JMOD: The JavaScript Mod Loader

A Tweaking Tool for Modpack Authors

Designed by JeffPeng
Documented by Reteo

October 17, 2016

Contents

Introduction

Modding Minecraft is a complex task. There are a lot of steps involved, and requires a significant amount of knowledge in programming, particularly the ability to understand object-oriented programming in Java. Because of this, mod tweaking programs started appearing on the scene with simpler configurations, such as the ever-popular MineTweaker (and its addon, ModTweaker). However, because they are creating their own configuration languages, the results are less-than-stellar. They spend time in adding tweaking features, but they do not put as much effort into the language itself.

Because of this, I, the creator of the Survival Industry, started piling on tweak tool after tweak tool; Minetweaker and Modtweaker, SquidUtils, Block Properties, CustomItems... It was getting to the point that I had more tweak tools than mods. Then I managed to get JeffPeng on my team, and he started working on this tool.

We decided to create a new mod format (called the jmod file), which is a zipfile containing all the scripts and assets (language files and textures) made for the modpack, which, once compressed, is simply placed in the mods folder. This means that you can create scripts for your modpack, and distribute them as an actual mod, only requiring JMOD to be included as the modloader. For your convenience, we'll separate the loader and the mods it loads by case; we'll call the loader "JMOD," and the javascript mod file a "jmod."

JMOD includes many important features for tweaking Minecraft, including:

- Creation of new items, blocks, and materials.
- Customization of block and item properties, including settings for food, armor, and tools.
- Adjustment of chest loot, either specific to locations, or globally.
- A built-in ore generation engine, to distribute any ores created in the scripts.
- Custom recipes, both shaped and shapeless.
- New creative tabs for your custom items.
- Custom log messages and tooltips, complete with colored text.
- Ore Dictionary tweaking.
- An alloying feature making use of a crucible and a mold (you'll need to make these, though).
- Built-in support for Chisel's chisel (including the creation of additional chisels), as well as RotaryCraft machinery.

To simplify the process, we use the JavaScript language built into Java 8 (called "Nashorn") to make the language much more complete. Loops, conditionals, classes, functions... they're all there, and working without an issue. And JavaScript is a standard that many programmers know or could easily learn, with documentation everywhere.

However, there are commands specific to JMOD that are not documented anywhere else, which is where this document comes into play. In the following sections, I will be listing all the major commands for this scripting tool, including their usage, and some examples for good measure. Hopefully, by the time you're done with this document, you should know enough about JMOD to start making your own jmods. And no, you are not limited to just one jmod. If it helps, split them out as you wish.

Note that JMOD is alpha software, and the commands are likely to change in the future to accommodate new features, so while I'll try and keep this document up-to-date, I can't guarantee the information is accurate in reference to a version of JMOD newer than the above date. However, this should give you a good head-start. Note that as of this document, the current version of JMOD is Alpha-1.0.5

1 The Jmod File

Before we begin with the scripts, I want to make sure you understand this. A jmod is a zipfile with the `.jmod` extension. This prevents Forge's modloader from trying to load this file, and ensures that JMOD does.

The jmod zipfile contains the following:

- A `mod.json` file with the mod information for the modloader.
- One or more javascript files with the actual mod commands.
- An image file with the logo for your jmod, if desired.
- An "assets" folder containing all the language and texture resources used by the jmod.

1.1 mod.json

This file is a JSON file, and it is the key to getting the jmod working. It lists several things:

modid The Mod ID of your jmod, which will be used to ID objects created in the jmod.

name The actual name of the jmod.

description A description of the jmod, which will be visible from the Forge mods list.

version The version number of the jmod, provided you are releasing new versions as you go.

scripts This lists the scripts that JMOD will need to run in order to make the mod happen.

authors More informational data for the Forge mod list.

url A website address where the jmod (or modpack) can be acquired.

logo This points to the above image file, if you desire a logo.

credits It's just polite to credit your sources.

1.2 Javascript Files

These are the meat of the jmod. You can have one script covering everything, or else, you can set things up so that the first script (identified in the `mod.json` file) loads other scripts, so you can keep features separated. It is in these files that you can place the commands listed in the following sections.

1.3 Logo Image

As previously mentioned, JMOD will actually add your jmods into the Forge mod list. This means that it wouldn't hurt to include a logo image to make it look good in the list. The file should be a PNG file.

1.4 Assets

This is the folder containing all the resources for the jmod. The sub-folder containing the resources for the custom items created in the jmod should be named after the `modid` field of the `mod.json` file.

For example, if the jmod's `modid` is "Greatmod", then that is the folder you should make for custom language and texture files in assets, e.g. `assets/Greatmod`.

1.5 Making the jmod

To complete the process of making the JMOD, make all the above files (as needed), then zip them up. You should then change the suffix of this zipfile from `.zip` to `.jmod`. Once the suffix is changed, copy the newly-minted jmod file to your mods folder, and the jmod should be ready for JMOD to load it.

2 The JMOD Commands

The following are the commands used specifically by JMOD to make the tweaks to your game.

2.1 Basic Script and Mod Commands

These commands cover the complex task of loading other scripts at the right times, as well as checking for mods, either as a way to test their presence, or a warning for players who remove a

mod needed for the specific pack.

2.1.1 Loading Other Scripts

Loads another JavaScript file from within the jmod.

Command: `loadjs("filename")`

filename: This is the filename of the script you wish to load.

2.1.2 Checking for Other Mods

Checks to see if a mod is loaded. This command returns "true" if the mod is loaded, and "false" otherwise.

Command: `isModLoaded("modid")`

modid: This is the Mod ID of the mod you are checking.

2.1.3 Modpack Dependencies

This command will pop up a message if the specified mod is not loaded. Useful in modpacks depending on certain mods, just in case players manually remove them.

Command: `dependency("emphmodid", "mod name")`

modid: This is the Mod ID of the mod you are checking. Returns "true" if the mod is loaded, "false" if not.

mod name: This is the actual name of the mod you are checking.

2.2 Logging and Tooltips

These commands cover the information in the game, either with tooltips that appear when you mouse over an item, or log messages that you can place to help debug problems in your scripts.

2.2.1 Adding a Log Message

Displays the message in the minecraft log.

Command: `log("message")`

message: This is the the message that gets displayed in the log.

2.2.2 Adding a ToolTip

Displays a tooltip when the mouse hovers over a specific object. The tooltip lines should be identifiers; you should actually write the tooltips in the appropriate language file. This will allow tooltips to be translated to other languages without changing the javascript. Additionally, tooltip messages do not wrap to the next line, so try to keep the tooltip lines short, and add lines to complete the message.

Command: `addToolTip("target","tooltip lines")`

target: The object the tooltip messages will be applied to.

tooltip lines: A comma-separated list of strings (each enclosed by quotes), each is one line of the tooltip being applied.

2.3 User Interface Tweaks

These make changes to the existing user interfaces for Creative Mode and Not Enough Items.

2.3.1 Making Creative Mode Tabs

Creates new tabs in the creative mode user interface.

Command: `addCreativeTab("tabid", "Tab Name", "target")`

tabid: This is the ID name for the tab.

Tab Name: This is the name of the tab that will appear on the tab's tooltip.

target: This is the first item to be added to that tab. You cannot have an empty creative mode tab.

2.3.2 Hiding Something from "Not Enough Items"

When assigned an item ID, the item will be removed from the NEI item list (on the right side of tile entities' GUIs). Useful if trying to hide items not used in a modpack.

Command: `hideFromNEI("target")`

target: The object that will be hidden from NEI

2.4 Creating Things

You can create items, blocks, and materials without limit. Note, however, this does not include tile entities (blocks that have special functions); those need to be coded in Java, and cannot be made using JMOD.

2.4.1 Adding Tool Materials

This creates the material that defines the ability of tools made from that material. This command can also be used to change existing tool materials, such as STONE, WOOD, and IRON. If you use the tool repair features of JMOD, it is recommended to do this with all known materials, even if only to add the repair material to them (for example, plankWood to WOOD, or ingotGold to GOLD).

Command: `addToolMaterial("material id", harvest level, durability, efficiency, damage, enchantability, "repair material")`

material id: The ID of the tool material. Usually, the ID is the name of the material, all in uppercase (such as "IRON")

harvest level: The tool can harvest blocks of this harvest level and lower.

durability: This is the number of uses a tool has before it's broken.

efficiency: This is how fast the tool can harvest blocks.

damage: This is how much damage is done by a weapon.

enchantability: This is how enchantable a tool is. The higher the number, the greater the enchantability.

repair material: This is the item ID of the object needed to repair tools using the advanced repair features of JMOD.

2.4.2 Adding Armor Materials

This creates the material that defines the ability of armor made from that material.

Command: `addArmorMaterial("material id", base damage reduction, helmet modifier, chestplate modifier, leggings modifier, boots modifier, enchantability, "repair material")`

material id: The ID name of the armor material.

base damage reduction: The base amount of damage reduction, doubles as durability.

helmet modifier: The amount of extra damage reduction applied to a helmet.

chestplate modifier: The amount of extra damage reduction applied to a chestplate.

leggings modifier: The amount of extra damage reduction applied to leggings.

boots modifier: The amount of extra damage reduction applied to boots.

enchantability: This is how enchantable armor made from this material is.

repair material: This is the item ID of the object needed to repair the armor using the advanced anvil repair features of JMOD.

2.4.3 Adding Alloys

Defines an alloy based on two input ingots. This is applied by JMOD to the crucible/mold alloying process (see 3.1.2).

Command: `addAlloy("result", "first ingot", "second ingot", resulting amount)`

result: This is the item ID of the object formed by the alloying process.

first ingot: This is the item ID of one of the items used in the alloying process.

second ingot: This is the item ID of another of the items used in the alloying process.

resulting amount: This is how many of the resulting material is created by alloying.

2.4.4 Ore Multiplying

This uses the alloying command above, but only takes in one material, and is useful in making a general-purpose ore multiplying mechanic without the need for special tile entities.

Command: `addAlloy("output", "input", output amount)`

output: This is the item ID of the ingot formed by the process.

input: This is the item ID of the ore used in the process.

resulting amount: This is how many of the resulting material is created by the process.

2.4.5 Adding Items

The item addition command is a complex one, because it includes not just the basic item, but also three special types of item (armor, tools, food) that each have their own special options.

Command: `addItem("item id name", "class", stack size, "creative tab id")`

item id name: This is the item ID of the item being created, without the mod name prefix (the name before the first colon).

class: This is the item class, and can use classes from both vanilla Minecraft, as well as mods.

stack size: This is the maximum stack size of the item.

creative tab id: This is the ID name of the creative mode tab this item will be placed in.

There are three important types of item that can be added with special properties assigned to them in the script: food, tools, and armor.

Note: if you wish to create metal ingots, there is another command below you can use instead of this one that will simplify the process.

Food

Food is a special item that can be eaten in order to add "shanks" to your hunger bar. Each "shank" is worth two hunger. To add hunger satisfaction and other properties to food items, simply add:

Method: `.FoodData(hunger, saturation, wolf food, always edible)`

hunger: This determines how much hunger food satisfies.

saturation: This determines how much saturation food provides.

wolf food: Can this be fed to wolves to befriend them? Answer is boolean, meaning true or false.

always edible: Is this food always edible (even when full)? Answer is boolean, meaning true or false.

the .FoodData method also has a separate method that allows food to supply enchantments to the player eating it.

Method: `.buffdata("enchantment id", duration, level, chance)`

enchantment id: The ID name of the enchantment that buffs (or debuffs) the player.

duration: This determines how long (in seconds) the enchantment lasts.

level: This is the level of the enchantment provided by the food.

chance: The chance (in %) that the enchantment will be applied by the food.

Tools

A tool applies the properties assigned by the tool material it is made of. However, for this to happen, you need to assign the material to the tool item.

Method: `.ToolData("material")`

material: The name of the tool material applied to the tool item.

Additional properties can be applied to the tool using additional ToolData methods.

First, there's the tool's durability:

Method: `.durability(durability)`

durability: The tool's individual durability value.

Then, its tool class:

Method: `.toolclass("tool class")`

tool class: The class name of the tool.

Next, we can decide if we want the tool to be breakable or not:

Method: `.unbreakable(value)`

value: Does the tool ignore endurance? Answer is boolean, meaning true or false.

Finally, we can determine if the tool has modes (only applies for tool classes that have available modes):

Method: `.hasModes(value)`

value: Does the tool have modes? Answer is boolean, meaning true or false.

Armor

Finally, armor has its own properties, based on what it's made of, and what part is being defined.

Method: `.ArmorData("material", "type")`

material: The material ID used by the armor piece.

type: The type of armor this is; options are "Helmet", "Chestplate", "Leggings", and "Boots".

2.4.6 Adding Metal Ingots

You can add ingots of a specific type of metal. This is mostly a convenience; ingots created with this will automatically be given appropriate classes, item IDs, and ore dictionary names.

Command: `addMetalIngot("name")`

name: This is the material name of the ingot being created.

2.4.7 Adding Blocks

Items are things that can have a function while being held. Blocks, on the other hand, are things that can be placed into the world. Note, this does not include tile entities, which are blocks that have actual functions while placed in the world. This is just for blocks. Due to the nature of tile entities, it is not currently feasible to make them using JavaScript, as they would perform horribly.

Command: `addBlock("block id name", "class", hardness, blast resistance, "tool", harvest level, "material", "tab")`

block id name: This is the block ID of the item being created, without the mod name prefix.

class: This is the block class, and can use classes from vanilla Minecraft, as well as mods.

hardness: This is the hardness value of the block; a higher value means it takes longer to mine.

blast resistance: This is the blast resistance value of the block; the higher the value, the more power an explosion will need to break it.

harvest level: This is the harvest level a tool will need to be in order to harvest this block.

material: This is the material of the block.

tab: This is the name of the creative mode tab the block will be placed in.

2.4.8 Adding Metal Blocks

You can add blocks of a specific type of metal. This is mostly a convenience; blocks created with this will automatically be given appropriate classes, item IDs, and ore dictionary names.

Command: `addMetalBlock("name")`

name: This is the material name of the block being created.

2.4.9 Adding Fluids

Fluid blocks are unlike other blocks in that they have a source and flow block, as well as several qualities to pick from.

Command: `addFluid("name")`

name: This is the name of the fluid being created.

Now, fluids have several methods to set various states of the fluid in question.

The first is viscosity, the speed at which a fluid flows.

The tickrate of a fluid (how often it updates) is determined by its viscosity value using the following calculation:

$$tickrate = \frac{viscosity}{200}$$

1000 is water's viscosity; higher numbers increase the number of ticks between updates, while lower numbers decrease this number.

Method: `.viscosity(value)`

value: The viscosity value for the fluid.

The next is density. This is calculated in kilograms per cubic meter. Fluids with a density of 0 can stay in place, while negative density values will cause the "liquid" to float up.

Method: `.density(value)`

value: The density value for the fluid.

Onto temperature. The temperature of a liquid affects a few things. While fluids created by addFluid can't change to ice, a fluid with a negative temperature can turn adjacent water to ice. Similarly, a fluid that is over 300 degrees can start setting fires (think "lava").

In addition to freezing water and setting fires, there's also the fluid's affect on the player. At -50, "cold damage" will be applied to the player (the effect was added by JMOD). At -100, the damage becomes much worse, and it can slow a player down (hypothermia). On the other side, 100 degrees can do damage to a player, and 300 degrees will both damage a player and set them on fire.

Method: `.temperature(value)`

value: The fluid's temperature in Celcius.

Next up: Color. The fluid's color in block and bucket forms are pretty much set by the jmod's textures, but this value sets the color used by tanks and other mod interactions with the fluids in order to keep the fluid's appearance consistent.

Method: `.setColor(red value, green value, blue value)`

red value: How much red is in the color, from 0-255.

green value: How much green is in the color, from 0-255.

blue value: How much blue is in the color, from 0-255.

Speaking of buckets, there may be cases where you don't want a fluid to have a bucket, such as negative-density fluids that a bucket would not be able to hold. To accomodate those who want to make the call themselves, the hasBucket() method was added to make whether or not a fluid has a bucket or not explicit. There are no arguments; only the presence of this method matters.

NOTE: This only affects the vanilla bucket (and those interacting with forge fluids); the wooden bucket is strictly limited to water and milk.

Method: `.hasBucket()`

Some fluids can poison a player. Using this command tells Minecraft that this added fluid is one such fluid.

Method: `.isPoisonous()`

When submerged in a fluid, there is a chance that you can drown. However, forge treats gases as fluid, too, for the sake of simplifying fluid and gaseous dynamics. So, to separate breathable gases and non-breathable fluids, this command determines a fluid to be gaseous.

NOTE: being gaseous does not affect how a fluid flows; it will still flow like water (think "dry ice smoke"). The direction of flow is still controlled by density, and the speed of flow is still controlled by viscosity.

Method: `.isGaseous()`

2.5 The Ore Dictionary

These allow you to add, remove, change, or consolidate ore dictionary entries.

2.5.1 Adding to the Ore Dictionary

You can add items to the ore dictionary, add new entries to the ore dictionary, or both.

Command: `addOreDict("block id name", "ore dictionary entry")`

block id name: This is the ID of the item being added to the ore dictionary. You can use another ore dictionary name in place of the block ID, and it will merge it into the desired ore dictionary entry.

ore dictionary entry: This is the ore dictionary name you want to add to.

2.5.2 Removing from the Ore Dictionary

You can also remove items from the ore dictionary.

Command: `removeOreDict("block id name", "ore dictionary entry")`

block id name: This is the ID of the item being removed from the ore dictionary.

ore dictionary entry: This is the ore dictionary name you want to pull from.

2.6 Recipes

One of the mostly-used tweaks is the ability to alter recipes.

2.6.1 Removing Recipes

Deletes an exsiting crafting table recipe for an item.

Command: `removeRecipes("block id name")`

block id name: This is the ID of the item whose recipe needs to be removed.

There is also a command for removing furnace recipes.

Command: `removeSmeltingRecipes("block id name")`

block id name: This is the ID of the item whose recipe needs to be removed.

There are other machines to whom recipes can be added, but they will be addressed later covering mod-specific tweaks.

2.6.2 Adding Recipes

Adds a new recipe for an item. For the crafting table, there are commands for shaped and shapeless recipes.

Command: `addShapelessRecipe("block id name", ["ingredient", "ingredient", ... "last ingredient"])`

block id name: This is the ID of the item to which the recipe is to be added.

ingredient list: A comma-separated list of item IDs or ore dictionary entries that go into making the object. Since the recipe is shapeless, order doesn't matter, but you cannot have more than 9 ingredients (for obvious reasons).

Command:
```
addShapedRecipe("block id name", [
["ingredient", "ingredient", "ingredient"],
["ingredient", "ingredient", "ingredient"],
["ingredient", "ingredient", "ingredient"]])
```

block id name: This is the ID of the item to which the recipe is to be added.

ingredient list: A comma-separated list of 3 lists, each containing three Item IDs, also comma-separated. These correspond to the first, second, and third slots of the top, middle, and bottom rows of the crafting grid (in that exact order). Empty slots are simply represented by a "null" (without the quotes). To make it easier to understand, it is displayed here with "ingredient" in place of whatever is placed in that slot, and the whole line split into lines to show the rows.

For the furnace, you can add smelting recipes.

Command: `addSmeltingRecipe("block id name", "ingredient")`

block id name: This is the ID of the item to which the recipe is to be added.

ingredient list: This is the ID of the item that is smelted into the above item.

2.7 Changing Things

Making your own blocks is well and good, but it's important to be able to adjust them or other blocks as well. This is especially important if you expect a block to drop something other than the block itself when broken.

2.7.1 Setting Block Properties

Before you can set block properties, you need to tell JMOD what block you will be setting the properties on.

Command: `setBlockProperties("block id name")`

block id name: This is the ID of the block to be edited.

This command simply identifies the block to be changed. The actual changes are methods added to the command that can affect block properties.

Hardness determines how long it takes to mine a block. For example, your typical ore has a hardness of 3, while obsidian has a hardness of 50.

Method: `.hardness(value)`

value: The specific hardness value.

Slipperiness defines how far a player or items slide on a block. It can also speed up water flow on top of it. Slipperiness values above 1.02 can increase the movement of objects sliding on it.

Method: `.slipperiness(value)`

value: The specific slipperiness value. Ranges from 0 (not slippery at all) to 1.0 (has no friction at all) and above (increases object's speed).

Harvest Level of a block is tied to the harvest level of a tool. If the tool's harvest level is above the block's harvest level, the block can be harvested by that tool. If the block's harvest level is higher, then it takes longer to break, and drops nothing.

Method: `.harvestlevel(<meta,> harvestlevel)`

meta: The block's meta the harvest level is applied to. This is optional, but useful for mods that topload all their ores into a single block ID using meta values.

harvestlevel: The specific harvest level. It must match the lowest-leveled tool that is needed to harvest this material. For example, redstone ore has a harvest level equal to the iron pickaxe.

The tool type identifies which tool type is effective for collecting the material.

Method: `.tool("tooltype")`

tooltype: This should be the specific tool required to harvest the block. Tools include "shovel", "axe", "pickaxe", and "sword".

The sound type determines the sounds assigned to a block when being walked on, harvested, etc.

Method: `.sound("soundtype")`

soundtype: This is based on vanilla's sound types, and include:

- "stone"
- "grass"
- "snow"
- "wood"
- "cloth"
- "ladder"
- "gravel"
- "sand"
- "anvil"

The opacity of a block determines how much light gets through the block.

Method: `.opacity(value)`

value: The specific opacity value. Ranges from 0 (completely transparent) to 255 (doesn't block any light). Note, this value affects the block's ability to allow light through. It does not affect the appearance of the block, which is determined by the texture.

The blast resistance of a block determines how likely the block will survive an explosion, and whether it will drop blocks.

Method: `.blastresistance(value)`

value: The specific blast resistance value. Ranges from 0 (steve could sneeze and break it) to 30 (most solid blocks) to 500 (most liquids) to 6000 (obsidian) to 18000000 (18 million; indestructable blocks like bedrock).

2.7.2 Adding Block Drops

You can add drops to existing blocks, in case you want to add a specific type of item that drops from a block (such as redstone or diamonds).

Command: `addSmeltingRecipe("block id name", "item id name", chance, exclusive, playeronly)`

block id name: This is the ID of the block to break.

item id name: This is the ID of the item that you want to drop from the block.

chance: This is the percentage chance of the drop happening, in case you want the drop to happen only occasionally. An example would be how gravel only sometimes drops flint.

exclusive: Determines if the block drops with the item. An example would be how gravel will not drop both gravel and flint. Boolean (true or false)

playeronly: Does the block only drop its items for players? An example would be how coal ore drops coal when mined, but coal ore blocks in a creeper explosion. Boolean (true or false).

2.7.3 Setting Armor Material Color

Unlike items and blocks, there's currently no way to set armor textures in JMOD (yet). However, you can assign color to armor materials in order to ensure the armor you make will be uniquely colored.

Command: `defineColor("material id", red, green, blue)`

material id: This is the all-capital ID of the armor material; for example, "IRON".

red: This is the red value of the color, ranging from 0-255

green: This is the green value of the color, ranging from 0-255

blue: This is the blue value of the color, ranging from 0-255

2.8 Chest Loot

Sometimes you want your custom items to appear in chests. Other times, you want specific objects to be kept from chests. Either way, we've got you covered!

2.8.1 Adding Chest Loot

This allows you to add things to chests in the world.

Command: `AddChestLoot("item id name", minimum, maximum, weight, "target chest")`

item id name: Defines what you want to add to the chest.

minimum: This is the minimum stack size added to the chest

maximum: This is the maximum stack size added to the chest

weight: This is the weight value, or the chance that the object will be added to the chest over something else.

target chest: (Optional) This is the type of chest you want this loot to go into. The arguments are based on Forge's "ChestGenHooks" class, and the options include:

- "bonusChest"
- "villageBlacksmith"
- "mineshaftCorridor"
- "dungeonChest"
- "strongholdCorridor"
- "strongholdLibrary"
- "strongholdCrossing"
- "pyramidDesertyChest"
- "pyramidJungleChest"
- "pyramidJungleDispenser"

Chests added by other mods are also accessible if you know their ChestGenHooks names. If you don't include this option, then all chests are affected.

2.8.2 Removing Chest Loot

Sometimes, you don't want something in specific chests. Other times, you just don't want something in *any* chests. Can be used to enforce some kind of game progression.

Command: `removeChestLoot("item id name", "target chest")`

item id name: Defines what you want to add to the chest.

target chest: (Optional) This is the type of chest you want this loot to be excluded from. Arguments are much the same as the "AddChestLoot" command. If you don't include this option, then all chests are affected.

2.9 Chisel Tweaks

The Chisel mod includes a special GUI that allows you to make variations on a specific block. In some cases, however, you might want to add support for your own blocks, complete with your own custom variations. All Chisel commands are prepended with "Chisel." For the sake of simplicity, this has been included in the explanations below.

2.9.1 Adding a Chisel Group

A chisel group is a group of blocks that are related to a specific type. This can be a collection of differently-patterned stone, different types of light source, perhaps even different colors of the same block. A group is what you see when you plug a block into the chisel's base slot.

Command: `Chisel.addGroup("group name")`

group name: Name of the group that you want to add specific blocks to. The name does not affect the blocks in any way; it just gives the group a name, similar to how the

Ore Dictionary works. It must be created, however, in order to assign blocks to the group.

2.9.2 Adding Specific Blocks to a Chisel Group

Once you've named your chisel group, it's just a matter of adding the desired blocks to it.

Command: `Chisel.addVariation("group name", "block id name")`

group name: Name of the group that you want to add specific blocks to. Must be created using Chisel.addGroup.

block id name: The block ID of the block you wish to add to your new Chisel group.

2.10 RotaryCraft Tweaks

JMOD was written to support Survival Industry, which needed quite a few tweaks to integrate RotaryCraft with other mods more deeply than Reika's already significant investment allows. As such, JMOD is one of the few tweaking tools that can handle adding complex recipes for the Blast Furnace, as well as recipes for various other RotaryCraft machines.

As with other mods' tweaks, these commands need to be started with "RotaryCraft."

2.10.1 Blast Furnace Recipes

RotaryCraft uses its blast furnace to create high-strength, low-alloy steel, which is used in all its machines. This being a useful form of alloying, it seems silly not to include a way to add other alloying recipes to this setup. And since the input area is a 3x3 grid, this opens up all kinds of options for items that require crafting, but need to be "baked" into their forms.

Command:
```
RotaryCraft.addBlastFurnaceRecipe("item id name", temperature,
speed, xp, [["ingredient","ingredient","ingredient"],
["ingredient","ingredient","ingredient"],
["ingredient","ingredient","ingredient"]])
```

block id name: The item ID of the item you wish to craft in the blast furnace.

temperature: The temperature (in Celsius) the blast furnace needs to be at for the smelting to begin.

speed: The time (in seconds) that the smelting process takes.

xp: The experience gained from crafting/smelting.

ingredients: This is the shaped crafting recipe, in the same fashion as addShapedRecipe above. If a slot should be empty, the ingredient field should be "null" (without the quotes).

2.10.2 Blast Furnace Alloying

The blast furnace is an advanced alloying system. The idea is that the materials in the slots on the left are used to help convert the main materials (in the 3x3 grid) into the desired material.

Command: `RotaryCraft.addBlastFurnaceAlloying("item id name", "main ingredient", temperature)`

item id name: The item ID of the item you wish to craft in the blast furnace.

main ingredient: This is the main ingredient (as an item id name or ore dictionary entry) needed in the 3x3 grid to make the desired material. You can only have one material in that grid.

temperature: The temperature (in Celsius) the blast furnace needs to be at for the smelting to begin.

This command does not do much useful by itself; there are several methods that you use to complete the recipe in question.

First, is the other main component of the recipe; the items you place on the right-hand slots. You can apply this method three times, in order to configure all three slots. However, the number of items produced is equal to the largest of the matching materials; for example, if gunpowder can produce 2 steel, whereas the other ingredients are guaranteed only 1, then the blast furnace will produce 2 steel.

Method: `.input(number, "item id name", chance, decrease)`

number: Side slot used for this material (options are 1, 2, or 3)

item id name: The item ID of the side ingredient for the alloy.

chance: The chance that this item will be consumed upon alloying.

decrease: The amount of this ingredient consumed if the chance roll is successful.

Sometimes, you need a minimum amount of a material in the main grid in order to make something (an example would be 4 quartz to make a block of EnderIO fused quartz).

Method: `.required(number)`

number: Required number of materials in the main grid to make resulting output.

You can configure a specific alloying recipe to provide bonus output. The chance of the bonus is hardcoded into RotaryCraft, so there is no way to affect it from here. As such, this method has no arguments.

Method: `.bonus()`

You can also have the blast furnace provide XP for successfully smelting an alloy.

Method: `.setXP(number)`

number: Amount of XP provided by the process.

2.10.3 Grinder Recipes

RotaryCraft's grinder can also have recipes applied, similar to how the vanilla furnace recipes work.

Command: `RotaryCraft.addGrinderRecipe("ingredient", "result")`

ingredient: The item ID of the ingredient that goes into the grinder.

result: The item ID of the output of the grinder.

The grinder's output numbers seem to be hardcoded into RotaryCraft; there does not seem to be a way to alter them from here.

2.10.4 Centrifuge Recipes

RotaryCraft's centrifuge can set outputs for specific inputs, along with the chances of that output occurring. This can be useful if using RotaryCraft with skyblock maps, as the centrifuge can be used as an advanced sifter.

Command: `RotaryCraft.addCentrifugeRecipe("ingredient")`

ingredient: The item ID of the ingredient that goes into the centrifuge.

The centrifuge's outputs are controlled by methods, rather than identified in the main command. There are two output types: item outputs and fluid outputs.

For item output, you have the following method. Note that the chance is a floating-point number, so you can make use of decimals.

Method: `.addOutput("output", chance)`

output: The item ID of what we want to output. If you end the output id with "@" and a number, you can determine a quantity.

chance: The chance, in percentage form, that the item will be extracted from the ingredient.

For fluids, you have the following method.

Method: `.addOutput("output", chance)`

output: The fluid ID of what we want to output. Like with item, the @ and a number can give an amount, which is measured in millibuckets.

chance: The chance, in percentage form, that the item will be extracted from the ingredient.

2.10.5 Pulse Jet Furnace Recipes

The pulse jet furnace is far hotter and faster than a friction-heated furnace or blast furnace can manage. The main use of the Pulse Jet Furnace is to melt items back into their key metal, but it also is used to make blast-resistant glass. With this command, you can make your own pulse-jet furnace recipes.

Command: `RotaryCraft.addPulseJetRecipe("input", "output")`

input: The item ID of the item that goes into the pulse jet furnace.

output: The item ID of the output of the pulse jet furnace.

2.10.6 Compactor Recipes

Where the point to the blast furnace and the pulse jet furnace is "temperature," the point to the compactor is "pressure." This machine can be used to make things that are reasonably expected to be formed from heat and pressure (for example, there is a recipe for diamonds by default). With this command, you can add your own pressurized items.

Command: `RotaryCraft.addCompactorRecipe("input", "output", temperature, pressure)`

input: The item ID of the item that goes into the compactor.

output: The item ID of the output of the compactor.

temperature: The temperature (in Celcius) needed for the compactor to work. This number is an integer, do not use decimals.

pressure: The pressure level (in kilopascals) required by the compactor (applied through torque). This number is an integer, do not use decimals.

2.10.7 Drying Bed Recipes

The drying bed is simply a machine that takes a liquid, and "dries it out" to result in... whatever's not water. When using this on water, the result is salt. With this command, you can add your own drying bed recipes.

Command: `RotaryCraft.addDryingBedRecipe("input", "output")`

input: The name of the fluid that goes into the drying bed.

output: The item ID of the output of the drying bed.

2.10.8 Rock Melter Recipes

The rock melter is the opposite of the drying bed. Where the drying bed dries out a liquid to produce an item, the rock melter uses heat to melt items into liquids.

Command: `RotaryCraft.addRockMelterRecipe("input", "output", temperature, power)`

input: The item ID of the item that goes into the rock melter.

output: The fluid name of the output of the rock melter.

temperature: The temperature (in Celcius) needed for the rock melter to work. This number is an integer, do not use decimals.

power: The power (in Watts) required by the rock melter. This number can include decimals.

2.10.9 Liquefaction Machine Recipes

The liquefaction machine mixes a liquid into a solid, making a different solid. For example, if you add a mud block, then you can add a recipe for this machine to combine water and a dirt block to make your mud block

Command: `RotaryCraft.addLiquefactionRecipe("input", "output", "liquid input", time)`

input: The item ID of the item that goes into the liquifaction machine.

output: The item ID of the output of the liquifaction machine.

liquid input: The name of the liquid that goes into the liquifaction machine.

time: The amount of time required by the liquifaction machine. This number is an integer, do not use decimals.

2.11 AppleCore Tweaks

JMOD can work with AppleCore to allow tweaks of existing food values. That way, if bread is too satisfying, or apples are not satisfying enough, or if melons should do more saturation than they do, you can change them from within JMOD. AppleCore is obviously required for this to work.

2.11.1 Modifying Food Values

You can change the hunger and saturation of a food item.

Command: `Applecore.modifyFoodValue("food", hunger, saturation)`

food: The item id of the food in question.

hunger: How much satisfaction (in half-shanks, up to 20) the food can satisfy. Must be an integer (no decimals).

saturation: How long before the hunger bar starts to fall again. You can use a float (decimals allowed) up to a maximum of 10.0.

2.12 Ex Nihilo Tweaks

JMOD can work with Ex Nihilo to affect what the mod can do. This can allow significant customization of a skyblock game.

2.12.1 Adding Sifting Recipes

You can add additional recipes to the sifter to allow drops of additional items (such as things made with JMOD).

Command: `ExNihilo.addSifting("output", "input", rarity)`

output: The item ID of what we want to get from the sifter.

hunger: The item ID of the item we're sifting in the sifter.

saturation: The rarity of an item, as a ratio of 1:rarity. Must be an integer.

3 Additional Features

Some additional features have been added to JMOD as a way to expand gameplay.

3.1 Basic Ore and Smelting Support

Among the things included in this system is the ability to add worldgen for any custom ores you make, as well as a way to create alloys without needing special machines to do so.

3.1.1 Adding Ore Generation for Custom Ores

If you create ore blocks, you can use this command (and its attendent methods) to configure Minecraft to spawn the ore as desired. That way, you don't need any external ore generation mods to handle the process.

Command: `addOreGeneration()`

By itself, this command doesn't do much; you need to add methods to it to assign the values to the ore generation engine.

First, you need to assign a block ID to be placed.

Method: `.blockToGenerate("block id name")`

number: Block ID of the ore block you wish to be generated.

Next, you need to assign a block ID that it will replace. You can use this method multiple times.

Method: `.blockToReplace("block id name")`

number: Block ID that you wish to replace with the ore block.

Then, you need to tell the generator the maximum number of times (per chunk) ore veins will generate. This only provides the maximum number of chances; Minecraft uses a random number generator for each chance to determine if ores will spawn that time.

Method: `.chancesPerChunk(value)`

value: Number of chances that ore veins will spawn in a single chunk.

Now that we know how many chances Minecraft will have to spawn veins per chunk, let's define exactly how big said veins should be. Once again, this is a maximum number, and veins have the chance to spawn smaller.

Method: `.veinSize(value)`

value: Size of vein in number of ore blocks.

Next, you need to let the generator know in which dimensions it needs to spawn the ore in. The generator uses dimension IDs, including 0 for the Overworld, 1 for the End, and -1 for Nether.

Method: `.dimension(value)`

value: Dimension number that hosts the ore.

Now that we know what dimensions we want to use, let's determine how deep we want the ore spawning. There are two methods, one determining the lower limit, and one determining the upper limit.

Method: `.startY(value)`

value: Lowest Y level where associated ore spawns.

Method: `.endY(value)`

value: Highest Y level where associated ore spawns.

Sometimes, you want to make sure ores do not spawn close to one another, even randomly.

Method: `.spread(value)`

value: Minimum distance from the nearest vein of the same ore.

And finally, you sometimes want an averaged distribution, with weight at a specific level, and everything outside of that to be more rare.

Method: `.weight(value)`

value: Y level where most of the ore spawns.

3.1.2 Alloying using Crucibles and Molds

Making alloys is impossible in the vanilla furnace, and not everyone wants to use some complex machinery mods to in order to have them.

To resolve this concern, JMOD has included code to allow a custom form of alloying known as "green sand casting." This is a real-world metalcasting technique where metal is melted down in a clay cup, called a crucible, and then poured into a mold made of a mix of wet sand and clay. The process provides a cheap way to make crude metal sculptures and machinery.

In the item creation code, we have included two new item classes: "ItemCrucible" and "ItemMold", which you can create using `addItem`. Note, however, that the crucible will always need to be smelted after being made; it is supposed to be a clay cup. Once you have these items, you will also want to create alloy recipes using the `addAlloy` command; as that will define what goes into the crucible, and what (and how much) will be made from the process.

Once made, the crucible will accept whatever recipes you defined with `addAlloy`, which you add by crafting the crucible with the one or two input materials on a crafting table.

Once you have the filled crucible, place it in any standard furnace, and it will smelt into a "hot crucible." This hot crucible contains the molten liquid of the new alloy material (as defined in `addAlloy`).

Once you put the hot crucible and the mold together on the crafting table, you are then able to take the output as defined by `addAlloy`.

While this does add the crucible and mold, note that it does not require any special machinery or power systems to work, nor does it affect how the furnace or crafting table normally work. As such, this should provide good early-game alloying or ore multiplication, even in a vanilla-themed modpack.

3.2 Tool Features Inspired by Tinker's Construct

Tinker's Construct has some useful features that does not exist for vanilla tools. JMOD attempts to add these features to vanilla tools and armor for gameplay when Tinker's Construct is not an option.

3.2.1 Non-Breaking Tools

If enabled, vanilla tools will not break. Unlike Tinker's Construct, it will not be obvious when a tool is broken (as it won't change its texture). However, there will be a tooltip notifying you that the tool is broken.

Command: `Global.preventToolBreaking(value)`

value: Do you want tools to remain after breaking? Boolean (true or false).

3.2.2 Crafting Grid Tool Repair

If enabled, the crafting grid can be used to restore endurance to a tool using its key component, defined in `addToolMaterial`. So, if you configured wood to use woodPlank as a repair material, than 1 wood plank would repair a wooden pickaxe by about 30%. Note that this operates at a discount; 3 wooden planks will only repair 90% of an axe or pickaxe.

Command: `Settings.craftingGridToolRepair(value)`

value: Do you want to use endurance repair in the crafting grid? Boolean (true or false).

3.2.3 Enhanced Anvil Repair

As mentioned, using the crafting grid to repair tools works at a discount, you get less repair value out of your materials than if you were to make a brand new one. With the anvil, however, the opposite is true; you repair with a bonus. Each material is worth more than its original value in making the tool, meaning repairs will last longer if you use the endurance repair with an anvil instead of a crafting grid. Not really useful with stone or wood, but with diamond tools, you can really get your money's worth!

Command: `Settings.craftingGridToolRepair(value)`

value: Do you want to use endurance repair in the anvil? Boolean (true or false).

3.3 Tooltip Features

These options provide some additional information in certain items' tooltips.

3.3.1 Tool Harvest Level Display

This provides a line in your tooltips that identifies a tool's harvest level; useful when dealing with expanded mining progressions using the `addToolMaterial` command.

Command: `Settings.showToolHarvestLevels(value)`

value: Do you want to see harvest levels in your tools' tooltips? Boolean (true or false).

3.3.2 Block Harvest Level Display

What `showToolHarvestLevels` does for tools, this one does for ore blocks. If you mouse over an ore block, the tooltip will show what harvest level that block is at.

Command: `Settings.showBlockHarvestLevels(value)`

value: Do you want to see harvest levels in the ore blocks' tooltips? Boolean (true or false).

3.3.3 Armor Values Display

Armor already has a little bar to show endurance, but if you want detailed information about a piece of armor you picked up off of that zombie you killed, this might be helpful. This option provides a tooltip for the armor to show how much endurance it has left.

Command: `Settings.showArmorValues(value)`

value: Do you want to see the endurance levels of your armor in their tooltips? Boolean (true or false).

3.4 Mod-Specific Workarounds

JMOD has had to add some custom code in order to work around the behavior of certain mobs that would otherwise prevent tweaks from working.

3.4.1 Forcing Sync to Behave

Sync tends to reset recipes for its items after all other mods have loaded. If you intended to gate Sync using altered recipes, this can work against you. JMOD includes a custom tweak to disable this behavior, and guarantee that any recipe changes for Sync sticks.

Command: `Sync.preventRecipeReload(value)`

value: Do you want to use endurance repair in the crafting grid? Boolean (true or false).

Conclusion

At this point, all the options have been listed, explained, and demonstrated. I hope you have found this documentation complete enough to make your very own jmods, and I look forward to seeing how modpack authors make use of JMOD.