor dem Spieler befinden soll, sieht so aus:

```
Location loc = player.getLocation().add(
  player.getLocation().getDirection().setY(0).normalize().multiply(2));
```

Das Fallenlassen an dieser Position erfolgt über die Methode `dropItemNaturally` der Welt, in der sich der Spieler befindet (`getWorld`). Der Methode werden die zuvor berechnete Position sowie das Item als Parameter übergeben:

```
player.getWorld().dropItemNaturally(loc, gold);
```

Als letztes Beispiel für diesen Befehl sehen wir uns an, wie wir bestimmte Items als Rüstung anlegen können. Dafür stellt das Inventar des Spielers verschiedene Methoden zur Verfügung, die allesamt einen `ItemStack` als Parameter übergeben bekommen:

- **`setHelmet`** für das Anlegen eines Helms, wobei auch andere Blöcke als Helm getragen werden können
- **`setChestplate`** für das Anlegen einer Brustplatte
- **`setLeggings`** für das Anlegen von Beinschienen
- **`setBoots`** für das Anlegen von Stiefeln

Testen wir als Beispiel zwei dieser Aufrufe: Mithilfe von `setHelmet` setzen wir einen Glasblock (`Material.GLASS`) auf den Kopf des Absenders. Das würde im normalen Spiel nicht ohne Weiteres gehen, da nur bestimmte Items auf den Kopf gesetzt werden können. Über `setChestplate` legen wir außerdem eine Goldbrustplatte (`Material.GOLDEN_CHESTPLATE`) an. Die Items, die für beide Aufrufe erstellt wurden, habe ich diesmal nicht in einer separaten Variable abgelegt, sondern direkt über `ItemStack.of` an die jeweilige Methode übergeben:

```
player.getInventory().setHelmet(ItemStack.of(Material.GLASS));
player.getInventory().setChestplate(ItemStack.of(Material.GOLDEN_CHESTPLATE));
```

Nun, da wir die drei Beispiele für einfache Items umgesetzt haben, wird es Zeit, den Befehl zu testen. Den gesamten Quellcode siehst du in Listing 9.1.

```
package dev.schlosser.bookplugin.commands;

import io.papermc.paper.command.brigadier.BasicCommand;
import io.papermc.paper.command.brigadier.CommandSourceStack;
import org.bukkit.Location;
import org.bukkit.Material;
import org.bukkit.entity.Player;
import org.bukkit.inventory.ItemStack;

public class ItemTestCommand implements BasicCommand {
  @Override
  public void execute(CommandSourceStack stack, String[] args) {
    if (!(stack.getSender() instanceof Player player)) {
      stack.getSender().sendPlainMessage("Nur für Spieler!");
      return;
    }
```

```
    ItemStack sword = ItemStack.of(Material.STONE_SWORD);
    player.getInventory().addItem(sword);

    ItemStack gold = ItemStack.of(Material.GOLD_INGOT, 10);
    Location loc = player.getLocation().add(0, 1.5, 3);
    player.getWorld().dropItemNaturally(loc, gold);

    player.getInventory().setHelmet(ItemStack.of(Material.GLASS));
    player.getInventory().setChestplate(
      ItemStack.of(Material.GOLDEN_CHESTPLATE));
  }
}
```

Listing 9.1 Quellcode für den ItemTest-Befehl

Nach dem Registrieren des Befehls und dem Übertragen des Plugins auf den Server kannst du `/itemtest` ausführen: Wie in Abbildung 9.1 zu sehen ist, erhältst du das Steinschwert direkt in dein Inventar. Zudem landen die Goldbarren unmittelbar vor dir. Außerdem wird dir ein schicker Glasblock über den Kopf gezogen und eine Goldbrustplatte angelegt.

Abbildung 9.1 Der Befehl fügt alle Items so hinzu, wie wir es beabsichtigt haben.

9.1.1 Die »ItemMeta«-Daten ausnutzen

Items zu verteilen ist nicht schwer, doch nun wollen wir weiteres Potenzial aus der Erstellung eines Items herausholen. Dafür nutzen wir die *ItemMeta*-Daten: Diese kannst du dir wie ein »Extrapaket« vorstellen, das bestimmte Daten für dein Item ablegt und dieses verändert. Dazu gehören beispielsweise der Anzeigename, eine Beschreibung, Verzauberungen oder andere Daten, wie die Abnutzung eines Werkzeugs. Wir sehen uns in einem neuen Befehl an, wie wir einige dieser Dinge zu einem besonderen Schwert hinzufügen, das dem Absender anschließend in das Inventar gelegt wird. Lege dafür eine neue Klasse `GiveSwordCommand` an und lege die Basis für den Befehl an, sodass nur Spieler diesen nutzen können.

Das generelle Prozedere für das Manipulieren der `ItemMeta`-Daten besteht darin, die standardmäßigen Daten, die das Item dafür bereitstellt, abzurufen und im Anschluss nach unseren Vorstellungen zu verändern. Genau diesen Ablauf setzen wir ein, um ein Diamantschwert zu verändern. Dafür erstellen wir zunächst wie gewohnt einen neuen `ItemStack` für das Schwert (`Material.DIAMON_SWORD`). Die `ItemMeta`-Daten kannst du über verschiedene Wege bearbeiten, der einfachste ist jedoch die Methode `editMeta`. Dieser kannst du einen Lambdaausdruck übergeben, der einen Parameter für die Meta-Daten erhält. Die folgende Quellcodevorlage würde es dir somit erlauben, alle Änderungen an den `ItemMeta`-Daten innerhalb der geschweiften Klammern auf der Variable `meta` auszuführen:

```
ItemStack sword = ItemStack.of(Material.DIAMOND_SWORD);
sword.editMeta(meta -> {

});
```

Nun können wir verschiedene Veränderungen an den `ItemMeta`-Daten vornehmen! Beginnen wir, indem wir den Namen des Schwertes anpassen. Dabei wird eine Component verwendet, die wie in Kapitel 8 gezeigt beispielsweise Farben und Textdekorationen enthalten kann. Zur Anschaulichkeit lege ich einen goldenen Namen an, den ich mithilfe der Methode `displayName` für die `ItemMeta`-Daten einsetze:

```
Component name = Component.text("Legendenschwert", NamedTextColor.GOLD);
meta.displayName(name);
```

Als Nächstes wollen wir eine Beschreibung für das Schwert festlegen. Dafür verwenden wir die sogenannte *Lore*, die mehrere Textzeilen unter dem Namen anzeigen kann. Dafür müssen wir die gewünschten Zeilen als Liste an die Methode `lore` übergeben. Ein einfacher Weg ist das Anlegen der Einträge über die Hilfsmethode `List.of`, die mehrere

übergebene Werte in solch eine Liste umwandelt. Für das Hinterlegen von Texten nutzen wir erneut Components, die du bei Bedarf ebenfalls einfärben und formatieren könntest, hier nutzen wir allerdings einfachen Text ohne weitere Anpassungen:

```
meta.lore(List.of(
  Component.text("Ein Schwert, das seit vielen Generationen existiert."),
  Component.text("Du hast es von einem begnadeten Schmied erhalten.")
));
```

Nun wird es magisch, denn das Schwert wird verzaubert! Du kannst eine neue Verzauberung über die Methode `addEnchant` hinzufügen, wobei der Methode drei Parameter übergeben werden:

- die Art der Verzauberung über den Datentyp `Enchantment`
- die Stärke der Verzauberung
- ein `boolean`, der angibt, ob das von Minecraft vorgegebene Maximallevel der Verzauberung überschritten werden darf

In diesem Fall übertreibe ich es: Als ersten Effekt füge ich Schärfe (`Enchantment.SHARPNESS`) auf Level 10 hinzu. Beachte, dass das von Minecraft vorgegebene Maximallevel 5 wäre, sodass der dritte Parameter `true` sein muss, damit das Level tatsächlich angewendet wird. Außerdem füge ich den Haltbarkeitseffekt (`Enchantment.UNBREAKING`) auf Stufe 3 hinzu. Hier ist der dritte Parameter eigentlich irrelevant, da das maximale Level nicht überschritten wird. Die beiden Verzauberungen sehen dann so aus:

```
meta.addEnchant(Enchantment.SHARPNESS, 10, true);
meta.addEnchant(Enchantment.UNBREAKING, 3, true);
```

Die `ItemMeta`-Daten sind nun fertig bearbeitet! Abschließend kannst du den `ItemStack` wie gewohnt in das Inventar des Absenders hinzufügen. Den gesamten Quellcode für das Beispiel mit einer kleinen Chatausgabe am Ende siehst du in Listing 9.2.

```
package dev.schlosser.bookplugin.commands;

import io.papermc.paper.command.brigadier.BasicCommand;
import io.papermc.paper.command.brigadier.CommandSourceStack;
import net.kyori.adventure.text.Component;
import net.kyori.adventure.text.format.NamedTextColor;
import org.bukkit.Material;
import org.bukkit.enchantments.Enchantment;
import org.bukkit.entity.Player;
import org.bukkit.inventory.ItemStack;
```

```java
import java.util.List;

public class GiveSwordCommand implements BasicCommand {
  @Override
  public void execute(CommandSourceStack stack, String[] args) {
    if (!(stack.getSender() instanceof Player player)) {
      stack.getSender().sendPlainMessage("Nur für Spieler!");
      return;
    }

    ItemStack sword = ItemStack.of(Material.DIAMOND_SWORD);
    sword.editMeta(meta -> {
      Component name = Component.text("Legendenschwert", NamedTextColor.GOLD);
      meta.displayName(name);

      meta.lore(List.of(
        Component.text("Ein Schwert, das seit vielen Generationen existiert."),
        Component.text("Du hast es von einem begnadeten Schmied erhalten.")
      ));
      meta.addEnchant(Enchantment.SHARPNESS, 10, true);
      meta.addEnchant(Enchantment.UNBREAKING, 3, true);
    });

    player.getInventory().addItem(sword);
    player.sendPlainMessage("Du hast das Legendenschwert erhalten.");
  }
}
```

Listing 9.2 Quellcode für den GiveSword-Befehl

Rufst du den Befehl nach dem Registrieren auf, wird dir das Schwert in dein Inventar gelegt (siehe Abbildung 9.2). Name, Beschreibung und Verzauberungen werden dabei erfolgreich gesetzt. Die Lore wird dabei standardmäßig in der Schriftfarbe Lila angezeigt.

Abbildung 9.2 Das Legendenschwert lässt sich wirklich sehen!

9.1.2 Übung 09.01: Heu zu Gold spinnen

Dein Plugin soll Heu eine besondere Bedeutung zukommen lassen: Wird ein Strohballen (`Material.HAY_BLOCK`) abgebaut, so soll nicht das Standardmaterial fallen gelassen werden, sondern zehn besondere Goldnuggets (`Material.GOLD_INGOT`). Das Goldnugget soll den unterstrichenen goldenen Namen »Gold-Heu« haben (mit `NamedTextColor.GOLD`) und mit der Verzauberung »Verbrennung« (`Enchantment.FIRE_ASPECT`) auf niedrigster Stufe ausgestattet sein.

Nutze für die Umsetzung das `BlockBreakEvent`. Mithilfe des Event-Parameters kannst du über den Aufruf `setDropItems(false)` zunächst festgelegen, dass das »Standard-Item« beim Abbauen nicht fallen gelassen wird. Anschließend kannst du an der Position des abgebauten Heublocks das Gold so fallen lassen wie in Listing 9.1.

9.1.3 Informationen über Persistent Data Container austauschen

Als Nächstes sehen wir uns ein praktisches Werkzeug an, das es erlaubt, an ein Item Daten »anzuhängen«, die du zu einem späteren Zeitpunkt wieder auslesen kannst. Die Technik trägt den Namen *Persistent Data Container* (kurz PDC) und ermöglicht es, Daten so zu speichern, dass nur dein Plugin sie sehen kann. PDC können dabei nicht nur an Items angehängt werden, sondern beispielsweise auch an Monster. In diesem Kapitel konzentrieren wir uns auf die Basics und lernen PDC bei Items kennen, die Kenntnisse lassen sich aber auf die anderen Bereiche übertragen.

Generell arbeitet die Technik nach einem Prinzip, das einer `HashMap` ähnelt: Daten werden unter einem bestimmten Schlüssel abgelegt, der nachträglich eingesetzt wird, um den entsprechenden Wert wieder zu laden. Der Datentyp eines Wertes muss dabei bereits beim Speichern angegeben werden, wobei PDC auf bestimmte Datentypen limitiert sind.

Für ein erstes Beispiel setzen wir eine neue Funktion für das Plugin um: Wir speichern zu jedem gecrafteten Item, welcher Spieler das Item gebaut hat. Wenn mit einem Item ein Rechtsklick gemacht wird und ein Spieler als »Crafter« gespeichert ist, dann wird der Name des ursprünglichen Herstellers im Chat ausgegeben. Dafür speichern wir mithilfe eines PDC die UUID eines Spielers auf einem Item, sobald es gecraftet wird. Leg dafür eine neue Klasse mit dem Namen `CraftNameListener` an. Bevor es an eine Event-Methode geht, legen wir eine Variable innerhalb der Klasse an: Wir benötigen wie bei einer `HashMap` einen *Schlüssel*, unter dem die Daten gesichert werden. Dafür kommt ein `NamespacedKey` zum Einsatz. Dieser besteht aus zwei Komponenten:

1. einem »Namespace«, also eine Bezeichnung für einen Bereich, in dem dieser Schlüssel aktiv ist. Hier wird im Regelfall eine einheitliche Bezeichnung für alle Schlüssel eines Plugins verwendet.
2. dem Namen für den Schlüssel

Legen wir einen neuen Schlüssel an, unter dem wir anschließend die UUID des »Crafters« ablegen. Als Namespace verwende ich dabei einheitlich den Namen meines Plugins, `bookplugin`, für den Bezeichner `crafter_uuid`. Beide Daten werden als Parameter an den Aufruf `new NamespacedKey` übergeben, den Schlüssel speichere ich in einer passenden Instanzvariable:

```
public class CraftNameListener implements Listener {
  NamespacedKey key = new NamespacedKey("bookplugin", "crafter_uuid");
}
```

Nun folgt das erste Event, das beim Craften eines Items ausgelöst wird und somit die UUID im PDC speichert. Dazu dient das `CraftItemEvent`, das beim Bauen eines Items ausgelöst wird. Im Event überprüfen wir, ähnlich wie beim Auslösen eines Befehls, ob ein Spieler das Crafting durchgeführt hat. Dafür kannst du die `instanceof`-Überprüfung auf die Methode `getWhoClicked` des Event-Parameters durchführen. Anschließend laden wir uns den neu gecrafteten `ItemStack` über `getResult` des Rezeptes, was über `getRecipe` abgerufen wird. Da der PDC über die `ItemMeta`-Daten angelegt wird, bearbeiten wir diese mit der gewohnten Methode `editMeta`:

```
public void onItemCraft(CraftItemEvent event) {
  if (!(event.getWhoClicked() instanceof Player player)) {
    return;
  }

  ItemStack result = event.getRecipe().getResult();
  result.editMeta(meta -> {

  });
}
```

Als Nächstes müssen wir unter dem zu Beginn angelegten Schlüssel (`NamespacedKey`) neue Daten in den PDC setzen, den wir wiederum in den `ItemMeta`-Daten finden. Dafür nutzen wir unter dem Methodenaufruf `getPersistentDataContainer` die Methode `set`. Der Methode wird als erster Parameter der gerade angelegte Schlüssel übergeben. Als zweiter Parameter folgt der Datentyp der gespeicherten Daten. Wir speichern die UUID als Text, was `PersistentDataType.STRING` entspricht. Andere verfügbare Datentypen sind

beispielsweise BOOLEAN, INTEGER, FLOAT oder DOUBLE. Als letzter Parameter folgt danach der Wert, der gespeichert werden soll. Wir verwenden den Aufruf toString auf die UUID des Spielers, was den UUID-Datentyp passenderweise in einen String umwandelt:

```
meta.getPersistentDataContainer().set(
  key, PersistentDataType.STRING, player.getUniqueId().toString());
```

Um das Item schlussendlich als »Ergebnis« des Craftings zu übernehmen, benötigst du für den event-Parameter die Methode setCurrentItem, die im Hintergrund das Item, das gerade gecraftet wurde, mit deinem veränderten ItemStack austauscht:

```
event.setCurrentItem(result);
```

Nun fehlt noch der zweite Listener: Bei ihm überprüfen wir, ob ein Item rechtsgeklickt wurde, und versuchen in diesem Fall, den PDC zu laden. Dafür kommt das PlayerInteractEvent zum Einsatz, das beim Klicken mit der Maus ausgelöst wird. Als erste Abfrage überprüfen wir, ob ein Rechtsklick durchgeführt wurde. Dafür sind zwei Vergleiche mit der Methode getAction des Event-Parameters notwendig, nämlich mit Action.RIGHT_CLICK_BLOCK und Action.RIGHT_CLICK_AIR. Während erstere Aktion überprüft, ob der Rechtklick auf einen Block in Blickrichtung des Spielers durchgeführt wurde, steht die zweite Abfrage für das Rechtsklicken in der Luft. Der ItemStack, mit dem geklickt wurde, wird über die Methode getItem des Events abgerufen. Im ersten Schritt überprüfen wir, ob ein Item gefunden wurde (also ob dieser Abruf nicht null zurückgibt) und ob ItemMeta-Daten gefunden werden, die im Anschluss auf den PDC überprüft werden können. Für letztere Überprüfung kommt die Methode hasItemMeta zum Einsatz, die nur dann true zurückgibt, wenn der ItemStack ItemMeta-Daten verwendet. Die ItemMeta-Daten selbst rufst du dann über getItemMeta ab:

```
public void onRightClick(PlayerInteractEvent event) {
  if (event.getAction() != Action.RIGHT_CLICK_BLOCK
    && event.getAction() != Action.RIGHT_CLICK_AIR) {
    return;
  }

  ItemStack item = event.getItem();
  if (item == null || !item.hasItemMeta()) {
    return;
  }
  ItemMeta meta = item.getItemMeta();
}
```

Aus den ItemMeta-Daten heraus kannst du nun den PDC erneut über getPersistentDataContainer in einer Variable zwischenspeichern. Zunächst überprüfen wir für den PDC, ob Daten unter dem anfangs angelegten Schlüssel abgelegt sind. Dafür wird der Methode has des PDC der Schlüssel sowie der Datentyp, den die gesuchten Daten haben, angegeben. In unserem Fall suchen wir nach einem PersistentDataType.STRING, da wir diesen Typ zum Speichern der Daten verwendet haben:

```
PersistentDataContainer container = meta.getPersistentDataContainer();
if (container.has(key, PersistentDataType.STRING)) {
}
```

Innerhalb der if-Abfrage kannst du die Daten nun aus dem PDC laden, da du weißt, dass der String unter dem entsprechenden Schlüssel gespeichert wurde. Dafür verwenden wir die Methode get und die gleichen Parameter wie bei has. Um den String in eine UUID zu verwandeln, nutzen wir die Methode UUID.fromString:

```
String uuidStr = container.get(key, PersistentDataType.STRING);
UUID crafterUUID = UUID.fromString(uuidStr);
```

Im letzten Schritt musst du aus der UUID nur noch zum passenden Spieler kommen. Da es für dieses Beispiel egal ist, ob sich der Zielspieler gerade auf dem Server befindet, verwenden wir erstmalig nicht getOnlinePlayer. Stattdessen nutzen wir den Datentyp OfflinePlayer, der einen Spieler repräsentieren kann, der gerade nicht zwingend auf dem Server ist. Über die Methode getOfflinePlayer des Servers kannst du aus der UUID eine passende Variable ableiten und die bekannte Methode getName einsetzen, um den Namen des Crafters schlussendlich an den Spieler zu senden, der mit dem Item in der Hand geklickt hat.

```
OfflinePlayer crafter = event.getPlayer().getServer()
  .getOfflinePlayer(crafterUUID);
event.getPlayer().sendPlainMessage("Item von " + crafter.getName());
```

Das war eine Menge Stoff, doch nun ist es geschafft. Nimm dir einen Moment Zeit, um zu verstehen, wie der PDC genutzt wird, um Daten zu speichern und nachträglich abzurufen. Der vollständige Quellcode für die Listener-Klasse mit den beiden Event-Behandlungen ist in Listing 9.3 zu sehen. Dabei habe ich die Imports eingekürzt, da es wirklich viele importierte Klassen für die Logik benötigt.

```
package dev.schlosser.bookplugin.listeners;

// Imports...
```

```java
public class CraftNameListener implements Listener {
  NamespacedKey key = new NamespacedKey("bookplugin", "crafter_uuid");

  @EventHandler
  public void onItemCraft(CraftItemEvent event) {
    if (!(event.getWhoClicked() instanceof Player player)) {
      return;
    }

    ItemStack result = event.getRecipe().getResult();
    ItemMeta meta = result.getItemMeta();

    result.editMeta(meta -> {
      meta.getPersistentDataContainer().set(
        key, PersistentDataType.STRING, player.getUniqueId().toString());
    });
    event.setCurrentItem(result);
  }

  @EventHandler
  public void onRightClick(PlayerInteractEvent event) {
    if (event.getAction() != Action.RIGHT_CLICK_BLOCK &&
      event.getAction() != Action.RIGHT_CLICK_AIR) {
      return;
    }

    ItemStack item = event.getItem();
    if (item == null || !item.hasItemMeta()) {
      return;
    }

    ItemMeta meta = item.getItemMeta();

    PersistentDataContainer container = meta.getPersistentDataContainer();
    if (container.has(key, PersistentDataType.STRING)) {
      String uuidStr = container.get(key, PersistentDataType.STRING);
      UUID crafterUUID = UUID.fromString(uuidStr);

      OfflinePlayer crafter = event.getPlayer().getServer()
        .getOfflinePlayer(crafterUUID);
```

```
      event.getPlayer().sendPlainMessage("Item von " + crafter.getName());
    }
  }
}
```

Listing 9.3 Quellcode für die Crafting-Anzeige

Registriere den neuen Listener und lade das Plugin erneut auf den Server. Nun wird es spannend: Wenn du ein neues Item craftest, kannst du es rechtsklicken und siehst (hoffentlich) wie in Abbildung 9.3 deinen eigenen Namen. Dies funktioniert genauso, wenn du das Item eines anderen Spielers anschaut, selbst wenn sich dieser gerade nicht auf dem Server befindet.

Abbildung 9.3 Beim Rechtsklicken mit der gecrafteten Spitzhacke erscheint die gewünschte Ausgabe im Chat.

> **Weitere Verwendungen von PDC**
>
> Im Laufe des Buches werde ich an passenden Stellen Empfehlungen oder Beispiele geben, wenn PDC eingesetzt werden könnten. Es handelt sich um eine praktische Technik, mit der du viele komplexe Ideen umsetzen kannst. Immerhin kannst du für verschiedenste Dinge Daten »verstecken«, die nur dein Plugin abrufen kann.

9.2 Virtuelle Inventare erstellen

Nun wird es Zeit, nicht nur künstlich Items zu erstellen, sondern auch Inventare. Da kannst du beispielsweise Mitgliedern deines Servers erlauben, sich Items aus einem solchen Inventar zu nehmen. Außerdem kannst du virtuelle Inventare einsetzen, um vereinfachte Menüs anzubieten, durch die man sich durch das Anklicken von Items navigieren kann. Für beide Fälle entwickeln wir in diesem Abschnitt je ein Beispiel,

wobei wir mit den Grundlagen zur Erstellung eines Inventars starten, indem wir auf Befehl mehrere Items bereitstellen. Diese kann sich der Absender dann einfach nehmen, wie in einem Schlaraffenland. Erstelle dafür eine neue Klasse `LootCommand`, in der du die Basis für den Befehl, der nur von Spielern ausgeführt werden darf, umsetzt.

Für die Logik starten wir mit der Erstellung eines Inventars, das anschließend dem Absender präsentiert werden soll. Dafür verwenden wir die Methode `createInventory` des Servers. Dieser Methode werden zwei Parameter übergeben:

- Der erste Parameter steht für einen »Besitzer« des Inventars, was sinnvoll ist, wenn das Inventar einem bestimmten Monster, einer Truhe oder einem Spieler zugeordnet werden soll. Wenn es keinen bestimmten Besitzer für das Inventar geben soll, kann einfach `null` eingesetzt werden.
- Der zweite Parameter legt die Größe des Inventars fest. Bei diesem »Standardinventar« besteht jede Reihe aus 9 Slots, sodass ein Vielfaches von 9 als Zahl verwendet werden sollte.

Für das Beispiel benötigen wir keinen Besitzer, sodass der erste Parameter `null` ist. Außerdem wollen wir zwei Reihen von Slots, sodass wir insgesamt 18 Plätze benötigen. Besonders praktisch: Wir können einfach die Rechnung `9*2` in den Aufruf einsetzen, sodass die Zahl 18 automatisch berechnet wird und wir beim Berechnen keinen Fehler verursachen:

```
Inventory inv = stack.getSender().getServer().createInventory(null, 9*2);
```

Maximale Größe

Beachte, dass eine Truhe maximal 54 Slots, also sechs Reihen, enthalten kann.

Das Inventar können wir etwas weiter verschönern, indem wir einen farbigen Titel verwenden. Dafür wird ein dritter Parameter in Form einer Component angehangen. Für das Beispiel verwende ich einen grünen Text, den ich an die Methode weitergebe:

```
Component title = Component.text("Ausbeute", NamedTextColor.GREEN);
Inventory inv = stack.getSender().getServer().createInventory(null, 9*2, title);
```

Andere Formen von Inventaren

Der gezeigte Aufruf erzeugt eine »Standard-Inventaransicht«, die neben den Inventar-Slots ziemlich unspektakulär wirkt. Möchtest du einen bestimmten Inventartyp verwenden, kannst du dafür nach dem ersten Parameter des »Besitzers« einen `InventoryType` übergeben. Paper stellt für verschiedene Arten wie Werkbänke, Öfen oder besondere

Truhen sowie viele weitere Anwendungsfälle eigene Typen bereit. Der folgende Code würde mit InventoryType.ANVIL die Ansicht eines Amboss-Inventars erzeugen:

```
Inventory anvil = stack.getSender().getServer()
  .createInventory(null, InventoryType.ANVIL);
```

Der zuvor gezeigte Standardweg setzt automatisch InventoryType.CHEST ein, was eine Truhenansicht erzeugt.

Bevor wir das Inventar öffnen, müssen wir noch die Beute für den Absender hinzufügen. Dafür legen wir drei einfache Items ab: Den ersten ItemStack, 20 Goldbarren, fügen wir wie zuvor über die Methode addItem des Inventars hinzu. Damit wird das Item an die erste freie Stelle, also in den ersten Slot, platziert. Für das zweite Item verwenden wir die Methode setItem, womit das Item in einen festen Slot gesetzt wird. Wie in einem Array hat jeder Slot einen festen Index, wobei der erste Slot ganz oben links den Index 0 hat und von da an aufsteigend gezählt wird. Der erste Slot der zweiten Reihe hätte somit den Index 9, da das letzte Element der ersten Reihe den Index 8 zugewiesen bekommt. Für zwei Reihen, wie in diesem Beispiel, siehst du die Slot-Indizes in Abbildung 9.4.

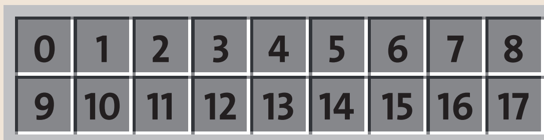

Abbildung 9.4 Slot-Indizes für ein virtuelles Inventar mit zwei Zeilen

Das dritte Item wird wieder über setItem gesetzt, diesmal wählen wir den Slot aber zufällig. Dafür wird mit new Random().nextInt eine Zufallszahl zwischen 0 (dem ersten Slot-Index) und 17 (dem größten Slot-Index) bestimmt. Beachte, dass der Diamant, den wir setzen, dabei ein anderes Item überschreiben kann. Wird beispielsweise zufällig der Index 6 berechnet, werden die zuvor eingesetzten Items auf dem Slot mit dem Index 6 überschrieben. Der Quellcode zum Hinzufügen der Items sieht so aus, wobei ich verschiedene Materialien für die Ausbeute eingesetzt habe:

```
inv.addItem(ItemStack.of(Material.GOLD_INGOT, 20));
inv.setItem(6, ItemStack.of(Material.EMERALD, 5));
int randomIndex = new Random().nextInt(9*2);
inv.setItem(randomIndex, ItemStack.of(Material.DIAMOND));
```

Im letzten Schritt musst du das Inventar nur noch für den Absender öffnen. Dafür kommt die Methode openInventory zum Einsatz, der das virtuelle Inventar, das wir gerade erstellt und befüllt haben, übergeben wird:

```
player.openInventory(inv);
```

Der gesamte Quellcode für das Beispiel deines ersten virtuellen Inventars, das durch einen Befehl geöffnet wird, siehst du in Listing 9.4.

```
package dev.schlosser.bookplugin.commands;

import io.papermc.paper.command.brigadier.BasicCommand;
import io.papermc.paper.command.brigadier.CommandSourceStack;
import net.kyori.adventure.text.Component;
import net.kyori.adventure.text.format.NamedTextColor;
import org.bukkit.Bukkit;
import org.bukkit.Material;
import org.bukkit.entity.Player;
import org.bukkit.inventory.Inventory;
import org.bukkit.inventory.ItemStack;

import java.util.Random;

public class LootCommand implements BasicCommand {
  @Override
  public void execute(CommandSourceStack stack, String[] args) {
    if (!(stack.getSender() instanceof Player player)) {
      stack.getSender().sendPlainMessage("Nur für Spieler!");
      return;
    }

    Component title = Component.text("Ausbeute", NamedTextColor.GREEN);
    Inventory inv = stack.getSender().getServer()
      .createInventory(null, 9*2, title);

    inv.addItem(ItemStack.of(Material.GOLD_INGOT, 20));
    inv.setItem(6, ItemStack.of(Material.EMERALD, 5));
    int randomIndex = new Random().nextInt(9*2);
    inv.setItem(randomIndex, ItemStack.of(Material.DIAMOND));
    player.openInventory(inv);
  }
}
```

Listing 9.4 Quellcode für den Loot-Befehl

Das Beuteinventar kannst du nach dem Registrieren des Befehls mit `/loot` öffnen. Das Ergebnis siehst du in Abbildung 9.5: Während Goldbarren und Smaragde immer im glei-

chen Slot liegen, wird der Diamant auf einen zufälligen Slot gelegt. Die Beute kannst du dir nun in dein eigenes Inventar ziehen.

Abbildung 9.5 Das Beuteinventar enthält die seltenen Items.

9.2.1 Auf Interaktionen mit virtuellen Inventaren reagieren

Eine weitere Schokoladenseite virtueller Inventare kommt zum Vorschein, wenn du ein Inventar als »Menü«, mit dem Spieler interagieren können, gestaltest. Dafür werden einzelnen Items, die sich im Menü befinden, beim Klicken Aktionen zugewiesen. Wie so oft gibt es kaum Grenzen für deine Kreativität, sodass ein Klick vom Teleportieren bis hin zum automatischen Bauen eines Gebäudes alles auslösen kann. Wir realisieren ein einfaches Beispiel, bei dem ein Befehl ein Inventar mit verschiedenen »Aktions-Items« öffnet. Jedes Item erfüllt dabei eine eigene Aktion, nämlich das Auffüllen der Herzen, das Hinzufügen einiger Erfahrungspunkte und das Teleportieren zum Spawn-Punkt. Die Aktionen erfassen wir über die in Abschnitt 9.1.3 vorgestellten Persistent Data Container. Die Logik teilen wir in zwei Bereiche: Ein Befehl öffnet lediglich das Inventar und fügt darin die Items mit den PDC hinzu. Die Abfrage des Menüklicks erfolgt anschließend über einen separaten Listener, der versucht, die Daten aus den Items im Menü zu laden und die zugehörige Aktion auszuführen.

> **Andere Wege zur Umsetzung eines Menüs**
>
> Der hier verwendete Weg eines PDC ist nur eine von verschiedenen Möglichkeiten, auf das Klicken in Inventarmenüs zu reagieren. Simplere Wege, wie das Überprüfen des angeklickten Item-Materials oder des gesetzten Namens in den `ItemMeta`-Daten, sind fehleranfälliger, weswegen ich sie im Buch nicht näher behandle.

Starten wir mit dem Befehl, den ich in einer neuen Klasse `ActionInvCommand` anlege. In der Klasse benötigen wir einen `NamespacedKey`, der für die Item-Einträge des PDC gelten soll. Ich lege ihn unter dem gewohnten Namespace mit einem neuen Namen an:

```
public NamespacedKey actionKey = new NamespacedKey("bookplugin", "action");
```

Im Anschluss geht es an die `execute`-Methode. Baue hier das Grundgerüst, damit lediglich Spieler den Befehl ausführen dürfen. Im Anschluss erstellen wir ein neues virtuelles Inventar ohne Besitzer, das diesmal die Form eines Trichters (`InventoryType.HOPPER`) erhalten soll. Außerdem setze ich einen Titel, der am oberen Menürand angezeigt wird:

```
Inventory inv = stack.getSender().getServer().createInventory(
  null, InventoryType.HOPPER, Component.text("Wähle eine Aktion"));
```

Gleich erstellen wir die drei Items, die jeweils beim Anklicken eine bestimmte Aktion auslösen sollen. Damit das leichter von der Hand geht, schreibe ich eine kleine Hilfsmethode, die einen `ItemStack` mit einem bestimmten Material sowie einem Anzeigenamen zurückgibt. Als dritten Parameter akzeptiert die Methode zudem einen `String`, der als `PersistentDataType.STRING` auf den PDC des neuen Items gesetzt wird:

```
public ItemStack createItem(Material material, String displayName,
    String actionValue) {
  ItemStack item = ItemStack.of(material);
  item.editMeta(meta -> {
      meta.displayName(Component.text(displayName));
      meta.getPersistentDataContainer().set(
        actionKey, PersistentDataType.STRING, actionValue);
    });
  return item;
}
```

Die neue Methode können wir nun einsetzen, um drei Items in das Inventar zu setzen. Dafür verwende ich den ersten, den dritten und den fünften Slot (mit den Indizes 0, 2 und 4) und weise jeweils passende Item-Typen und Namen zu. Besonders wichtig ist der letzte Parameter: Die hier übergebenen `String`-Werte (fett markiert) benötigst du später beim Listener, um zu überprüfen, welches Item angeklickt wurde, und so die passende Aktion auszuführen:

```
inv.setItem(0, createItem(Material.GOLDEN_APPLE, "Heilen", "heal"));
inv.setItem(2, createItem(Material.EXPERIENCE_BOTTLE, "Erfahrung", "exp"));
inv.setItem(4, createItem(Material.ENDER_PEARL, "Zum Spawn", "teleport"));
```

Das Inventar ist nun fertig befüllt und kann dem Absender des Befehls angezeigt werden. Dazu kommt, wie gewohnt, die Methode `openInventory` zum Einsatz:

```
player.openInventory(inv);
```

Den gesamten Quellcode für den ersten Teil des Menüs, nämlich den gerade entwickelten Befehl, siehst du in Listing 9.5.

```
package dev.schlosser.bookplugin.commands;

import io.papermc.paper.command.brigadier.BasicCommand;
import io.papermc.paper.command.brigadier.CommandSourceStack;
import net.kyori.adventure.text.Component;
import org.bukkit.Bukkit;
import org.bukkit.Material;
import org.bukkit.NamespacedKey;
import org.bukkit.entity.Player;
import org.bukkit.event.inventory.InventoryType;
import org.bukkit.inventory.Inventory;
import org.bukkit.inventory.ItemStack;
import org.bukkit.inventory.meta.ItemMeta;
import org.bukkit.persistence.PersistentDataType;

public class ActionInvCommand implements BasicCommand {
  public NamespacedKey actionKey = new NamespacedKey("bookplugin", "action");

  @Override
  public void execute(CommandSourceStack stack, String[] args) {
    if (!(stack.getSender() instanceof Player player)) {
      stack.getSender().sendPlainMessage("Nur für Spieler!");
      return;
    }

    Inventory inv = stack.getSender().getServer().createInventory(
      null, InventoryType.HOPPER, Component.text("Wähle eine Aktion"));

    inv.setItem(0, createItem(Material.GOLDEN_APPLE, "Heilen", "heal"));
    inv.setItem(2, createItem(Material.EXPERIENCE_BOTTLE, "Erfahrung", "exp"));
    inv.setItem(4, createItem(Material.ENDER_PEARL, "Zum Spawn", "teleport"));
```

```
    player.openInventory(inv);
  }

  private ItemStack createItem(Material material, String displayName,
    String actionValue) {
    ItemStack item = ItemStack.of(material);
    item.editMeta(meta -> {
      meta.displayName(Component.text(displayName));
      meta.getPersistentDataContainer().set(
        actionKey, PersistentDataType.STRING, actionValue);
    });
    return item;
  }
}
```

Listing 9.5 Quellcode für den »ActionInv«-Befehl

Weiter geht es mit dem zweiten Teil des Beispiels, in dem wir auf Klicks im Menü reagieren. Lege dafür eine neue Klasse `ActionInvListener` an. Um die im PDC gesetzten Daten abzurufen, benötigen wir in diesem Listener den gleichen `NamespacedKey` wie in Listing 9.5. Diesen lege ich als Variable in der Listener-Klasse an:

```
public NamespacedKey actionKey = new NamespacedKey("bookplugin", "action");
```

Es folgt die Methode, mit der wir auf das Anklicken eines Items reagieren: Dafür verwenden wir das `InventoryClickEvent`, das immer dann ausgelöst wird, wenn mit einem Inventar interagiert wird. Im ersten Schritt überprüfen wir, ob die Interaktion tatsächlich durch einen Spieler vorgenommen wurde, alle anderen Aufrufe sind nicht relevant. Den Aufruf `getWhoClicked` des Event-Parameters kannst du dabei mit `instanceof` wieder einmal so überprüfen wie bei anderen Beispielen. Handelt es sich um einen Spieler, wird er in der Variable `player` abgelegt, andernfalls erfolgt ein Abbruch der Methode durch Early Return:

```
public void onInventoryClick(InventoryClickEvent event) {
  if (!(event.getWhoClicked() instanceof Player player)) {
    return;
  }
}
```

Nun geht es an das Item, das angeklickt wurde und für das wir versuchen, die PDC-Daten auszulesen, die nur dann gesetzt sind, wenn es sich um das Menü aus dem Befehl

(Listing 9.5) handelt. Das angeklickte Item rufst du zunächst über die Methode getCurrentItem des Event-Parameters ab. In der ersten Überprüfung stellen wir fest, dass das Item nicht null ist und ein ItemMeta-Datum (das potenziell den passenden PDC enthält) besitzt. Ist eine der Bedingungen nicht gegeben, wird die Methode abgebrochen:

```
ItemStack clicked = event.getCurrentItem();
if (clicked == null || !clicked.hasItemMeta()) {
  return;
}
```

Über die ItemMeta-Daten versuchen wir nun, unter dem angegebenen NamespacedKey eine Aktion herauszufinden. Statt einer Überprüfung mit der Methode has des PDC wählen wir diesmal einen anderen Weg und vergleichen, ob die geladene Aktion den Wert null hat. Ist dies der Fall, wird die Methode abgebrochen. Wurde ein Wert geladen, wissen wir, dass ein Item in unserem Menü angeklickt wurde. In diesem Fall ist zunächst wichtig, dass wir das Event mittels setCancelled abbrechen, damit der angeklickte Eintrag nicht versetzt oder ins eigene Inventar verschoben werden kann:

```
ItemMeta meta = clicked.getItemMeta();
String action = meta.getPersistentDataContainer().get(
  actionKey, PersistentDataType.STRING);
if (action == null) {
  return;
}
event.setCancelled(true);
```

Nun muss nur noch, je nach angeklicktem Aktionsschlüssel, die passende Aktion ausgeführt werden. Dafür muss die zuvor angelegte Variable action mit jedem möglichen String, der in Listing 9.5 verwendet wurde, verglichen werden, wofür wir ein switch-Statement verwenden. Beim Schlüssel heal heilen wir exakt zehn Herzen, während bei exp 50 Erfahrungspunkte über die Methode giveExp vergeben werden. Das Teleportations-Item teleportiert den Spieler an die Standard-Spawn-Position, die du über die Methode getSpawnLocation der zugehörigen Welt abrufen kannst. Außerdem werden für alle Fälle passende Nachrichten ausgegeben:

```
switch (action) {
  case "heal" -> {
    player.setHealth(20);
    player.sendPlainMessage("Du wurdest geheilt!");
  }
  case "exp" -> {
```

```
    player.giveExp(50);
    player.sendPlainMessage("Du hast Erfahrung erhalten!");
  }
  case "teleport" -> {
    player.teleport(player.getWorld().getSpawnLocation());
    player.sendPlainMessage("Du bist nun am Spawn!");
  }
}
```

Nachdem eine Aktion gewählt wurde, ist der Befehl abgeschlossen, und du kannst das offene Inventar über die Methode `closeInventory` schließen:

```
player.closeInventory();
```

Der vollständige Quellcode für die Listener-Klasse des zugehörigen ActionInv-Befehls aus Listing 9.5 ist in Listing 9.6 aufgeführt.

```
package dev.schlosser.bookplugin.listeners;

import org.bukkit.NamespacedKey;
import org.bukkit.entity.Player;
import org.bukkit.event.EventHandler;
import org.bukkit.event.Listener;
import org.bukkit.event.inventory.InventoryClickEvent;
import org.bukkit.inventory.ItemStack;
import org.bukkit.inventory.meta.ItemMeta;
import org.bukkit.persistence.PersistentDataType;

import java.util.List;

public class ActionInvListener implements Listener {
  public NamespacedKey actionKey = new NamespacedKey("bookplugin", "action");

  @EventHandler
  public void onInventoryClick(InventoryClickEvent event) {
    if (!(event.getWhoClicked() instanceof Player player)) {
      return;
    }

    ItemStack clicked = event.getCurrentItem();
    if (clicked == null || !clicked.hasItemMeta()) {
```

```java
      return;
    }

    ItemMeta meta = clicked.getItemMeta();
    String action = meta.getPersistentDataContainer().get(
      actionKey, PersistentDataType.STRING);
    if (action == null) {
      return;
    }
    event.setCancelled(true);

    switch (action) {
      case "heal" -> {
        player.setHealth(20);
        player.sendPlainMessage("Du wurdest geheilt!");
      }
      case "exp" -> {
        player.giveExp(50);
        player.sendPlainMessage("Du hast Erfahrung erhalten!");
      }
      case "teleport" -> {
        player.teleport(player.getWorld().getSpawnLocation());
        player.sendPlainMessage("Du bist nun am Spawn!");
      }
    }

    player.closeInventory();
  }
}
```

Listing 9.6 Quellcode für den ActionInv-Listener

Anlegen von plugin-internen »Inventarbesitzern«

Statt der direkten Verwendung des angeklickten Items erfolgt häufig eine Überprüfung des Inventars. So kannst du den Item-Klick nur dann behandeln, wenn du weißt, dass sich der Spieler tatsächlich in deinem Menü befindet. Einfache Ansätze verwenden beispielsweise einen Vergleich des Inventarnamens. Dieser Ansatz ist allerdings nicht empfehlenswert, da beispielsweise andere Plugins den exakt gleichen Namen verwenden können und die Menüs somit als identisch erkannt werden, obwohl lediglich der Name

zufällig übereinstimmt. Stattdessen bietet Paper die Möglichkeit, eigene »Inventarbesitzer« anzulegen, mit denen du einen solchen Vergleich anstellen kannst.

Dieser Ansatz ist allerdings etwas fortgeschrittener, sodass ich ihn hier zunächst nicht näher beleuchte. Alle Informationen findest du zur eigenen Recherche unter *https://docs.papermc.io/paper/dev/custom-inventory-holder/*.

Sobald du alle Bestandteile des Beispiels registriert hast, also Befehl und Listener, kannst du das Menü ausprobieren: Bei der Verwendung von `/actioninv` öffnet sich das Inventar wie in Abbildung 9.6. Sobald du einen der drei Einträge wählst, wird die passende Aktion ausgeführt.

Abbildung 9.6 Das Aktionsinventar erfüllt seinen Zweck und kann mithilfe der PDC genau zuordnen, welche Aktion du gerade gewählt hast.

9.2.2 Übung 09.02: Glückseliger Goldblock

Programmiere einen Listener, der beim Rechtsklicken eines Goldblocks reagiert. Dabei soll ein virtuelles Inventar mit sechs Zeilen geöffnet werden, wobei in jedem Slot eine bestimmte Anzahl von Goldblöcken liegt. Im ersten Slot soll es genau ein Goldblock sein, im zweiten Slot zwei Blöcke und so weiter. Beim Anklicken eines Goldblocks soll der Spieler so viele Goldbarren erhalten, wie es der Anzahl der Goldblöcke im Menü entspricht.

Einen angeklickten Block kannst du im `PlayerInteractEvent` über die Methode `getClickedBlock` des Event-Parameters abrufen, wobei `null` zurückgegeben wird, wenn kein Block geklickt wurde.