Programmieren lernen mit Minecraft-Plugins

Wie Du mit Java und CanaryMod Deine Welt erweiterst

von
Andy Hunt

1. Auflage

dpunkt.verlag 2015

Verlag C.H. Beck im Internet:
www.beck.de
ISBN 978 3 86490 220 8

Zu Inhaltsverzeichnis

schnell und portofrei erhältlich bei beck-shop.de DIE FACHBUCHHANDLUNG

Kapitel 5

Objekte

Dein Werkzeugkasten

In diesem Kapitel sehen wir uns eine Reihe neuer Java-Werkzeuge an:

- *Objekte* verwenden: Kombinationen aus Variablen und Funktionen
- den richtigen *Importvorgang* für ein Paket verwenden
- Objekte mit `new` erstellen

Wahrscheinlich hast du schon von objektorientierter Programmierung gehört oder davon, dass Java eine objektorientierte Sprache ist. Darum geht es in diesem Kapitel: die Verwendung von Objekten, um Elemente aus dem Spiel Minecraft darzustellen, seien es nun Spieler oder Kühe.

Im Code kannst du alles tun, was du auch im Spiel tun kannst, und sogar noch einiges mehr. Letzten Endes arbeitest du dabei meistens mit drei Arten von Dingen: Blöcken, Gegenständen und Entitäten (siehe unten).

In Minecraft ist alles ein Objekt

Die Welt von Minecraft besteht aus *Blöcken*. An jeder Position im Spiel gibt es einen Block, der aus Luft oder aus einem anderen Material besteht. Blöcke gibt es sowohl in der Welt als auch im Inventar des Spielers. Alles, was sich im Inventar eines Spielers befindet, wird durch einen *Gegenstand* dargestellt.

In dieser Welt aus Blöcken gibt es auch *Entitäten* (*entities*, im deutschsprachigen Minecraft eigentlich *Objekte* genannt), wozu u. a. Spieler, Creepers und Kühe gehören. Gegenstände, die sich in Bewegung befinden, sind ebenfalls Entitäten, beispielsweise ein fliegender Pfeil oder ein geworfener Schneeball oder Trank. All dies sind außerdem Objekte (im Sinne von Java).

Alles in unseren Plugins ist ein Objekt: Positionen, Blöcke, Entitäten, Kühe, Creeper, Spieler, ja, sogar das Plugin selbst.

Damit geht es jetzt erst richtig los! Du hast Variablen, um Daten festzuhalten, und Anweisungen in Funktionen, wozu auch einige Steueranweisungen gehören, um einzelne Teile des Codes zu wiederholen und Entscheidungen zu treffen. Als Nächstes siehst du, wie du all dies zu Objekten zusammensetzt.

Probiere es selbst aus!

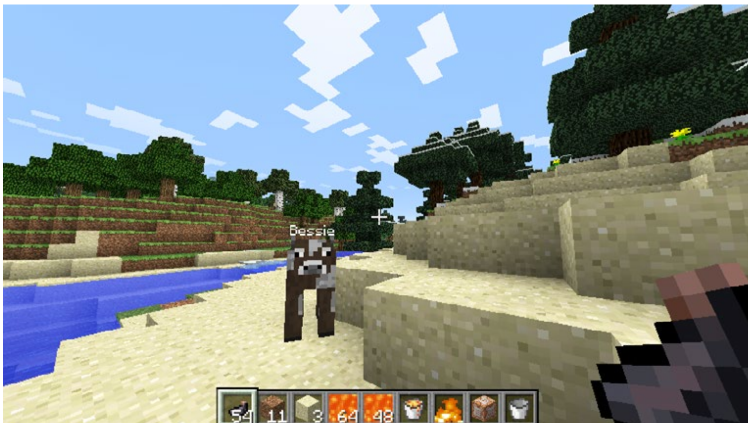
Versuchen wir uns an einer kleinen Veranschaulichung von Objekten in Minecraft. In dem heruntergeladenen Code findest du das Plugin `NameCow`. Installiere es:

```
$ cd Desktop
$ cd code/NameCow
$ ./build.sh
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

Halte den Server an, starte ihn erneut und stelle die Clientverbindung her. Jetzt kannst du den neuen Befehl `newcow` verwenden. Er spawnt eine neue Kuh und gibt ihr einen Namen. Im Minecraft-Client gibst du den Befehl ein, gefolgt vom gewünschten Namen der Kuh:

```
/namecow Bessie
/namecow Elise
/namecow Babe
```

Die Kühe erscheinen im Spiel. Wenn du genauer hinsiehst, kannst du jeweils ihren Namen erkennen.



Jedes Mal, wenn du dieses Plugin ausführst, spawnt es ein neues `Cow`-Objekt. Jede Kuh, die du spawnst, verfügt über interne Variablen, die ihren Status festhalten: den Namen der Kuh, ihre Position in der Minecraft-Welt, ihre Gesundheit usw.

Es ist zwar wichtig, dass jede Kuh durch ein eigenes Objekt dargestellt wird, aber das reicht noch nicht als Grund dafür aus, Objekte überhaupt zu benutzen. Warum verwenden wir also Objekte? Dafür gibt es einen wichtigen Grund.

Wozu Objekte?

Stell dir vor, du wärst der Gott deines eigenen Universums. Du hast Äonen damit zugebracht, jedes Quark-Teilchen, jedes Atom, jedes Molekül usw. bis hin zu allen Planeten und Galaxien so anzuordnen, wie es dir gefällt. Jetzt dreht sich alles, und du bist für jedes einzelne subatomare Teilchen im ganzen Universum verantwortlich – und zwar für alle auf einmal. Selbst für ein gottähnliches Wesen ist es zu viel verlangt, sich um jedes Elektron oder jedes Quark-Teilchen oder auch nur um jedes Molekül zu kümmern, um das Universum in Gang zu halten (ganz abgesehen davon, dass es unglaublich langweilig wäre).

Stattdessen kümmerst du dich um die einzelnen Aufgaben jeweils in der zugehörigen Größenordnung. Wenn es ein Problem mit einem falsch platzierten Planeten gibt, dann verschiebst du den Planeten. Wenn du an einer Galaxie herumfeilen musst, dann feilst du an der Galaxie – nicht an den einzelnen Planeten und mit Sicherheit nicht an allen einzelnen Lebensformen auf allen Planeten sämtlicher Sonnensysteme in dieser Galaxie.

Das mag sich zwar ein bisschen übertrieben anhören, hat aber starke Ähnlichkeiten mit dem Programmieren. Auch als Programmierer bist du der sehr mächtige Schöpfer eines eigenen kleinen Universums. Auch wenn du dich dabei nicht unbedingt gottähnlich fühlst, musst du dich auch dort mit Problemen befassen, die auf unterschiedlichen Ebenen auftreten – auf niedriger Ebene, vergleichbar mit Atomen und Molekülen, und auf sehr hoher Ebene, also praktisch bei den Lebewesen, Bergen, Planeten und Galaxien. Und all das geschieht gleichzeitig, wie du in Abbildung 5.1 siehst. Beim Programmieren musst du oft in die »Atome« hineinzoomen und aus den Galaxien »herauszoomen«. Alles ist miteinander verbunden und muss insgesamt Sinn ergeben.

Aus diesem Grund schreiben wir Code mithilfe von Objekten. Dies bietet eine Möglichkeit, um Daten (in Variablen) und Verhalten (in Funktionen) zu gliedern. Wenn wir dann mit einer Kuh arbeiten müssen, können wir sie als Kuh behandeln und müssen uns nicht mit den einzelnen Millionen Atomen herumplagen, aus denen die Kuh besteht – oder das Biom, die Welt oder die Fackel usw.

Was noch wichtiger ist: Wir können den Code dadurch so schreiben, dass nur eine Kuh über die Funktionen und Daten verfügt, die eine Kuh braucht. Es gibt nichts Schlimmeres als eine bunte Mischung von Funktionen, die ziellos überall verstreut sind. Das könnte dazu führen, dass eine Fackel muht oder eine Kuh leuchtet. Womöglich hast du dann auch einen Riesenhaufen Funktionen, bei denen du dir nicht sicher bist, welche von ihnen mit welchen Daten zusammenarbeiten soll.

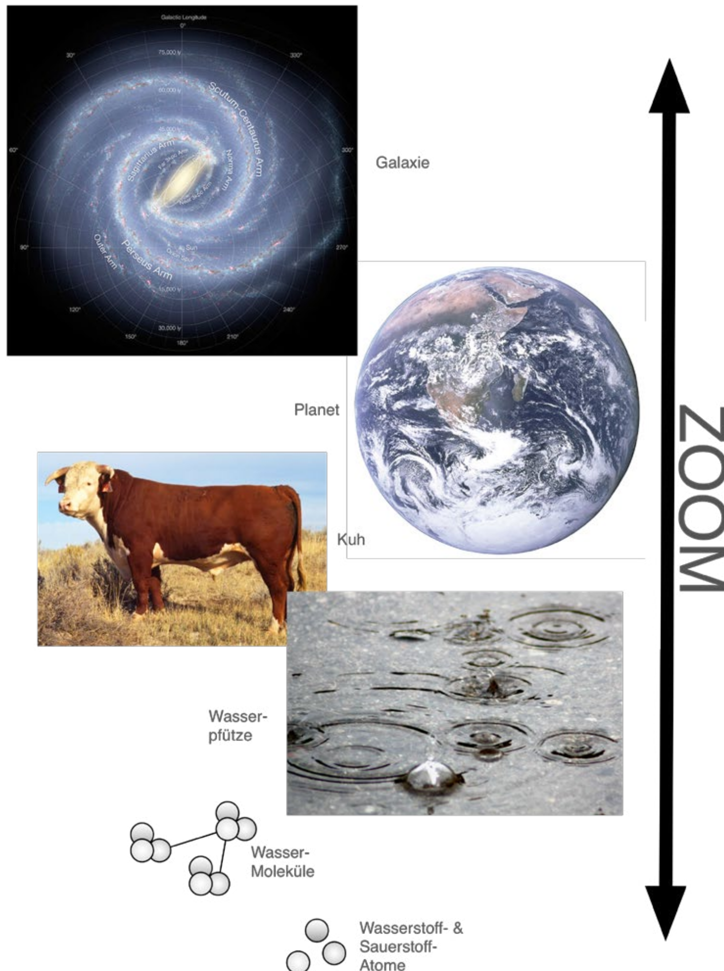


Abbildung 5.1: Ein- und Auszoomen zwischen Atomen und Galaxien

Beispielsweise hat ein `Cow`-Objekt (also ein Kuh-Objekt) in Minecraft eine Menge Funktionen, die du aufrufen kannst, darunter die folgenden:

- `teleportTo(Location` *Position*`)`
- `setFireTicks(int` *Ticks*`)`
- `setAge()`
- `getAge()`
- `swingArm()`

Das Objekt hält auch einige Angaben zum internen Status fest. Einige Variablen speichern die Position der Kuh, ihr Alter, die Information, ob sie brennt, usw.

Das sind alles praktische Dinge, die du mit einem `Cow`-Objekt anstellen kannst, allerdings sind sie für Spieler, Pfeile oder Bäume nicht unbedingt sinnvoll.

Alles Kuhhafte sollte auf `Cow`-Objekte beschränkt sein, alles, was mit Pfeilen zu tun hat, auf `Arrow`-Objekte usw. Anderenfalls kann es furchtbar schnell ein furchtbares Durcheinander geben, und du hast am Ende einen Haufen Code, den du selbst nicht mehr verstehst und mit dem du nicht mehr arbeiten kannst. Im Grunde genommen ist dies eine Frage der Hygiene, ebenso wie du Milch im Milchkarton im Kühlschrank aufbewahrst und nicht im Brotkasten. Normalerweise vermischt du ja nicht einmal Mais- und Kartoffelchips, um im Bild zu bleiben.

Dinge, die nicht zusammengehören, in getrennten Objekten vorzuhalten ist dabei die einfache Übung.

Der schwierigste Aspekt der Programmierung besteht darin, dass du dich manchmal gleichzeitig um Einzelheiten auf einer sehr niedrigen Ebene und einer sehr hohen Ebene kümmern musst. Wir sprechen hier auch von *Abstraktionsebenen*. Sehen wir den Tatsachen ins Auge: Wenn du Code für einen Spieler in Minecraft schreibst, handelt es sich dabei nicht um die reale Person, die sich am Spiel beteiligt. Es gibt diese reale Person zwar, und sie sitzt irgendwo, schwitzt, futtert Chips und hört laute Musik. Der Code, den du schreibst, ist eine abstrakte Darstellung dieser Spielerperson. Zu dieser Abstraktion gehören die Daten und die Verhalten, die für das Spiel gebraucht werden.

Wie in der Wirklichkeit bestehen auch diese Abstraktionen aus verschiedenen Teilen. Du kannst dich daher je nach Bedarf auf die Moleküle oder die Planeten konzentrieren – bzw. auf die Molekülobjekte und Planetenobjekte in der Software. Du kannst in die gewünschte Ebene hinein- oder aus ihr herauszoomen und jeweils mit den Teilen arbeiten, die du brauchst.

Sehen wir uns genauer an, was das in Minecraft und Java bedeutet.

Daten und Anweisungen zu Objekten kombinieren

Stell dir vor, du wolltest selbst ein Spiel wie Minecraft programmieren. In dem Spiel soll es mehrere Spieler geben, die jeweils einen eigenen Namen, ein eigenes Inventar, eigene Gesundheitspunkte usw. aufweisen. Die Struktur aller Spieler soll allerdings identisch sein.

Jedes Spielerobjekt verfügt also über die gleiche Menge von Variablen (Name, Gesundheit, Position usw.) und die gleichen Funktionen (um die Gesundheit des Spielers festzulegen, den Spieler an eine andere Position zu teleportieren usw.). Dazu sind Objekte praktisch. Du kannst in Java ein Objekt erstellen, das für alle Spieler im System steht. Anschließend kannst du Code für Spieler schreiben, der unabhängig davon, mit welchem konkreten Spieler du jeweils arbeitest, auf die gleiche Weise funktioniert. Das ist genau das, was die Programmierer von Minecraft getan haben.

In Java kannst du eine Menge Variablen definieren und einen Haufen Code schreiben, der diese Variablen nutzt, vergleichbar mit einem Rezept. Anschließend kannst du aus diesem Rezept ein Objekt erstellen und es verwenden. In Java heißen Rezepte dieser Art *Klassen*. Aus dem Klassenrezept erstellt Java ein funktionierendes Objekt. In der folgenden Abbildung siehst du einige Variablen und Funktionen für die Beispielklasse `Player` sowie einige Objekte, die daraus hergestellt werden können.

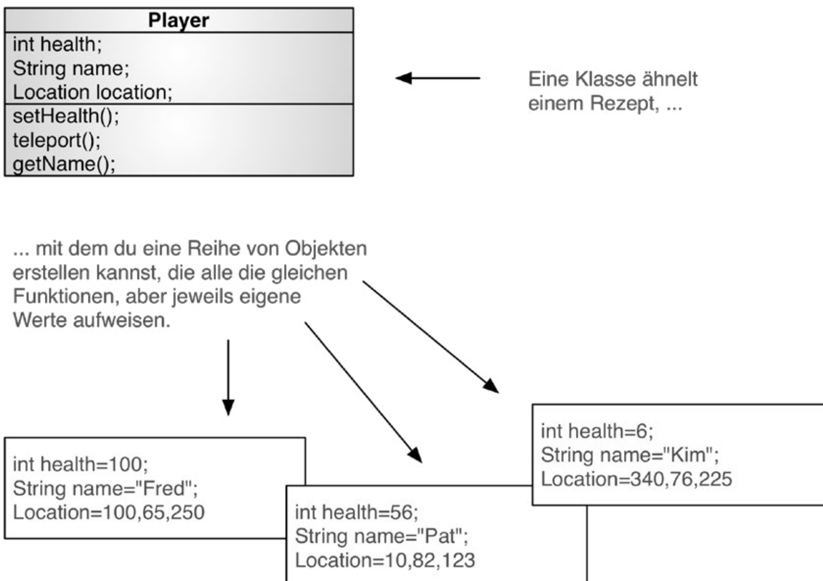


Abbildung 5.2: Eine Klasse ist ein Rezept, um Objekte zu erstellen.

Das ist so ähnlich wie das Bauen mit Lego-Steinen. Die Steine können alle identisch sein, aber wenn du der Bauanleitung in dem Kasten folgst, kannst du daraus ein Raumschiff konstruieren.

Beispielsweise hält Minecraft eine Menge interessanter Daten und Funktionen zu den einzelnen Spieler vor, die online sind. Das »Rezept« ist in der Klasse `net.canarymod.api.entity.living.humanoid.Player` definiert, die Objekte für einzelne Spieler erstellt. Bevor du `Player` in deinem Code verwendest, musst du am Anfang der Datei eine Importanweisung einfügen: in diesem Fall `import net.canarymod.api.entity.living.humanoid.Player`.

Wie können wir nun an eines dieser Spielerobjekte gelangen?

Da der Minecraft-Server jederzeit weiß, wer jeweils online ist, können wir ihn um eine Liste der `Player`-Objekte bitten. Wir können auch nach einem bestimmten Benutzernamen fragen, woraufhin uns der Server ein einzelnes `Player`-Objekt zurückgibt.

Innerhalb eines Plugins kannst du mit der Funktion `Canary.getServer()` auf den Server zugreifen. Dadurch erhältst du ein Objekt, das für den Server steht. Dieses Serverobjekt kannst du nun nach Spielern abfragen. Dazu verwendest du die Funktion `getPlayer` und rufst sie mit dem Namen des gewünschten Spielers auf.

Verwende die in diesem Abschnitt beschriebene Folge von Anweisungen. (Wir sehen uns zuerst die einzelnen Teile an. Am Ende steht ein vollständiges, lauffähiges Beispiel.)

Du beginnst mit den Importvorgängen für die Klassenrezepte, die du verwenden willst:

```
// Ganz am Anfang
import net.canarymod.Canary;
import net.canarymod.api.Server;
import net.canarymod.api.entity.living.humanoid.Player;
```

Weiter hinten, im Plugin-Code, kannst du dann Variablen dieser Typen verwenden (`Player` und `Server`):

```
Server myserver = Canary.getServer();
Player fred = myserver.getPlayer("fred1024");
```

Importe

Jedes Objektrezept in Java befindet sich in einem Paket, z. B. `java.util` oder `net.canarymod.Canary` usw. Dieses Paket musst du ganz am Anfang deiner Java-Quellcodedatei in einer Importanweisung deklarieren.

Wenn du das vergisst, erhältst du etwa folgende Fehlermeldung von `javac`:

```
cannot find symbol
```

Um den vollständigen Paketnamen zu ermitteln, musst du in der Dokumentation zu Canary oder Java nachschlagen. In unseren Beispielen verwenden wir viele gebräuchliche Pakete, wie `Player`, `Server`, `Location`, `Entity`, sowie verschiedene Java-Bibliotheken. Der Einfachheit halber sind all diese Pakete in Anhang 7, »Gebräuchliche Importe«, aufgeführt.

Wenn `fred1024` online ist, haben wir jetzt die Variable `fred` zur Verfügung, die für das zugehörige `Player`-Objekt steht. Ist Fred nicht online, so ist die Variable `fred` gleich `null`, das ist ein besonderer Wert, der so viel bedeutet wie »gibt's nicht«. Mit anderen Worten, `fred` ist auf kein Objekt gesetzt.[1]

Objekte bestehen aus einzelnen Teilen und haben Inhalte. Ein Integerwert wie 5 steht einfach nur für die Zahl 5, kann aber sonst nichts tun und außer der Zahl nichts speichern. Objekte aber haben Funktionen und Variablen, die du mithilfe der Punktschreibweise (.) abrufen kannst.

Wenn `fred` da ist, kannst du Daten über den Spieler abrufen (get) und festlegen (set), indem du hinter dem Punkt die Funktion oder Variable des Objekts angibst, auf die du zugreifen willst. Die folgenden Codefragmente zeigen, wie das geht:

```
// Schleicht Fred?
boolean sneaky = fred.isSneaking();

// Ist Fred hungrig?
int fredsHunger = fred.getHunger();

// Macht Fred hungrig
fred.setHunger(0);

// Wo ist Fred?
Location where = fred.getLocation();
```

Das `Player`-Objekt verfügt über die Funktion `isSneaking()`, die je nachdem, ob sich der Spieler im Schleichmodus befindet oder nicht, `true` oder `false` zurückgibt.

1 Wenn `fred` gleich `null` ist und du versuchst, eine Funktion von `fred` auszuführen, z. B. `fred.isSneaking()`, erhältst du eine Fehlermeldung, und dein Plugin stürzt ab.

Solche Angaben kannst du auch in `if`-Anweisungen nutzen:

```
if (fred.isSneaking()) {
   fred.setFireTicks(600); // Setzt ihn in Brand
}
```

Die Funktion `getHunger()` gibt einen `int`-Wert zurück, der besagt, wie hungrig der Spieler ist. Außerdem kannst du den Hunger eines Spielers festlegen, indem du die Funktion `setHunger()` verwendest. Im vorletzten Codeausschnitt hast du gesehen, wie du die aktuelle Position des Spielers in der Welt abrufst.

Wie du wahrscheinlich schon erraten hast, enthalten Objekte interne Variablen, mit denen ihre Funktionen arbeiten. In den Beispielen ist die Position des Spielers Fred in seinem `Player`-Objekt gespeichert. Wir können den Wert dieser Position abrufen und ihn festlegen, aber dabei behält Fred immer seine eigene interne Kopie des aktuellen Positionswerts.

Es gibt aber noch interessantere Möglichkeiten. Beispielsweise kannst du einen Befehl so ausführen, als wärst du Fred:

```
fred.executeCommand("tell mary179 I love you")
```

Mary glaubt jetzt, dass Fred ihr eine Liebeserklärung gemacht hat. Mal sehen, welche Streiche wir Fred spielen können, indem wir ein bisschen mit seiner Position herumspielen:

```
Location where = fred.getLocation();
```

Jetzt zeigt die Variable `where` auf das `Location`-Objekt, das Freds Position in der Welt darstellt.

Objekte erstellen

Eine Position kannst du auch von Grund auf neu erstellen:

```
double x, y, z;
x = 10;
y = 0;
z = 10;
Location whereNow = new Location(x, y, z);
```

Alternativ kannst du das auch in einem Schritt erledigen:

```
Location whereNow = new Location(10, 0, 10);
```

Das ist die Vorgehensweise, mit der du in Java ein neues Objekt anlegst: mit dem Schlüsselwort `new`.

Dabei erstellt Java für dich ein Objekt und führt dessen *Konstruktor* aus. Dabei handelt es sich um eine Funktion, die denselben Namen hat wie die Klasse (z. B. `public`

Location()). Der Konstruktor bietet dir die Gelegenheit, in dem Objekt alles einzurichten, was eingerichtet werden muss. Er gibt nichts zurück und ist auch nicht mit einem Rückgabetyp deklariert. Java gibt automatisch das neue Objekt zurück, nachdem du die Einrichtungsarbeiten erledigt hast.

Wie du deine eigenen Objektdefinitionen erstellst, erfährst du weiter hinten in diesem Buch. Zunächst aber nutzen wir das, was Minecraft uns bietet, sowie unser Plugin-Gerüst.

Da wir nun eine neue Position zur Verfügung haben, können wir Fred hinwegzaubern:

```
fred.teleportTo(whereNow);
```

Und plötzlich findet sich Fred – wo wieder? Im Grundgestein, wo er jämmerlich erstickt, da die y-Koordinate dieser Position 0 ist. Aua.

Plugin: PlayerStuff

Spielen wir noch ein bisschen mit Player herum.

Als Nächstes installieren wir das Plugin PlayerStuff und ändern einige der Eigenschaften dieses Objekts für Spieler in der Minecraft-Welt.

Das Player-Objekt enthält viele interessante Funktionen, um Informationen über einen Spieler abzurufen (get) und um die entsprechenden Werte festzulegen (set). Von diesen Funktionen sehen wir uns die folgenden an:

```
chat()
getWorld()
getDisplayName() (can set it too)
getExperience() (can set it too)
getHunger() (can set it too)
getHealth() (can set it too)
isSleeping()
getLocation()
```

Damit können wir eine Nachricht an einen Spieler senden, verschiedene Werte abrufen und noch einige Dinge mehr tun. Der folgende Code stellt ein vollständiges Plugin dar, das einige dieser Möglichkeiten vorführt. Es stellt den Befehl /whoami zur Verfügung. In anspruchsvolleren Plugins bietet diese Vorgehensweise eine großartige Gelegenheit, um Informationen über Objekte im Spiel anzuzeigen und damit Fehler an diesen Objekten zu beheben. Wenn der Code, den du geschrieben hast, nicht funktioniert, solltest du versuchen, einige Werte verschiedener Variablen auszugeben. Das ist eine gute Möglichkeit, um herauszufinden, was vor sich geht.

Den folgenden Code findest du im Plugin-Verzeichnis `Desktop/code/PlayerStuff`. Das vollständige Plugin verfügt auch über die notwendigen Dateien wie `Canary.inf` usw.

```
package playerstuff;

import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.api.world.position.Location;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.api.entity.living.humanoid.Player;
import com.pragprog.ahmine.ez.EZPlugin;

public class PlayerStuff extends EZPlugin {

   @Command(aliases = { "whoami" },
         description = "Displays information about the player.",
         permissions = { "" },
         toolTip = "/whoami")
   public void playerStuffCommand(MessageReceiver caller,
                                  String[] parameters) {
      if (caller instanceof Player) {
         Player me = (Player)caller;
         String msg = "Your display name is " + me.getDisplayName();
         me.chat(msg);
         me.getWorld().setRaining(true);
         me.getWorld().setRainTime(100); // 5 secs
         float exp = me.getExperience();
         int food = me.getHunger();
         float health = me.getHealth();
         Location loc = me.getLocation();

         me.chat("Your experience points are " + exp);
         me.chat("food is " + food);
         me.chat("health is " + health);
         me.chat("you are at " + printLoc(loc));
         me.chat("water falls from the sky ");
      }
   }
}
```

Listing 5.1: PlayerStuff/src/playerstuff/PlayerStuff.java

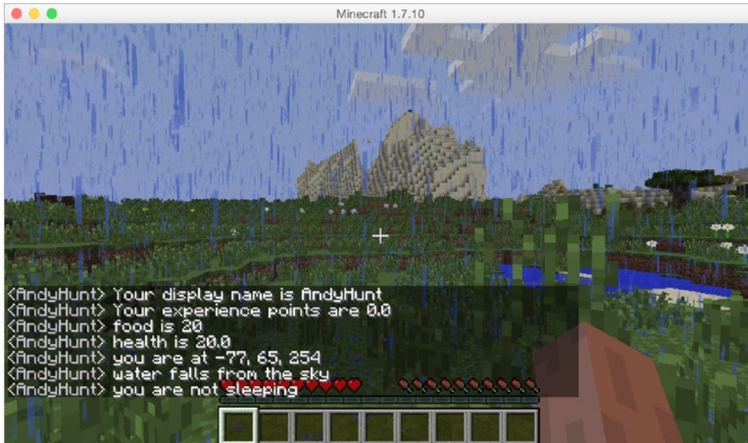
Installiere das Plugin jetzt:

```
$ cd Desktop
$ cd code/PlayerStuff
$ ./build.sh
```

Starte den Server neu und stell die Verbindung zum Minecraft-Client her.

Was geschieht nun, wenn du den Befehl `/whoami` im Minecraft-Client eingibst?

Ich erhalte dabei folgende Anzeige:



Gehen wir den Code Schritt für Schritt durch, um uns anzusehen, was im Einzelnen geschieht. Unser Ausgangspunkt ist das Spielerobjekt `me`. Darüber können wir den Namen des Spielers abrufen und ihm anschließend Nachrichten senden.

Nur zum Spaß lassen wir es auf den Spieler regnen (oder schneien), indem wir `Raining` auf `true` setzen und die Regendauer mit `setRainTime(100)` auf `5 secs` (Sekunden) einstellen.[2] Auf eine Sekunde kommen 20 Serverticks, weshalb 100 Ticks etwa 5 Sekunden entsprechen.

Als Nächstes rufen wir die Erfahrungspunkte für das nächste Level und die Nahrungspunkte ab und senden diese Werte als Nachrichten an den Spieler. Anschließend kannst du einige Zeit in der Welt herumspielen und dann erneut `/whoami` ausführen, um zu sehen, ob sich die Nahrungs- und Erfahrungspunkte inzwischen geändert haben.

Es ist wirklich so einfach wie in diesem Beispiel: `me` ist ein Objekt vom Typ `Player`, und darüber können wir verschiedene Spielerwerte abrufen und Befehle an `me` senden, um Dinge zu tun, die für Spieler sinnvoll sind.

Das ist der Sinn von Objekten!

2 Du kannst den Regen auch dadurch beenden, dass du `Raining` auf `false` setzt.

Probiere es selbst aus!

In dem Screenshot siehst du auch eine Zeile, die angibt, ob der Spieler schläft. Füge in `PlayerStuff` eine lokale Variable vom Typ `boolean` hinzu und setze sie je nachdem, ob der Spieler schläft oder nicht, auf `true` bzw. `false`. Erstelle dann eine entsprechende Meldung, um diese Angabe anzuzeigen.

Erstelle das Plugin mit `build.sh` und probiere es aus.

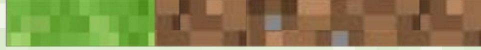
Wie ich diese Aufgabe gelöst habe, kannst du dir in `code/MyPlayerStuff/scrc/myplayerstuff/MyPlayerStuff.java` ansehen.

Weiter geht's!

In diesem Kapitel hast du gesehen, wie du Java-Objekte verwendest. Du hast gelernt, Java-Pakete und Klassen zu importieren, Objekte mit `new` anzulegen und die Eigenschaften von Objekten zu ändern, die sich auf das Spiel auswirken. All dies brauchen wir noch in den folgenden Kapiteln.

Im nächsten Kapitel sehen wir uns genauer an, was geschieht, wie die Plugins mit Minecraft verzahnt sind, wie du Befehle hinzufügst und Dinge in der Minecraft-Welt findest.

Der Inhalt deines Werkzeugkastens



Dein Werkzeugkasten ist jetzt zu 28 % gefüllt. Du kannst nun Folgendes tun:

- die Kommandozeilenshell nutzen
- Programme mit Java und `javac` erstellen
- einen Minecraft-Server ausführen
- ein Plugin bereitstellen
- Verbindung zu einem lokalen Server aufnehmen
- Java-Variablen für Zahlen und Strings verwenden
- Java-Funktionen verwenden
- `if`-, `for`- und `while`-Anweisungen verwenden
- Java-Objekte verwenden
- Java-Pakete importieren
- Objekte mit `new` erstellen