

---

# CAMP MINECRAFT MODS

---

– Tag 4 –

## Kapitel 1 – Einführung

Heute schauen wir uns mal ein neues Programm, und alle damit einher kommenden neuen Möglichkeiten an. Blockbench. Dieses großartige Programm ermöglicht uns, mit relativ wenig Aufwand Blöcke und Items zu Modellieren, vor allem besondere 3D Ressourcen zu erstellen – aber vor allem ermöglicht uns das Programm eigene Kreaturen Modelle zu erzeugen.

### Kapitel 1.1 – Grundlagen

Schauen wir uns mal gemeinsam die Steuerung und sonstige Grundlagen für Blockbench an.

Im 3-dimensionalen Raum (3D ist die Kurzschreibweise dafür), haben wir 3 Achsen, auf denen wir uns bewegen können – die X-Achse, Y-Achse und die Z-Achse (Manchmal auch anders benannt, aber standardmäßig werden sie so benannt). Übersetzt bedeutet das für uns also:

- X-Achse: Bewegung nach rechts und links
- Y-Achse: Bewegung nach oben und unten
- Z-Achse: Bewegung nach vorne und hinten

In Blockbench werden die drei Ebenen durch verschieden farbige Pfeile angezeigt. Der rote Pfeil repräsentiert die X-Achse, der grüne die Y-Achse und der blaue die Z-Achse.

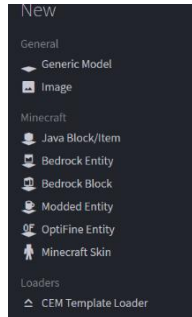
Um eure Sicht zu bewegen, könnt ihr die mittlere Maustaste gedrückt halten (also das Mausrad), um euch um den Fixpunkt zu rotieren, wenn ihr dabei noch die *shift* Taste gedrückt haltet, könnt ihr den Blick auf der Y-Achse verschieben. Mit dem Mausrad könnt ihr einfach hinein oder herauszoomen. Pfeile könnt ihr per linker Maustaste gedrückt halten ziehen und verändern.

Macht euch nun zunächst einmal Gedanken, was ihr überhaupt erstellen wollt. Soll eure eigene Kreatur ein friedliches liebes Tier sein, vielleicht ein Begleiter für euch, oder eine neue Kreatur, die die Weiten der Welt erkundet und glücklich ist? – Als Vorlage im Spiel haben wir hier das Stachelschwein (engl.: porcupine), oder soll die Kreatur, die ihr erschaffen wollt, lieber ein furchtsamer, aggressiver Räuber sein, eine Kreatur, die alle um sie herum in Angst und Schrecken versetzt beim Anblick?

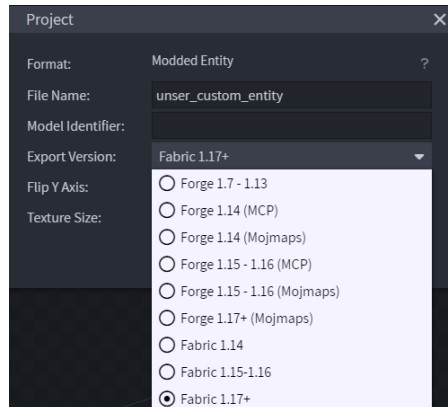
Je nachdem, wofür ihr euch entscheidet, könnt ihr euch gleich einmal Gedanken machen, ob ihr irgendein bereits vorhandenes Wesen als Vorlage nehmen wollt, oder lieber etwas ganz Neues, einfaches erschaffen wollt.

## Kapitel 2 – eigene Modelle

Um jetzt ein eigenes Modell zu erstellen, gehen wir auf der Fläche an der Seite, auf *Modded Entity* → *create new Model*, und geben dann in dem Feld, das sich öffnet, einen Namen ein.



Am besten benennt ihr es der Einfachheit halber *NameModel*. Und ersetzt dann den Namen mit irgendeinem großartigen Namen für eine Kreatur. Außerdem müsst ihr hier dann darauf achten, dass ihr die richtige Version zum Exportieren auswählt.



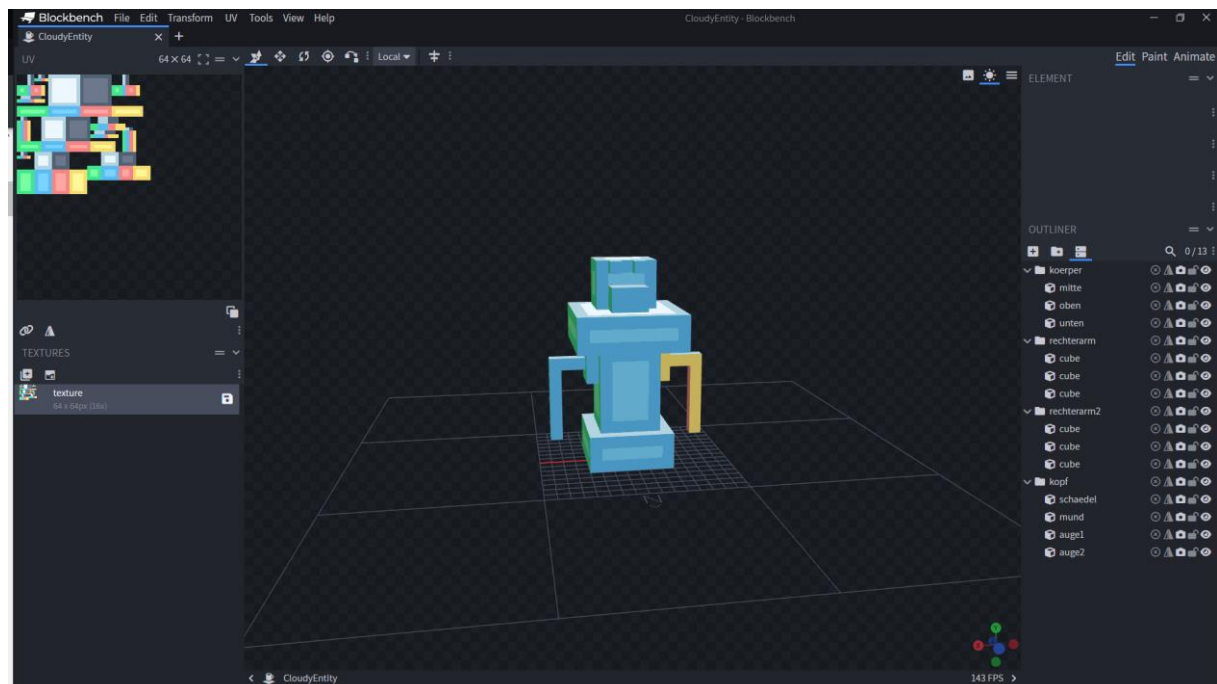
Hier müssen wir bitte einmal bei der *Export Version* darauf achten, dass wir *Fabric 1.17+* ausgewählt haben, nicht mit (Mojmaps) dahinter. Vergleicht also einmal euer Fenster mit dem Bild hier neben, das sollte exakt so aussehen.

Die restlichen Einstellungen sollten so weit passen, also danach einfach auf *confirm* klicken, und ihr solltet eine leere Modellfläche vorfinden.

Um nun ein neues Modell zu erstellen, brauchen wir – so wie Minecraft ja auch ausschließlich aus vielen kleinen Quadern besteht – einen neuen Quader. Alle Teile unseres Modells bestehen aus Quadern, welche ihr in Größe und Dimensionen anpassen könnt.

Schauen wir uns gemeinsam erst mal ein einfaches Modell an, welches wir gemeinsam Schritt für Schritt erstellen wollen.

Das Modell soll am Ende so aussehen:



## Kapitel 2.1 – Unser Slenderpeter

Dieses Modell besteht aus 13 Quadern, in 4 Gruppen, und das wollen wir jetzt einmal gemeinsam erstellen.

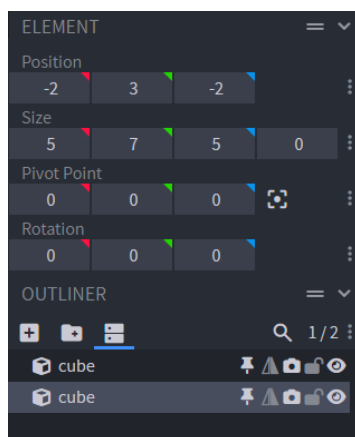
Zuerst benötigen wir einen Körper für das Monster – wir erstellen hier ein furchteinflößendes Monster, falls das nicht zu erkennen war...

Der Körper besteht aus 3 Quadern – fangen wir also an! Um einen neuen Quader zu erstellen, drücken wir einmal *strg + n*, oder drücken rechts in dem Feld *OUTLINER* auf die ganz linke Funktion *Add Cube*.

Diesen wollen wir jetzt bearbeiten, damit der auch die richtigen Dimensionen und Position hat. Dafür können wir einfach die Werte des Quaders oben rechts im Feld anpassen – oder die farbigen Pfeile ziehen.

Als Position wollen wir die Werte -3, 0, -3 haben – das sind die X-, Y-, und Z-Achsen Koordinaten des Quaders. Die Größe soll 7, 3, 7 betragen (wieder die X-, Y-, Z-Koordinaten des Gitters).

Nun haben wir eine schöne Basis, die „Beine“ des Monsters. Darauf kommen jetzt erst mal ein Rumpf und dann noch Schultern. Also brauchen wir einen neuen Quader mit den Maßen im Bild daneben:



Ihr könnt auch zur einfachen Erkennung direkt die Quader benennen, dafür einfach doppelt auf den Namen *cube* klicken in der Liste im *OUTLINER*, und den gewünschten Namen eingeben – also z.B. *beine* (bitte kleingeschrieben).

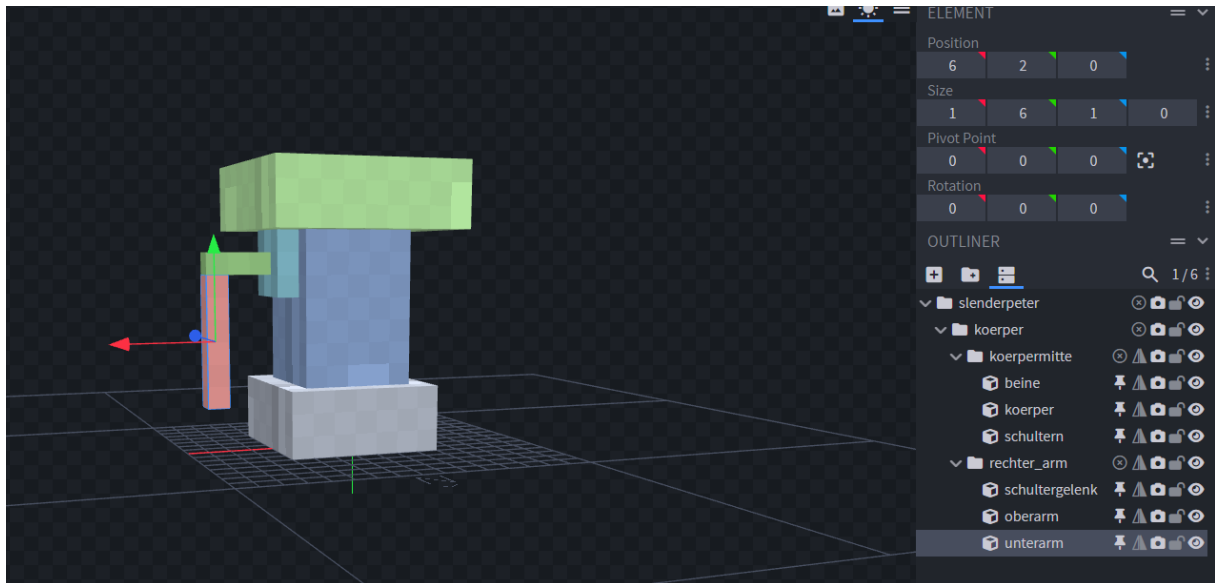
Den anderen Quader könnt ihr dann auch benennen, z.B. in *koerpermitte* (nur wichtig: keine Sonderzeichen – also kein ö, ä, ü).

Den letzten Quader erzeugen wir, und verändern ihn bitte auf die Werte -4, 10, -4 als Position – und 9, 3, 9 für die Größe. Diesen könnt ihr dann *schultern* nennen.

Als letztes wollen wir diese Körperteile alle in eine Gruppe, zur besseren Ordnung schmeißen. Dafür klicken wir auf das Zweite Icon – *Add Group* – oder aber drücken *strg + G*. Diese Gruppe nennen wir dann einmal genauso wie die Kreatur selbst, also z.B. *slenderpeter*, dann erstellen wir noch eine neue Gruppe und nennen diese Gruppe dann *koerper*, und eine die wir *koerpermitte* nennen.

Um jetzt die einzelnen Quader in die Gruppe zu bekommen, drückt ihr auf den obersten, haltet *shift* gedrückt und wählt dann den letzten in der Liste aus, und zieht alle drei Quader in die Gruppe *koerpermitte*. Dann zieht ihr die Gruppe *koerpermitte* – falls diese nicht bereits in der Gruppe *koerper* ist – in diese hinein, und diese dann in die Gruppe *slenderpeter*.

Nun wollen wir dem Slenderpeter noch Arme geben, also klicken wir auf die Gruppe *koerper*, und erstellen hier eine neue Gruppe: *rechter\_arm*, und in der Gruppe einen neuen Quader mit der Größe: (1, 3, 3) und der Position (3, 7, -1). Diesen nennen wir dann *schultergelenk*, erstellen einen zweiten an der Position (4, 8, 0) für den *oberarm* mit der Größe (3, 1, 1), und noch einen letzten an der Position (6, 2, 0) und Größe (1, 6, 1) für den *unterarm*. Nun sollte das Ganze in etwa so aussehen:



Jetzt braucht unser *Slenderpeter* nur noch den zweiten Arm, und einen Kopf. Dafür können wir einfach die Gruppe des *rechter\_arm* auswählen, und duplizieren diese per *strg+D* (oder rechtsklick und *Duplicate*), und geben bei *Rotation* einfach 180° bei Y an. Dann sehen wir, dass das Modell des linken Arms aber nicht ganz am Körper sitzt, das können wir aber auch einfach anpassen, indem wir einen der 3 Quader auswählen, dann die Gruppe auswählen, und dann die Position auf der X-Achse um 1 nach links setzen, und auf der Z-Achse eins nach hinten.

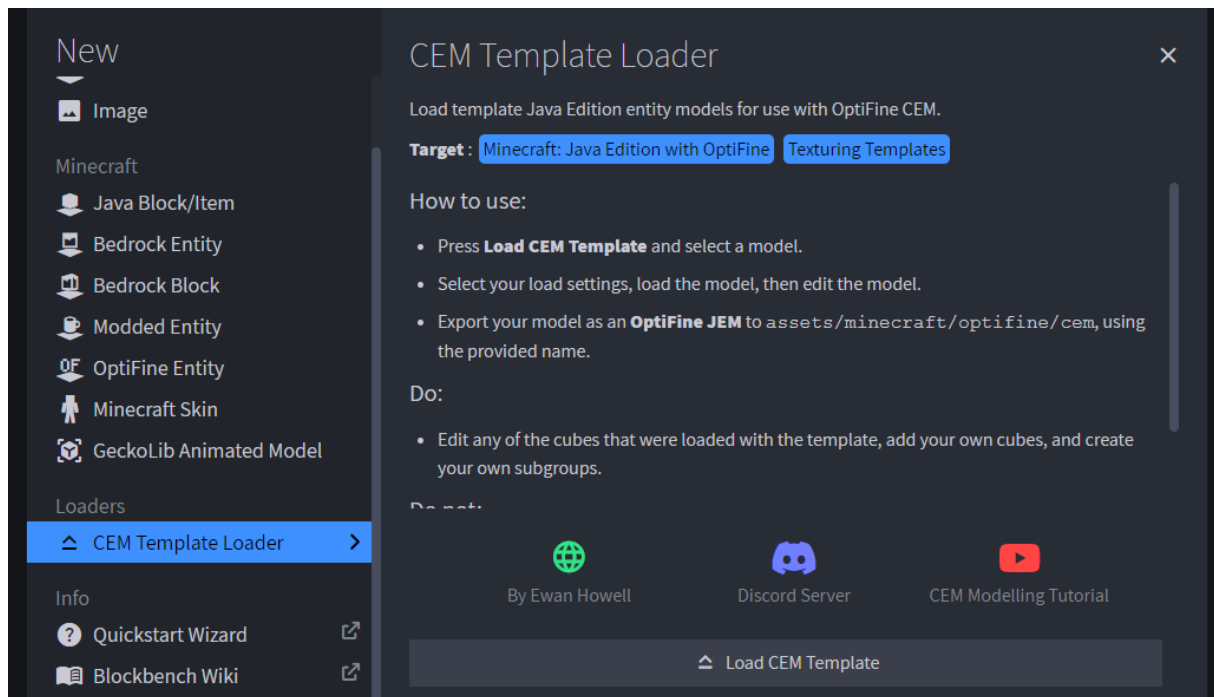
Nun haben wir unser Modell so weit fertig, was fehlt, ist ein Kopf, oder eventuell noch andere Merkmale unseres *Slenderpeters*. Nehmt euch jetzt einmal etwas Zeit, euch nochmal mit den einzelnen Schritten des Erstellens des Modells vertraut zu machen, und nehmt euch eine der Drei folgenden Aufgaben vor:

1. Fügt an das Modell vom *Slenderpeter* noch weitere Merkmale hinzu, wie einen Kopf, vielleicht Füße oder Hände, Klauen, eine Waffe oder sonstige Details am Modell
2. Erstellt ein komplett neues eigenes Modell
3. Verändert ein bereits bestehendes Modell aus *Minecraft*, um eure eigene Version zu erstellen

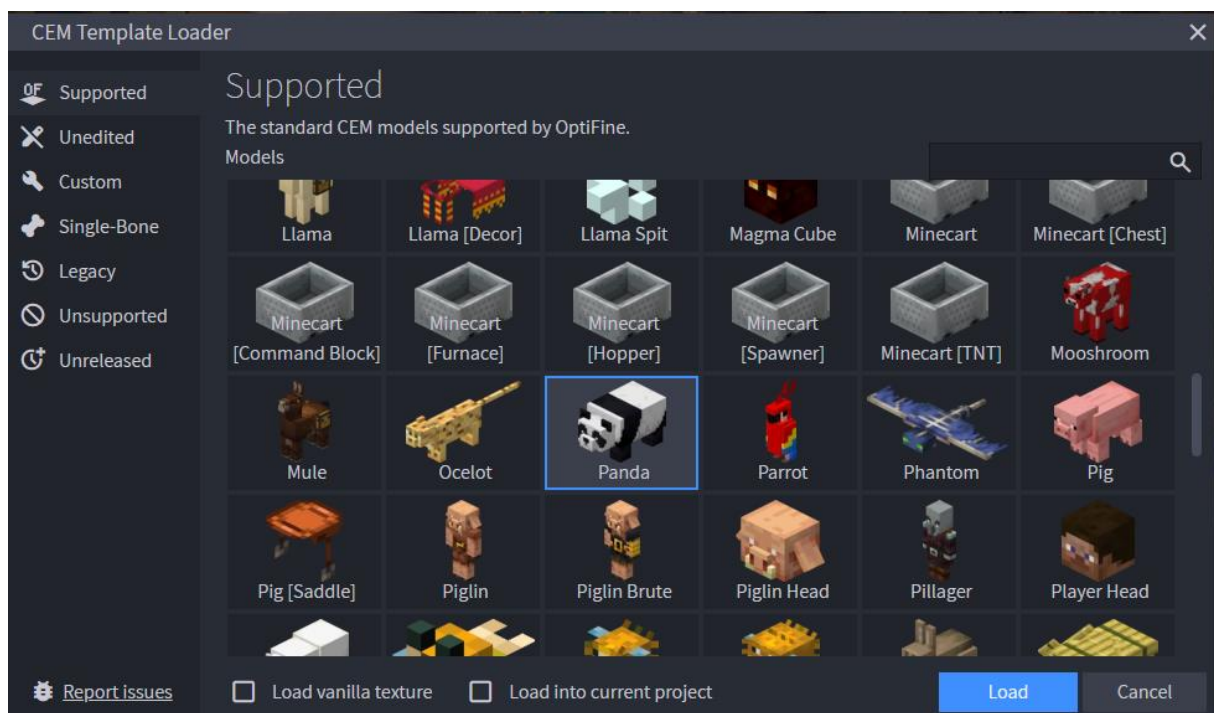
Für Aufgabe 1 und 2 wisst ihr eigentlich alles, was ihr brauchen solltet, ihr könnt euch gerne einfach an dieser Doku entlanghangeln, sollte ein Schritt unklar sein. Achtet auf eine ordentliche Ordnerstruktur, um die einzelnen Teile des Körpers besser zu finden und zu gruppieren. Fangt also gerne mit einem Ordner an, mit dem Namen eures Modells, in welchem ihr dann die Unterordner erstellt, und darin erst die Quader, die ihr braucht, das spart euch etwas Zeit.

## Kapitel 2.2 – ein Modell verändern

Um jetzt ein bereits vorhandenes Modell zu verändern, wollen wir ein Plugin benutzen – also eine Erweiterung für das Programm *Blockbench*. Dieses ist bei euch allen bereits installiert, alles, was ihr tun müsst, ist beim Erstellen eines neuen Projektes in *Blockbench* – statt *Modded Entity* auszuwählen – wählen wir jetzt den *CEM Template Loader* aus:



Hier gehen wir dann unten auf die Schaltfläche *Load CEM Template* ^, und sehen dann eine Bibliothek an vorhandenen Modellen. Dort könnt ihr euch unter den *SUPPORTED* Modells eins aussuchen, welches ihr anpassen wollt – also hier z.B. den Panda:



Dann geht ihr einfach auf *Load* und verändert das Modell, so wie ihr wollt. Wichtig hierbei:

Alle Ordner und Teile, die existieren, müssen bitte bestehen bleiben. Ihr dürft die Cubes, die bereits existieren verändern, **NICHT** aber die Ordner, in denen Sie sind. Alles klar?

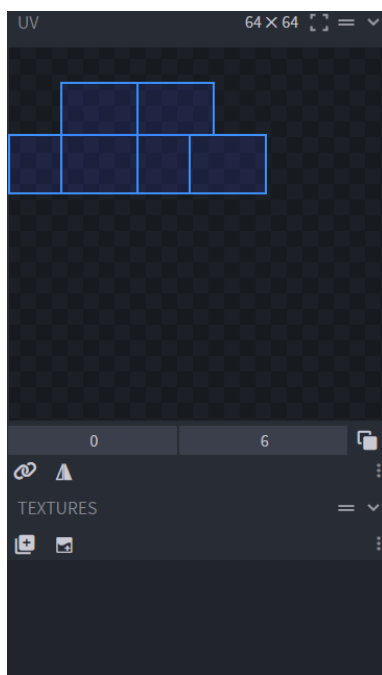
## ANPASSUNG EXPORT VERSION

Bei den Modellen aus dem Template Loader müssen wir einmal unter *File -> Convert Project* unser Projekt in ein *ModdedEntity* Format ändern, und unter *File -> Project* einmal die Exportversion auf Fabric 1.17+ ändern (nicht Mojmaps).

## Kapitel 3 – Texturen

Unser Modell von vorhin – der Slenderpeter – sah ja ganz schön seltsam aus. Und damit meine ich nicht die Körperform, sondern die fehlenden Texturen, immerhin war unser guter Slenderpeter ganz schön farblos, und nicht wirklich einheitlich von der Textur her.

Das können – und wollen wir natürlich ändern. Dafür haben wir auf der linken Seite des Overlays die Funktion *Textures – create Textures*:



Das UV zeigt euch die 3D-Darstellung eures Modells in 2D an, also die Textur, die das Programm dann interpretiert, um daraus ein 3D-Modell zu machen. Die einzelnen Quader, die ihr auswählt in dem Modell, oder rechts im Outliner, werden auch immer dann Blau markiert im UV.

Unter *TEXTURES* seht ihr dann die Möglichkeit eine neue Textur zu erstellen (rechte Funktion, Ebene mit einem Plus unten drin und einer geschwungenen Linie) – wenn ihr daraufklickt, wird eine generische Textur über das Modell gelegt, die ihr dann selbst einmal anpassen und verändern könnt. (Lasst die Einstellungen einfach alle so, wie sie sind, verändert nur den Namen, kleingeschrieben, zu eurer Kreatur)

Dafür müsst ihr auf der rechten Seite, oberhalb vom Outliner, einfach statt *Edit* – auf den Reiter *Paint* gehen, und dann ähnlich wie in *Paint.net* könnt ihr eure Textur mit den Werkzeugen oben am Bildschirmrand dann erstellen. Ihr seht die Veränderung dann auch direkt im UV und der Textur Datei an der Seite.

Sobald wir die Textur fertig haben, und zufrieden sind, geht's jetzt wieder zum Code zurück. Zumindest fast – zuerst speichern wir die Textur einmal im Texturen Ordner ab. Also wählen wir die Textur aus, drücken rechtsklick darauf und wählen *Save as* aus, und speichern unter: *Desktop -> CampMinecraftMods\_V1 -> src -> main -> resources -> assets -> forscherefreundemod -> textures -> entity*. Dann gehen wir noch auf *Datei (evt. Auf Englisch File) -> Export -> Export JavaEntity* und speichern die Datei dann unter:

*Src -> main -> java -> net -> forscherefreunde -> mod -> entity -> client* ab.

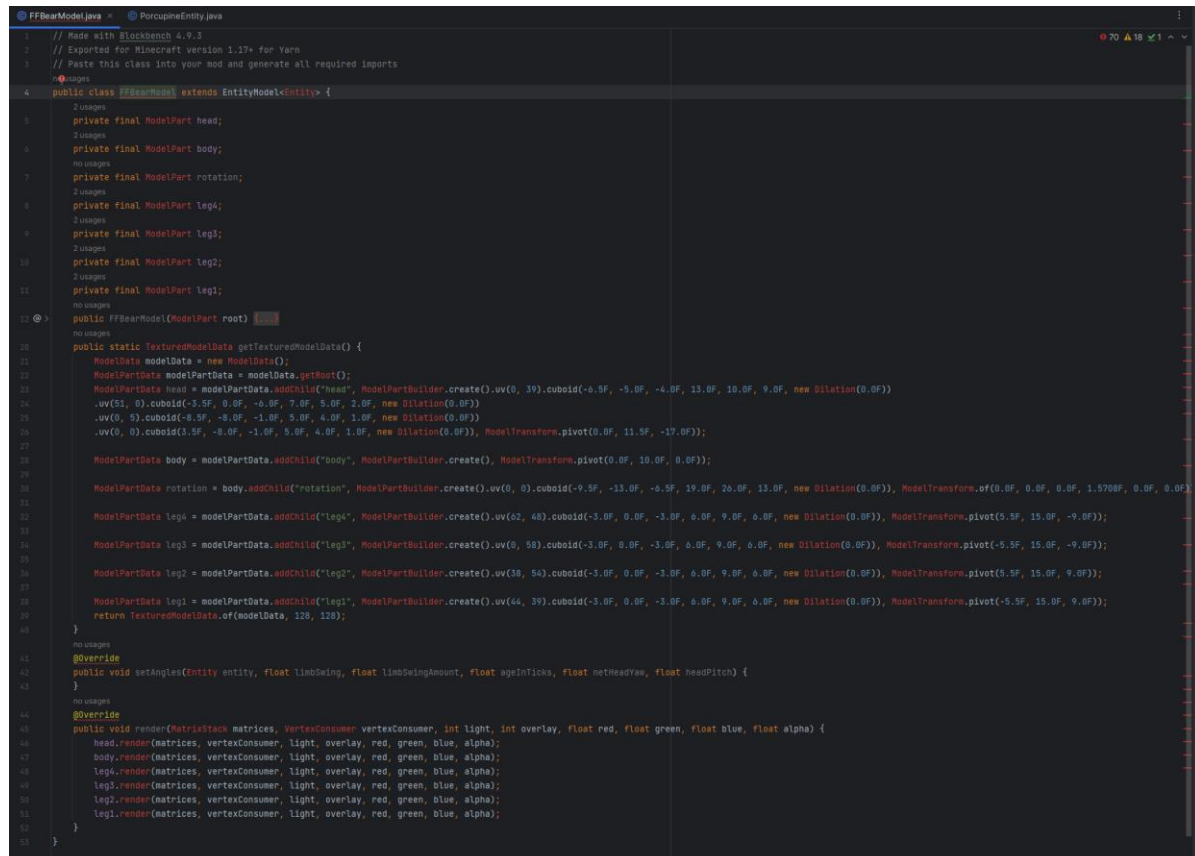
Als Namen gebt ihr eurer Kreatur einfach euren Namen + Model (also: *nameModel*). Dort sollten schon 7 Dateien sein, das *PorcupineModel.java*, *CloudyModel.java* und die *FFBearModel.java* Dateien, sowie der *PorcuPineRenderer.java*, *CloudyRenderer.java* und der *FFBearRenderer.java*, und zuletzt noch die *ModModelLayers.java* Klasse. Dann begeben wir uns jetzt in IntelliJ, um die Kreaturen auch zum Leben zu erwecken!



## Kapitel 4 – Die ModelKlasse

Um nun eine eigene Entity in Minecraft zu bekommen, haben wir wieder den Code für euch so einfach wie möglich versucht zu bekommen. Wir werden aber nicht umher kommen, ein paar Klassen selbst zu erstellen, und eine Menge Code anzupassen. Fangen wir also mit der eben Exportierten Model Klasse an.

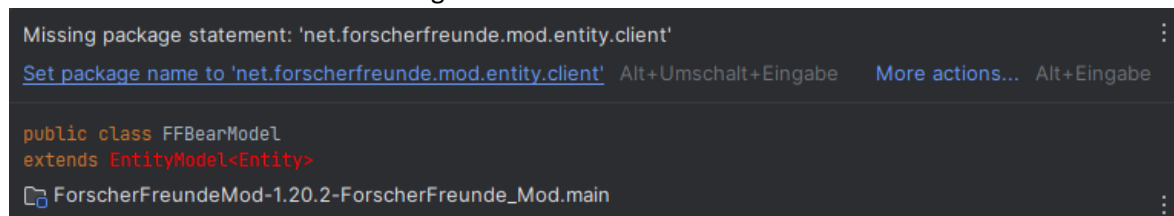
Wenn ihr in IntelliJ den Ordner *entity* öffnet, solltet ihr eure eigene Kreatur Klasse dort finden, mit dem Namen, den ihr der Datei gegeben habt. In unserem Beispiel hier, ist das die *FFBearModel*:



```
1 // Made with Blockbench 4.9.3
2 // Exported for Minecraft version 1.17+ for Yarn
3 // Paste this class into your mod and generate all required imports
4 @SuppressWarnings("unused")
5 public class FFBearModel extends EntityModel<Entity> {
6     // Usages
7     private final ModelPart head;
8     // Usages
9     private final ModelPart body;
10    // Usages
11    private final ModelPart rotation;
12    // Usages
13    private final ModelPart leg4;
14    // Usages
15    private final ModelPart leg3;
16    // Usages
17    private final ModelPart leg2;
18    // Usages
19    private final ModelPart leg1;
20    // Usages
21    public FFBearModel(ModelPart root) {
22        // Usages
23        public static TextureModelData getTextureModelData() {
24            ModelData modelData = new ModelData();
25            ModelPartData modelPartData = modelData.getRoot();
26            ModelPartData head = modelPartData.addChild("head", ModelPartBuilder.create().uv(0, 39).cuboid(-0.5f, -0.5f, -4.0f, 13.0f, 10.0f, 9.0f, new Dilation(0.0f))
27                .uv(51, 0).cuboid(-3.5f, 0.0f, -0.0f, 7.0f, 5.0f, 2.0f, new Dilation(0.0f))
28                .uv(0, 5).cuboid(-0.5f, -0.0f, -1.0f, 5.0f, 4.0f, 1.0f, new Dilation(0.0f))
29                .uv(0, 0).cuboid(0.5f, -0.0f, -1.0f, 5.0f, 4.0f, 1.0f, new Dilation(0.0f)), ModelTransform.pivot(0.0f, 11.5f, -17.0f));
30            ModelPartData body = modelPartData.addChild("body", ModelPartBuilder.create(), ModelTransform.pivot(0.0f, 10.0f, 0.0f));
31            ModelPartData rotation = body.addChild("rotation", ModelPartBuilder.create().uv(0, 0).cuboid(-9.5f, -13.0f, -0.5f, 19.0f, 20.0f, 13.0f, new Dilation(0.0f)), ModelTransform.of(0.0f, 0.0f, 0.0f, 1.5708f, 0.0f, 0.0f));
32            ModelPartData leg4 = modelPartData.addChild("leg4", ModelPartBuilder.create().uv(0, 40).cuboid(-3.0f, 0.0f, -3.0f, 0.0f, 9.0f, 0.0f, new Dilation(0.0f)), ModelTransform.pivot(0.5f, 15.0f, -9.0f));
33            ModelPartData leg3 = modelPartData.addChild("leg3", ModelPartBuilder.create().uv(0, 50).cuboid(-3.0f, 0.0f, -3.0f, 0.0f, 9.0f, 0.0f, new Dilation(0.0f)), ModelTransform.pivot(-5.5f, 15.0f, -9.0f));
34            ModelPartData leg2 = modelPartData.addChild("leg2", ModelPartBuilder.create().uv(20, 54).cuboid(-3.0f, 0.0f, -3.0f, 0.0f, 9.0f, 0.0f, new Dilation(0.0f)), ModelTransform.pivot(0.5f, 15.0f, 9.0f));
35            ModelPartData leg1 = modelPartData.addChild("leg1", ModelPartBuilder.create().uv(44, 39).cuboid(-3.0f, 0.0f, -3.0f, 0.0f, 9.0f, 0.0f, new Dilation(0.0f)), ModelTransform.pivot(-5.5f, 15.0f, 9.0f));
36            return TextureModelData.of(modelData, 120, 120);
37        }
38        // Usages
39        @Override
40        public void setAngles(Entity entity, float limbSwing, float limbSwingAmount, float ageInTicks, float netHeadYaw, float headPitch) {
41            // Usages
42        }
43        // Usages
44        @Override
45        public void render(MatrixStack matrices, VertexConsumer vertexConsumer, int light, int overlay, float red, float green, float blue, float alpha) {
46            head.render(matrices, vertexConsumer, light, overlay, red, green, blue, alpha);
47            body.render(matrices, vertexConsumer, light, overlay, red, green, blue, alpha);
48            leg4.render(matrices, vertexConsumer, light, overlay, red, green, blue, alpha);
49            leg3.render(matrices, vertexConsumer, light, overlay, red, green, blue, alpha);
50            leg2.render(matrices, vertexConsumer, light, overlay, red, green, blue, alpha);
51            leg1.render(matrices, vertexConsumer, light, overlay, red, green, blue, alpha);
52        }
53    }
```

Wie man erkennen kann, haben wir eine MENGE Fehler, und alles ist rot irgendwie... das ist aber ganz normal – also wollen wir mal etwas aufräumen.

Zuerst schwebt ihr mit der Maustaste über dem Namen der Klasse – also hier dem grauen Text *FFBearModel* in Zeile 4. Dann geht ihr in dem sich öffnenden Fenster auf die Funktion:

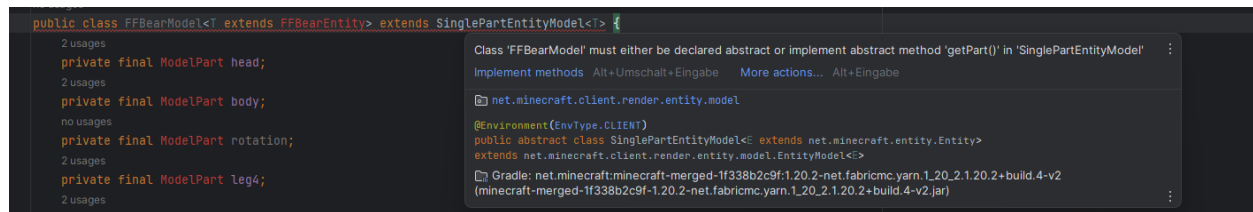


```
Missing package statement: 'net.forscherfreunde.mod.entity.client'
Set package name to 'net.forscherfreunde.mod.entity.client' Alt+Umschalt+Eingabe More actions... Alt+Eingabe

public class FFBearModel
extends EntityModel<Entity>
ForscherFreundeMod-1.20.2-ForscherFreunde_Mod.main
```

Dann schauen wir uns den Code mal etwas genauer an. Sobald ihr das Package gesetzt habt, sollte IntelliJ direkt zum nächsten Fehler im Code übergehen – und euch dort Vorschlägen, (in diesem Fall das rote *Entity*) diese Klasse doch bitte zu Importieren. Das wollen wir auch machen – zumindest so etwas in der Art. Zunächst wollen wir nämlich das *EntityModel<Entity>* hinter dem **extends** durch eine andere Klasse ersetzen: *SinglePartEntityModel<T>*. Dafür müssen wir aber zuvor noch die Klasse

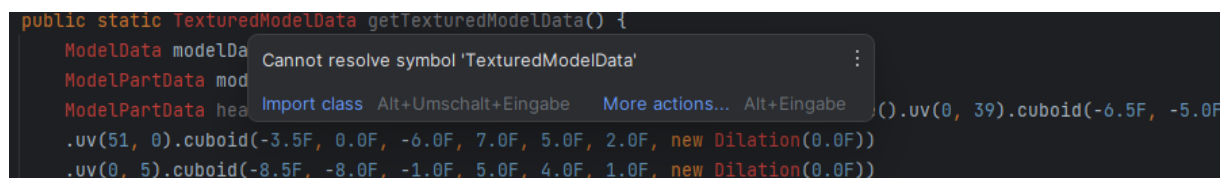
*FFBearModel* etwas erweitern, indem wir hinter die Klasse noch einen Typen Parameter schreiben. Eure Zeile sollte danach so aussehen:



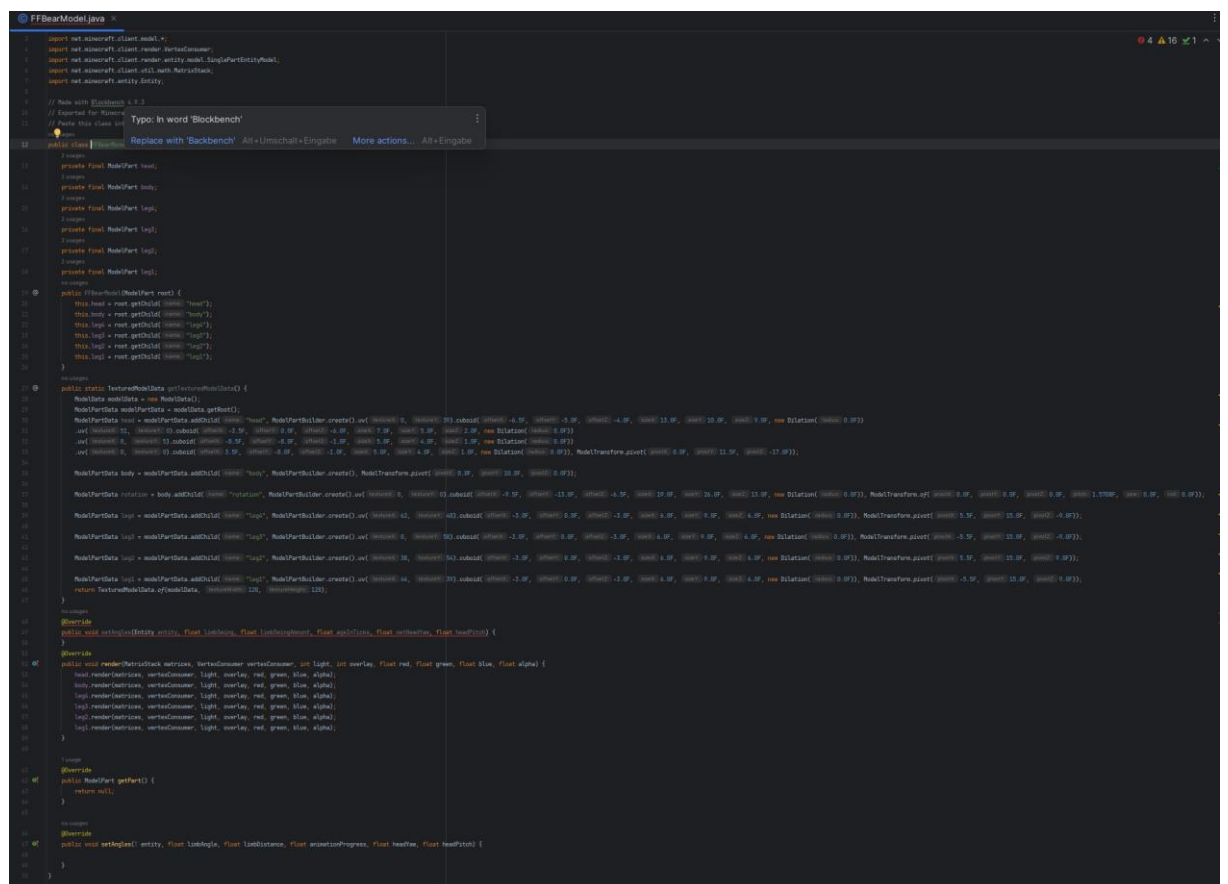
Also: *FFBearModel<T extends FFBearEntity> extends SinglePartEntityModel<T> { ... }*.

Nun gehen wir in diesem sich öffnenden Fenster auf *Implement methods* und drücken danach in dem Fenster auf *ok*.

Nun gehen wir den Code weiter durch, und bei allen noch roten Textfeldern, schweben wir mit der Maus drüber, bis sich ein Fenster öffnet, und Importieren die nötige Klasse:



Wenn wir damit durch sind, sollte die Klasse so aussehen:



Nun wollen wir den Code noch etwas aufräumen – denn wir haben eine Funktion, die noch rot unterstrichen sein sollte, diese können wir einfach entfernen (Die Funktion *setAngles(...)* ist *doppelt*, die *rot unterstrichene brauchen wir nicht mehr*).



Nun wollen wir der Funktion *getPart()* noch einen Sinn geben -> also schreiben wir dort:

```
@Override
public ModelPart getPart() {
    return this.body;
}
```

Bzw. in eurem Fall den Namen eures Models, welches auch der Name des Grundordners in Blockbench sein sollte bei euch.

Nun brauchen wir noch eine Entity Klasse für unsere Kreatur, und eine Renderer Klasse, damit das Model auch richtig dargestellt werden kann.

## Kapitel 4.1 – Die Entity-Klasse

Für die Entity-Klasse schauen wir uns mal ein paar Vorlagen an – da haben wir nämlich auch etwas für euch vorbereitet, um euch die Arbeit zu vereinfachen. Geht deshalb doch mal in den Ordner *vorlagen*, und schaut euch die Klasse *CustomEntityVorlage* einmal an. Die Klasse dient als Vorlage für euch, die ihr einfach so kopieren könnt, um eure eigene Entity Klasse (Friedliche Kreatur) zu erstellen. Dafür wollen wir die wichtigen Teile der Klasse einmal genauer besprechen.

```
3  > import ...
15
16  /*
17   * BITTE GENAU LESEN!! +
18   * Diese Klasse ist eine Beispielklasse zum Kopieren und Anpassen - alle genauen Schritte sind in der Doku [Tag 4] in
19   * [Kapitel 4.1] zu finden. Bitte achtet genau auf die Doku, da ihr sonst eventuell Fehler einbaut, die schwer zu finden sind.
20   * Bei Rot unterstrichenen Fehlern wie hier - Alle - bitte euren Teamer fragen, dass er sich den Fehler im Code
21   * einmal anschauen kann. Diese Fehler führen dazu, dass das Programm gar nicht erst ausgeführt werden kann. Ihr bekommt
22   * dann auch direkt einen Fehler in der Konsole ausgespuckt. Vermutlich liegt es dann an einem Rechtschreibfehler oder
23   * Syntaxfehler. Meistens gibt euch IntelliJ auch schon Vorschläge, jedoch bitte nicht einfach die Vorschläge
24   * ausführen, da das manchmal auch zu noch mehr Fehlern führen kann.
25   *
26   * Viel Spaß mit der Klasse.
27   * @Author
28   * Matthias Mendler
29   */
30
31
32
33  ▲ ForscherViking *
34  public class CustomEntityVorlage extends ModEntity{
35      no usages  ▲ ForscherViking
36      protected CustomEntityVorlage(EntityType<? extends AnimalEntity> entityType, World world) {
37          super(entityType, world);
38      }
39
40      ▲ ForscherViking
41      @Override
42      public @Nullable PassiveEntity createChild(ServerWorld world, PassiveEntity entity) {
43          return (PassiveEntity) ModEntities.ModEntitiesMap.get("name_von_custom_entity").create(world);
44      }
45
46      ▲ ForscherViking
47      @Override
48      public boolean isBreedingItem(ItemStack stack) { return stack.isOf(Mod.HoleItem( name: "name_von_Item_zum_Paaren")); }
49
50      1 usage  ▲ ForscherViking *
51      @Override
52      protected void initCustomGoals() {
53          // Hier dann neue Ziele von Liste -- Bitte mit Priorität: 3 Anfangen (Also einfach 3, setzeZiel("..."))
54
55          this.goalSelector.add( priority: 3, setzeZiele( goal: "name_von_Ziel_von_Liste", entity: this));
56      }
57  }
```

Diese Klasse erbt von einer anderen Klasse – die also in der Hierarchie über der Vorlage steht – nämlich die Vorlage für alle Entities. Diese Klasse heißt *ModEntity*. Diese Klasse müsst ihr euch aber nicht

anschauen, alle wichtigen Anpassungen und Veränderungen könnt ihr ganz entspannt in der *CustomEntityVorlage* Klasse vornehmen. Die wichtigen Funktionen der Vorlagen Klasse sind:

- *createChild(...)* {...}
- *isBreedingItem(ItemStack stack)* {...}
- *initCustomGoals()* {...}
- *setzeVerfuehrungsItem()* {...}

Die erste Funktion dient dem Erzeugen eines Babys, wenn sich zwei dieser Tiere paaren. Dafür müssen wir nur in den Aufruf in den grünen Teil den Namen unserer Entity reinschreiben. Bedeutet also z.B. für das Stachelschwein steht dann da:

```
@Override
public PassiveEntity createChild(ServerWorld world, PassiveEntity entity) {
    return (PassiveEntity) ModEntities.ModEntitiesMap.get("porcupine").create(world);
}
```

Wichtig ist, dass ihr hier – genau wie bei den Items im Code – bitte immer genau dieselbe Schreibweise für eure Entities benutzt. Und kleingeschrieben versteht sich....

Als nächstes haben wir die Möglichkeit ein Paarungsitem festzulegen – um die Babys aus der Funktion zuvor zu ermöglichen. Solltet ihr die Funktion leer lassen, ist das Standarditem vom Spiel aus Getreide. Hier könnt ihr jetzt aber z.B. eines eurer eigenen FoodItems als Paarungsitem verwenden. Das Stachelschwein z.B. verwendet:

```
@Override
// Anpassen des Paarungs-Essens
public boolean isBreedingItem(ItemStack itemStack) { return itemStack.isOf(Mod.HoleItem( name: "tomato")); }
```

Die CustomGoals dienen dazu, der Entity ein Wesen zu verleihen – also Ziele, die es ausführt in der Welt. Sonst steht sie nämlich nur dumm in der Gegend herum. Wir haben eine Liste mit Zielen, die ihr verwenden könnt, zusammengestellt. Diese Liste sieht wie folgt aus:

- Verführen\_Ziel → Die Kreatur lässt sich von einem bestimmten Item Locken
- Folge\_Eltern\_Ziel → Die Kreatur folgt ihren Eltern in der Welt umher
- Herumlaufen\_Ziel → Die Kreatur läuft frei in der Welt umher
- Anschauen\_Ziel → Die Kreatur schaut den Spieler in einer bestimmten Reichweite an
- Herumschauen\_Ziel → Die Kreatur schaut sich in der Welt um
- Grasen\_Ziel → Die Kreatur grast das gras vom grasblock grasend ab
- Ausser\_Sonne\_Ziel → Die Kreatur läuft freiwillig aus der Sonne
- Atme\_Ziel → Die Kreatur braucht unbedingt Sauerstoff – und versucht saubere Luft zu atmen

Auf die Ziele könnt ihr – wie in der *PorcupineEntity* Klasse zu sehen – wie folgt benutzen:

```
@Override
protected void initCustomGoals() {

    //Custom Goals anpassen - Schlüsselwörter in Doku
    this.goalSelector.add( priority: 3, setzeZiele( goal: "Verführen_Ziel", entity: this));
    this.goalSelector.add( priority: 4, setzeZiele( goal: "Folge_Eltern_Ziel", entity: this));
    this.goalSelector.add( priority: 4, setzeZiele( goal: "Herumlaufen_Ziel", entity: this));
    this.goalSelector.add( priority: 5, setzeZiele( goal: "Anschauen_Ziel", entity: this));
    this.goalSelector.add( priority: 6, setzeZiele( goal: "Herumschauen_Ziel", entity: this));
    this.goalSelector.add( priority: 7, setzeZiele( goal: "Grasen_Ziel", entity: this));
}
```

Als letztes haben wir die Möglichkeit – falls nötig – ein Verlockungsitem festzulegen – also ein Item, welchem die Kreatur folgen wird. Das ist nur relevant, solltet ihr das Verführen\_Ziel in der Liste haben.

Wenn ihr ein Item aus Minecraft nehmen wollt, schreibt ihr folgendes:

```
public ItemConvertible setzeVerfuehrungsItem() {  
    VerfuehrungsItem = Items.APPLE;  
    return VerfuehrungsItem;  
}
```

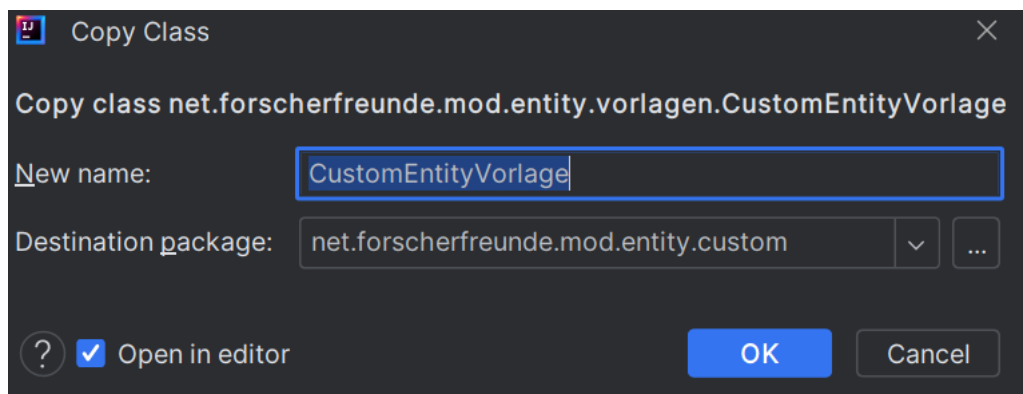
und schreibt dann statt APPLE eben das Item, welches ihr nutzen wollt. Wenn ihr *Items*. schreibt, wird euch auch eine Liste an Optionen angezeigt, an Items, die ihr verwenden könnt.

Die Alternative ist, euer eigenes Item zu verwenden. Dafür schreibt ihr:

```
public ItemConvertible setzeVerfuehrungsItem() {  
    VerfuehrungsItem = ModItems.GetItem("tomato");  
    return VerfuehrungsItem;  
}
```

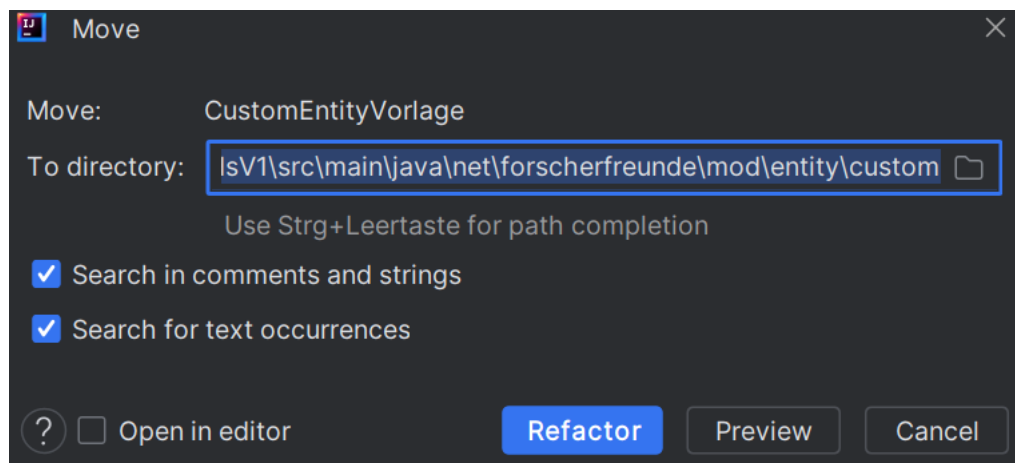
und ersetzt dann *tomato* mit eurem eigenen FoodItem.

Wenn ihr jetzt eure eigene Entity Klasse erstellen wollt, haltet ihr die Klasse *CustomEntityVorlage* einfach mit der linken Maustaste gedrückt, und zieht diese – während ihr *strg* gedrückt haltet – in den Ordner *custom* hinein. Dann sollte sich folgendes Dialogfenster öffnen:



Hier geben wir dann den Namen eurer Entity ein, also z.B.: *SlenderPeterEntity* und klicken dann auf ok.

Sollte sich stattdessen folgendes Fenster geöffnet haben:



Dann klicken wir bitte auf *cancel*. Achtet darauf, dass ihr die Klasse **kopiert** und nicht **bewegt** (also **Copy Class**, nicht **Move**).

Wenn ihr stattdessen ein Monster erstellt habt, dann schauen wir uns mal die Klasse *CustomMobVorlage* an. Die sieht der anderen Klasse sehr ähnlich:

```
9
10  /*
11  + BITTE GENAU LESEN!! +
12  * Diese Klasse ist eine Beispielklasse zum Kopieren und Anpassen - alle genauen Schritte sind in der Doku [Tag 4] in
13  * [Kapitel 4.2] zu finden. Bitte achtet genau auf die Doku, da ihr sonst eventuell Fehler einbaut, die schwer zu finden sind.
14  * Bei Rot unterstrichenen Fehlern wie hier - Alle - bitte euren Teamer fragen, dass er sich den Fehler im Code
15  * einmal anschauen kann. Diese Fehler führen dazu, dass das Programm gar nicht erst ausgeführt werden kann. Ihr bekommt
16  * dann auch direkt einen Fehler in der Konsole ausgespuckt. Vermutlich liegt es dann an einem Rechtschreibfehler oder
17  * Syntaxfehler. Meistens gibt euch IntelliJ auch schon Vorschläge, jedoch bitte nicht einfach die Vorschläge
18  * ausführen, da das manchmal auch zu noch mehr Fehlern führen kann.
19  *
20  * Viel Spaß mit der Klasse.
21  * @Author
22  * Matthias Mendler
23  */
24
25  1 usage  ▲ ForscherViking *
26  public class CustomMobVorlage extends ModMob{
27      no usages  ▲ ForscherViking
28      protected CustomMobVorlage(EntityType? extends HostileEntity> entityType, World world) {
29          super(entityType, world);
30      }
31
32      no usages  ▲ ForscherViking *
33      public static DefaultAttributeContainer.Builder createMobAttributes() {
34          //hier werden die Basisattribute des Mobs gesetzt - hier könnt ihr diese also ganz einfach anpassen.
35          return MobEntity.createMobAttributes()
36              .add(EntityAttributes.GENERIC_MOVEMENT_SPEED, baseValue: 0.2f)
37              .add(EntityAttributes.GENERIC_ATTACK_DAMAGE, baseValue: 5)
38              .add(EntityAttributes.GENERIC_ATTACK_KNOCKBACK, baseValue: 1);
39      }
40
41      1 usage  ▲ ForscherViking *
42      protected void initCustomGoals() {
43          //hier dann custom Ziele des Mobs angeben - eigentlich sind hier keine nötig
44      }
45  }
```

Jedoch seht ihr, dass es weniger Funktionen gibt. Wir haben nur eine wichtige Funktion – *createMobAttributes() {...}*.

Diese Funktion gibt eurem Monster alle wichtigen Grund Attribute, also Bewegungsgeschwindigkeit, Angriffsschaden, Angriffs Rückstoß usw. ...