
# Minecraft U Sequence 4: Java and Modding Minecraft

Learn the basics of Java, the most common programming language in the world, and use them to create your own modification of Minecraft. Add any item of your design to the game, expanding Minecraft in whatever direction you desire. At the end, you'll understand the syntax and applications of Java, and how they apply to Minecraft.

## Table of Contents



# Setup

## Downloads

To get started writing mods, you'll need a good development environment. Always try to keep your enviromnet organized, since it's what you'll have to use to write all your code.

- Download IntelliJ from their site: JetBrains IntelliJ IDEA Community Edition. Find the button towards the bottom of that page that says `Download Community`. This is a free version of their commercial IDE, and it has lots of the same features.
- We'll also need to download Forge, which is a modding API. In the section marked "Minecraft Versions" hover over "1.7" and select `1.7.10` from the drop down menu. Down and to the right under "Download Recommended", click `Src`. Wait for the countdown on the top right, and click skip. This should start downloading a file called `forge-1.7.10-10.13.#.####-src.zip`
- Also download the Java SDK. Download the correct version for your system from that page.

# Installation

1. Install the Java SDK by running the file you downloaded.
2. Run the IntelliJ installer, and follow its instructions. Make sure you have installed the Java SDK before installing IntelliJ, it makes things easier.
3. Make a new folder on the desktop and name it `Minecraft Dev` and drag the `forge` folder into it. The forge folder is a .zip file so we need to extract it (if you don't have an extraction tool we recommend 7-zip). Put the extracted folder into the Minecraft Dev folder.

Note: If you wish, you can rename the extracted `forge` folder to whatever you want your mod to be called, e.g. from `forge` to `DiamondCutter`. Keep the Forge zip file in case you want to start a fresh mod.

# Getting started for Windows

1. Hold shift and right click inside the folder you renamed. Click on `Open command window here`. Then type `gradlew setupDecompWorkspace --refresh-dependencies` and press `Enter`. This will download the Minecraft source code and decompile it so we can work with it to make our mod.
2. Once you get `BUILD SUCCESSFUL` type `gradlew idea` and press `Enter`.
3. After you get `BUILD SUCCESSFUL` again, run IntelliJ (the installation should have put a shortcut on your desktop). You most likely won't have any setting to import (click `ok`). Unless you have coding experience and have preferences, click `Skip All and Set Defaults`.
4. On the pop-up screen, select `Open` and navigate to the folder you renamed.
5. Find the file with the IntelliJ icon - it should be (Your_Folder_Name.ipr) and open it.

# For MAC

1. Double-click the zipped forge
2. Drag the forge folder that is created to your desktop
3. Download gradle: https://gradle.org/gradle-download/
   - click the Binary only distribution (second option)
4. Unzip it and put it in the forge folder on your desktop
5. Then press command+spacebar and type in "terminal" and press Enter (this will open a terminal window)
6. Type "cd Desktop" and press Enter

7. Type “cd forge” and press Tab (it should auto-complete to the folder name) and then press Enter
8. Type “gradle-2.5/bin/gradle setupDecompWorkspace –refresh-dependencies” and press Enter
9. After BUILD SUCCESSFUL shows up type: “gradle-2.5/bin/gradle idea”

# Troubleshooting:

The most common error is the build failing with a message saying “JAVA_HOME does not point to JDK”:

1. Click start
2. Right-click Computer (for Windows 8 open up file explorer and right-click “This PC”)
3. Click properties
4. On the left side click advanced system settings
5. At the bottom of the pop-up, click Environment Variables
6. Under system variables (second section of pop-up) click new
7. Name it JAVA_HOME
8. The value should point to your JDK 7 folder (something like “C:\Program Files\Java\jdk1.7.0_51”)
9. Click ok
10. Close and re-open command prompt and run the command again.

# Testing

1. On the top left, double click the folder icon (the name of your folder should be next to it).

2. Double-click the folders: src > main > java > com.example.examplemod

3. There should be an `ExampleMod` class (blue circle with the letter ‘c’ in it) and if you double-click it you should see the following code (you might have to change `preInit` to `init` and `FMLPreInitializationEvent` to `FMLInitializationEvent`):

```
package com.example.examplemod;

import net.minecraft.init.Blocks;
import cpw.mods.fml.common.Mod;
import cpw.mods.fml.common.Mod.EventHandler;
import cpw.mods.fml.common.event.FMLInitializationEvent;

@Mod(modid = ExampleMod.MODID, version = ExampleMod.VERSION)
public class ExampleMod
{
    public static final String MODID = "examplemod";
    public static final String VERSION = "1.0";

    @EventHandler
    public void init(FMLInitializationEvent event)
    {
        // some example code
        System.out.println("DIRT BLOCK >> "+Blocks.dirt.getUnlocalizedName());
    }
}
```

4. Left-click on `ExampleMod` and press `Shift + F6`. Change the name to something that you like and click `Refactor`. This will change all instances of `ExampleMod` to a mod name of your choice. For our examples, we use `CopperMod` since our mod adds copper to the game. We'll also have to change the `MODID` and `VERSION` to what we want. You can delete the example `println` statement if you wish.

## Publishing your mod

Running the command `gradlew build` or `gradle build` will package your mod into a .JAR file in the build/libs folder. You can then add it to Minecraft like any other mod.

# Brief introduction to classes and methods

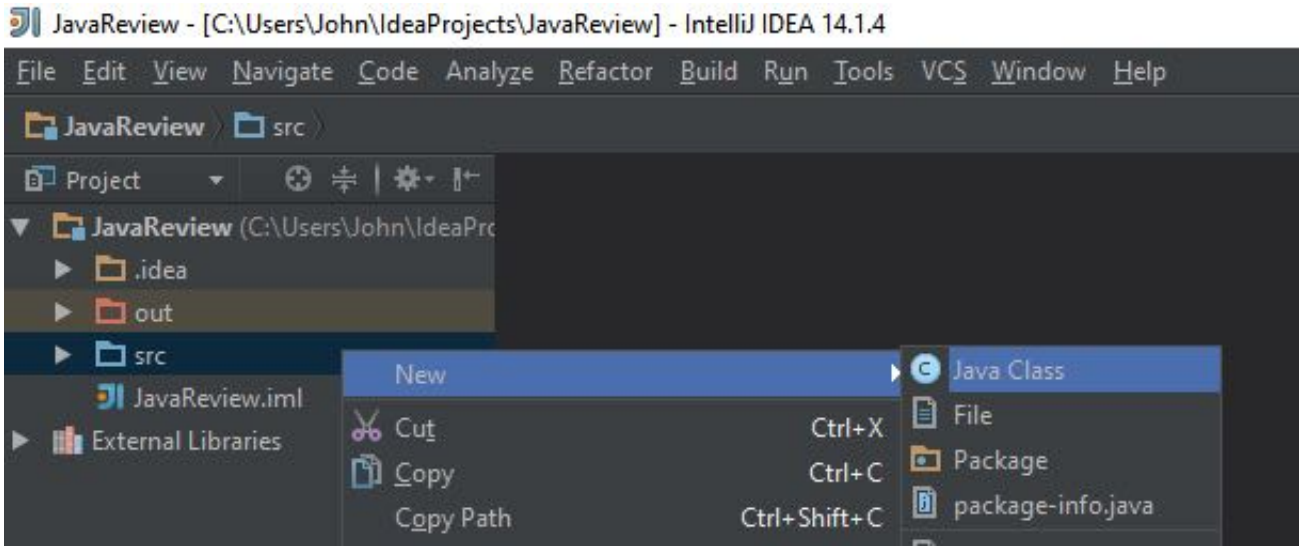
Before we dive into the Minecraft environment, you should understand what the words *class* and *method* mean in Java. At its most basic, a *class* represents a real-world object by using code. Classes can have both state and behavior. In other words, they both store information and do something with that information. For example, in a spaceship game you may have a spaceship class that holds the position and health of the player, and it may move around space and fire blasters.

## Navigating IntelliJ

Open up IntelliJ using either the shortcut on your desktop, or hitting the Windows key and searching `IntelliJ`. Choose `Create new project` and give it a name. We'll be calling ours `JavaReview`. Once IntelliJ opens, in the top left, you should see a folder icon called `JavaReview` or whatever you named the project. Double-click it and a sidebar should appear with two options: `JavaReview` and `External Libraries`. We'll talk about "External Libraries" later, double-click `JavaReview` for now. You should have a few options appear, including an `out` folder and a `src` folder.

## Creating a main class

Go ahead and right-click on the `src` folder and select `Create new class`.



Give it a name, we'll be calling ours `Main`, and click `OK`. For this class only, you'll need to add the method `main` (note it is lowercase `main` since Java is case sensitive). It should look like this:

```
public class Main {
    public static void main(String[] args) {

    }
}
```

# Writing a new method

If you remember from Sequence 3, a function is a block of code that performs some action. In Java we call functions *methods*. In this case, our method will calculate and print the sum of two integers.

```
public static int sum(int a, int b) {
    int total = a+b;
    return total;
}
```

This block defines a method called `sum` that takes two *parameters* (a and b). The first line is called the method signature and defines its return type `int`. In the second line, we define a new variable called `total` that is equal to `a+b`. The third line returns `total`. (Don't worry about the word `static` for now; it's necessary here but we will explain it later.)

Finally, we're going to *call* our `sum` function and pass it two *arguments*, in this case two integers. These *arguement* values fill in the *parameters* from the method definition. Then we'll assign the result to a variable `result` and print it. Your final code should look something like this:

```
public class Main {
    public static void main(String[] args) {
        int result = sum(2,3);
        System.out.println(result);
    }

    public static int sum(int a, int b) {
        int total = a+b;
        return total;
    }
}
```

Use `System.out.println()` whenver you want to print something to the screen. To execute this code, go to `Run -> Run 'Main'`. At the bottom it should print out the result, in this case 5. `Process finished with exit code 0` just means the program ran successfully.

# Defining a new class

At the beginning of this section, you read that classes have both state and behavior. So far, we've only created a class with behavior (a method that sums two numbers). Let's create a class that also contains information. Go ahead and create a new class called `Rectangle` just as you created the class called `Main`. Then let's give it some properties: length, width, perimeter, area.

```
public class Rectangle {
    public int length;
    public int width;
    public int perimeter, area;

    public Rectangle(int l, int w) {
        this.length = l;
        this.width = w;
        this.perimeter = length*2 + width*2;
        this.area = length * width;
    }

}
```

The only new part here, is the `public Rectangle(int l, int w)` method. This method is called the `Constructor` and will be used to *instantiate* our object in our next section.

## Creating an object from a class

Now that a `Rectangle` class has been defined, we can *instantiate* a `Rectangle` *object* in our main. To *instantiate* an object means to create an *instance* from a class. Imagine that our class is the blueprint for constructing a particular model of car, and that objects are the actual cars that are being driven. All of these cars behave the same way and contain the same types of information (think speed, color, license plate). Now that we have our Rectangle blueprint, we can make a couple Rectangle objects with different sizes.

In your `Main` class, in the `main` method, create a few Rectangle objects by calling the Rectangle constructor, passing in a width and a height. We'll make the first one a 2x3 Rectangle and the second a 3x4 Rectangle. We'll also want to print out the area and perimeter. Your code should look like this:

```
public static void main(String[] args) {
    Rectangle rect1 = new Rectangle(2,3);
    Rectangle rect2 = new Rectangle(3,4);

    System.out.println("Area of rect1: " + rect1.area);
    System.out.println("Perimeter of rect1: " + rect1.perimeter);
    System.out.println("Area of rect2: " + rect2.area);
    System.out.println("Perimeter of rect2: " + rect2.perimeter);
}
```

# Making basic blocks

> Anytime you copy Java code in this section, be sure to match ALL capitalization, punctuation, and spacing.

In the Minecraft source code, just about everything is represented as a class. There's a class for a diamond pickaxe, for an iron ore, and all other blocks and items. These classes tell what a block (or item) should look like, how it should behave, as well as where it spawns or how it can be crafted. By creating our own classes, we can add our own blocks and items to the game. Let's say that we wanted to make a new type of block. Minecraft already has a `Block` class that defines what a block is in the game (all blocks can be broken and have a texture, for example). We can *extend* the existing `Block` class and make our own new block. It will have all the normal properties of a block but we can set our own texture, hardness, and sound.

For this lesson, we're going to create a new resource in Minecraft: copper. We'll have to make copper ore, copper ingots, copper tools, and all the other items associated with it (think of the tools and other items made from iron or diamond). First, open up the IntelliJ project that was created when you ran the gradle builds. Then, you should create a new class called `CopperBlock` by right-clicking on the package and choosing *Create new class*. Name it `CopperBlock` and press OK. The file that opens up will have code almost matching what I've written below. You should add the lines with comments (use `//` to make a comment in Java) after them so that it matches exactly. Note: if you add `extends Block` first, you can press `Alt-Enter` on top of the error and IntelliJ will offer to add the import for you. Be sure that any imports are of the form `net.minecraft.something`.

```
package com.example.coppermod;

import net.minecraft.block.Block; //Add this line

public class CopperBlock extends Block //Add the second half of this line
{

}
```

The keyword `class` means that we are creating a new class. By extending the `Block` class, our `CopperBlock` class will inherit all of the attributes of the existing `Block` class. We call `Block` the *parent class* of `CopperBlock`, while `CopperBlock` is the *child class* of `Block`.

An error will come up regarding a missing *constructor*. A constructor is a function that runs whenever a new block is created in the game. The following constructor should go inside the braces of the `CopperBlock` class. If your cursor is over `CopperBlock`, you should be able to press Alt-Enter and select `Add missing constructor`. Then change the name of the `Material` parameter to `mat` and fill in the rest of the constructor.

```
package com.example.coppermod;

import net.minecraft.block.Block;
import net.minecraft.block.material.Material;
import net.minecraft.creativetab.CreativeTabs;

public class CopperBlock extends Block
{
    protected CopperBlock(Material mat) {
        super(mat);
        this.setBlockName("copper_block");
        this.setHardness(5.0F);
        this.setStepSound(Block.soundTypeMetal);
        this.setCreativeTab(CreativeTabs.tabBlock);
        this.setBlockTextureName("coppermod:copper_block");
        this.setHarvestLevel("pickaxe", 2);
    }
}
```

`super` calls the constructor of `CopperBlock`'s parent class, `Block`. It's very important, but we mainly want to focus on the rest of the functions since they allow more customization! Each of them defines an attribute of our `CopperBlock` block, most of which should be obvious from the name. For example, `setStepSound` determines which sound the block will make when placed. `setHarvestLevel` determines what type and level of tool is required to successfully mine it (the number `2` means iron tools are required). The keyword `this` means that the function is part of the class whose constructor you're currently in, `CopperBlock` in this case (don't worry about this too much, either).

However, simply making a new class in our project is not enough. To actually add our block into the game, we need to register it with Minecraft Forge. Open the `CopperMod` class from the left side of the screen (it might still be called `ExampleMod` initially; just rename it to `CopperMod` if so). Add the variable declaration line and the `registerBlock` line shown below. The second argument of `registerBlock` sets up the name of the block as "*modid_blockname*" and lets us use the same code to create a standardized naming system for all of our blocks. In addition, by using our *MODID* in the names of our blocks, we can make sure there won't be any overlaps with any other mods we may want to add.

```
public static CopperBlock copperBlock;  //static variable declaration

@EventHandler
public void init(FMLInitializationEvent event)
{
    copperBlock = new CopperBlock(Material.iron);
    GameRegistry.registerBlock(copperBlock, MODID + "_" + copperBlock.getUnlocalizedName());
}
```

This block of code registers our newly-created block with the game. Again, `MODID + "_" + copperBlock.getUnlocalizedName()` creates a unique name based on our `MODID`, so that our blocks (or items) will not conflict with other mods.

So to recap, our `CopperMod` and `CopperBlock` classes should look as follows.

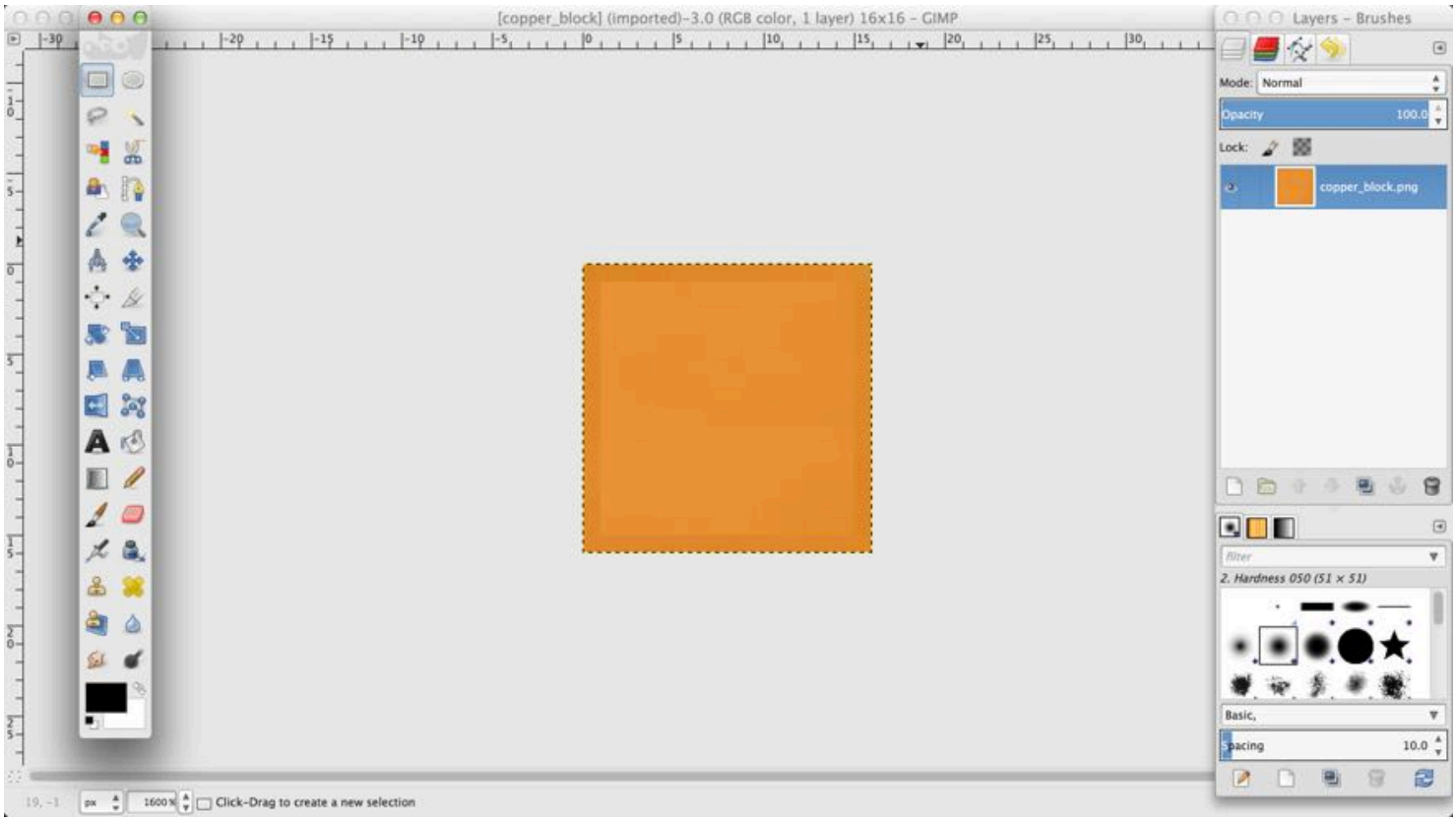
```
package com.example.coppermod;

import cpw.mods.fml.common.Mod;
import cpw.mods.fml.common.Mod.EventHandler;
import cpw.mods.fml.common.event.FMLInitializationEvent;
import cpw.mods.fml.common.registry.GameRegistry;
import net.minecraft.block.material.Material;

@Mod(modid = CopperMod.MODID, version = CopperMod.VERSION)
public class CopperMod
{
    public static final String MODID = "coppermod";
    public static final String VERSION = "1.0";

    public static CopperBlock copperBlock;

    @EventHandler
    public void init(FMLInitializationEvent event)
    {
        copperBlock = new CopperBlock(Material.iron);
        GameRegistry.registerBlock(copperBlock, MODID + "_" + copperBlock.getUnlocalizedName());
    }
}
```

```
package com.example.coppermod;

import net.minecraft.block.Block;
import net.minecraft.block.material.Material;
import net.minecraft.creativetab.CreativeTabs;

public class CopperBlock extends Block
{
    protected CopperBlock(Material mat) {
        super(mat);
        this.setBlockName("copper_block");
        this.setHardness(5.0F);
        this.setStepSound(Block.soundTypeMetal);
        this.setCreativeTab(CreativeTabs.tabBlock);
        this.setBlockTextureName("coppermod:copper_block");
        this.setHarvestLevel("pickaxe", 2);
    }
}
```

To actually launch our modded Minecraft, run the project by clicking on the green arrow at the top of the IDE. Make a new creative world and try placing your block on the ground. It should be under the normal `Blocks` tab at the very bottom and will be called `tile.copper_block.name`. The coloring should be a purple and black checkerboard, the default color scheme when a texture is not specified. Even though we said the texture file's name should be `copper_block` in the code, Minecraft can't find the texture file because we haven't made one!

## Adding a texture to a block

To add our texture to our block, we first need to create the folder that will hold our textures. The full path is `src/main/resources/assets/coppermod/textures/blocks`. Start at the `resources` folder (it should already exist) and create one folder after another until you get down to the final one, `blocks`. Open up Paint (or another image-editing program) and create a new empty canvas with a square resolution. Most Minecraft textures are 16x16 but you could also try 32x32 or 64x64. Take a few minutes and make your own texture!

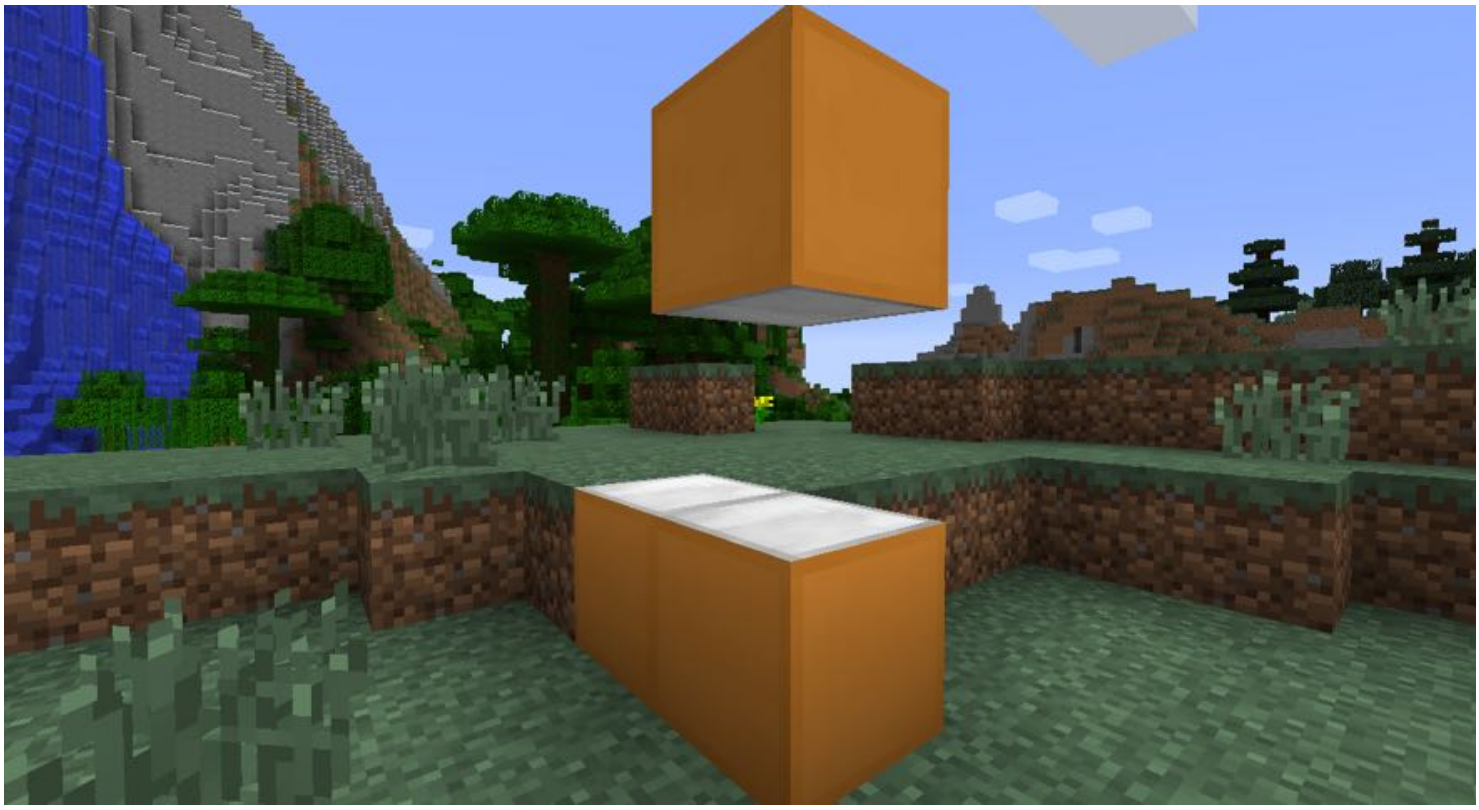



For right now, the block will have the same texture on all six sides like cobblestone or obsidian. Name the texture `copper_block` and save it as a `.png` in the `blocks` folder. After your texture has been saved, run Minecraft. Now check out the texture of your block!


## Multi-sided textures

What about blocks like grass that have different textures on different sides? The following code in the `CopperBlock` class registers three different textures and tells the program which textures belong on the top, sides, and bottom. If we want, we could even make the side textures different (like a furnace or dispenser). You will have to create three texture files matching the names that are being registered. I've made it so that my block has a silver-colored top and bottom. You can add additional `if` statements if you would like the other sides to have different colors as well. The `meta` parameters can be used to alter which textures that are returned (such as how logs face differently based on their angle of placement).

```
//Creating our icon variables
private IIcon topIcon;
private IIcon sideIcon;
private IIcon botIcon;

//This function registers our textures to their specific icons
@Override
public void registerBlockIcons(IIconRegister ir)
{
    this.topIcon = ir.registerIcon("coppermod:copper_block_top");
    this.sideIcon = ir.registerIcon("coppermod:copper_block_side");
    this.botIcon = ir.registerIcon("coppermod:copper_block_bottom");
}

//This function tells which icons to use for which sides of the block
@Override
public IIcon getIcon(int side, int meta) //side = the side of the block
{
    if (side == 0) //Bottom
        return botIcon;

    else if (side == 1) //Top
        return topIcon;

    else //all other sides, numbers 2,3,4,5
        return sideIcon;
}
```



# Making basic items

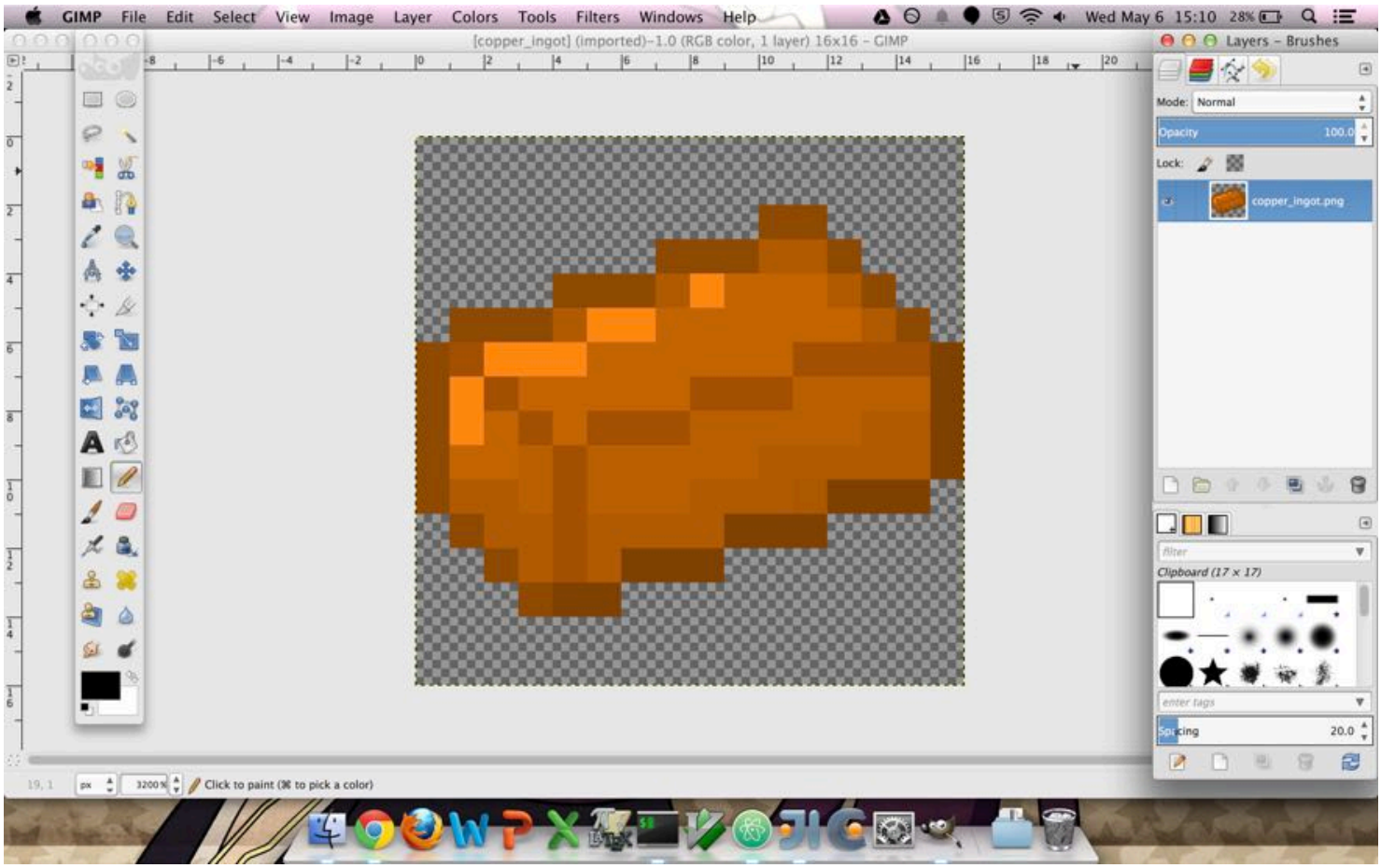
Making an item is very similar to making a new block, except that we will be extending the `Item` class rather than the `Block` class.

```
package com.example.coppermod;

import net.minecraft.item.Item;

public class CopperIngot extends Item {

}
```

Just as we did with our `CopperBlock` class, we can also add a constructor and give some properties to our `CopperIngot`.

```
public CopperIngot()
{
    this.setUnlocalizedName("copper_ingot");
    this.setCreativeTab(CreativeTabs.tabMaterials);
    this.setTextureName("coppermod:copper_ingot");
}
```

One important note regarding textures is that item textures should have transparent backgrounds, or there will be a white square around the item in the game. Transparency backgrounds are indicated by a grey and white checkerboard on the background of the image file.




We'll also need to register our item with the game using the `registerItem` function. You should create a static `copperIngot` variable in `CopperMod` class just as we did with our static `copperBlock` variable.

```
copperIngot = new CopperIngot();    //initializing the variable you should declare in the class
GameRegistry.registerItem(copperIngot, MODID + "_" + copperBlock.getUnlocalizedName());
```

# On-click effects

There are two methods that are called when the player right-clicks while holding an item. `onItemUse` is called when the player is targeting a block in range (ie. the block has the black wireframe around it), while `onItemRightClick` is called regardless of what the player is targeting (ie. even if they are looking at the sky).

`onItemUse` gives us several parameters that we can use to make *something* happen when an item is used.

```
@Override
public boolean onItemUse(ItemStack itemstack, EntityPlayer player, World world, int p_77648_4_, int p
int p_77648_6_, int p_77648_7_, float p_77648_8_, float p_77648_9_, float p_77648_10_)
{
    return false;
}
```

```
`onItemRightClick` is a bit more general and primitive. There are fewer arguments given to us,
  @Override
public ItemStack onItemRightClick(ItemStack itemstack, World world, EntityPlayer player)
{
    return itemstack;
}
```

# Setting names

The names that we've given our new blocks and items so far are all hard-coded into our mod. They are all lowercase and use underscores to separate words, but they don't look like the typical names of items in Minecraft. Also, what if we want people in other countries who speak different languages to play our mod? We can use what are called *language packs* to give our items language-specific names that will actually show up in the game. The packs also replace the cumbersome "package.item.item"-type names with real names such as "Iron Ingot" or "Dirt".

## File extensions

Before we create our language packs, make sure you have "hide file extensions for known file types" disabled. To check if its disabled, go look at your textures. If they show up as "*name*.png" you're good. Otherwise, follow these instructions.

**Windows**

1. Start -> Control Panel -> Appearance and Personalization -> Folder Options
2. Click on "View" tab
3. Click "Advanced settings"
4. Uncheck the box next to "Hide extensions for known file types" then click "OK"

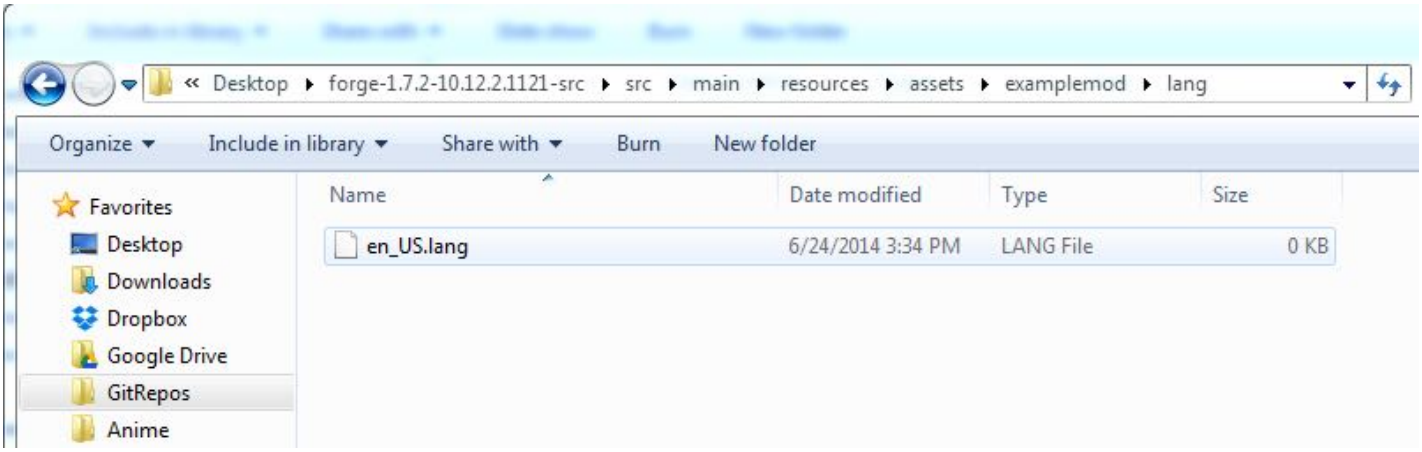
**Mac**

1. Select Finder -> Preferences -> Advanced
2. Select "Show all filename extensions"

## Making the language pack

1. Create a folder called "lang" in the "assets/examplemod" folder.

2. Create a new text file called “en_US.lang” in the folder.



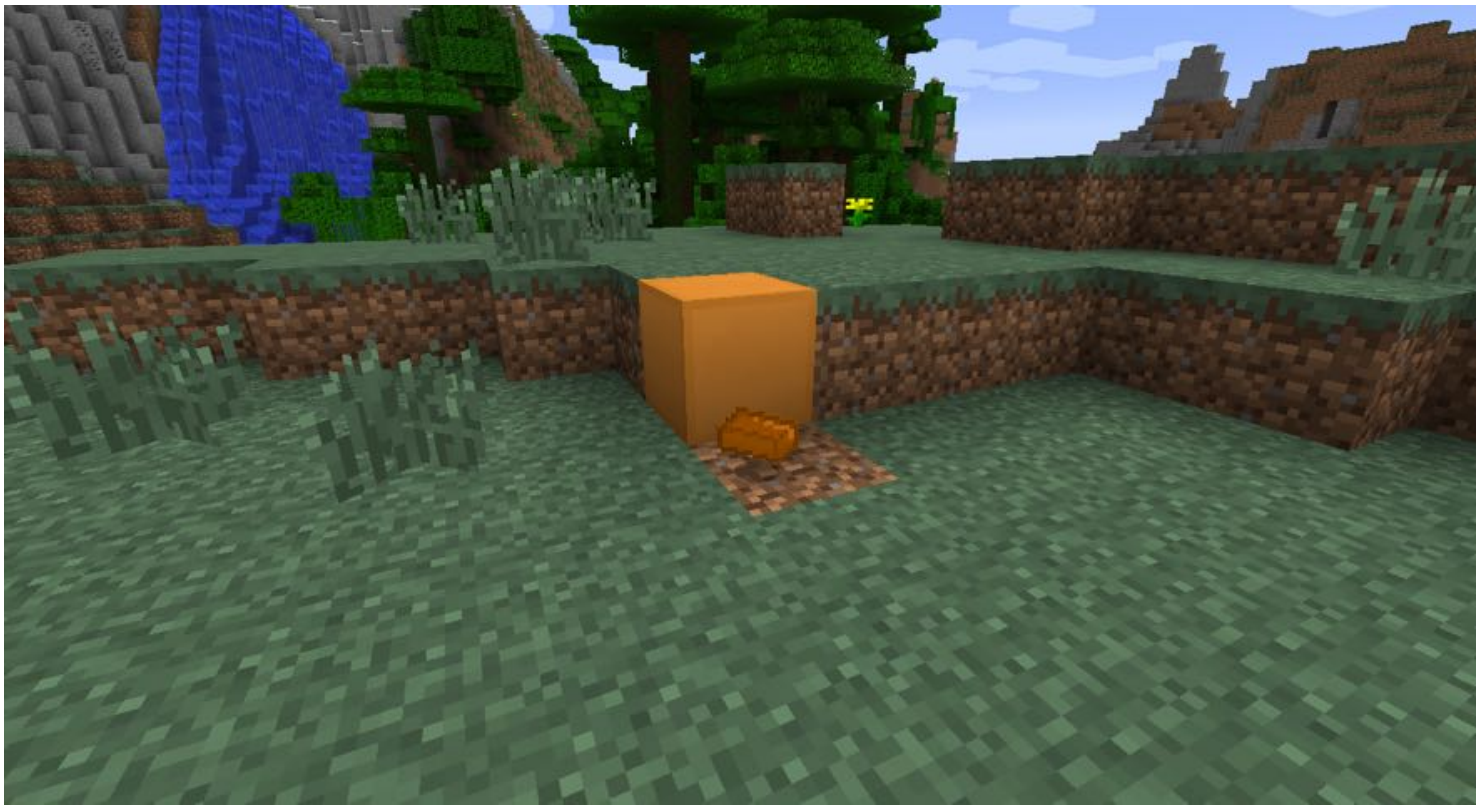

3. Right-click on the file and choose to edit it. The default entry is something similar to: `category.blockName.name=Item Name`. The name that appeared for our initial block, `tile.copper_block.name` is what we would use in our case. So my line would be `tile.copper_block.name=Copper Block`.

4. Save the file and run Minecraft. Your name should now appear in-game.


# Adding custom drops and recipes

Right now, our `CopperBlock` only drops itself when broken, like wood does. But what about blocks suchas glowstone? Glowstone drops an item (specifically glowstone dust) when broken. We can modify our existing block easily to drop an item (or multiple items) when broken. Add the following function declaration to the `CopperBlock` class. Launch the game, and try breaking your block. Our copper block will now drop an ingot when mined.

```
@Override
public Item getItemDropped(int metadata, Random random, int fortune)
{
    return CopperMod.copperIngot;   //this is why we use static variables
}
```



What if we wanted to make our block drop several ingots when broken? It generally takes 9 ingots to make a block, so let's use the `quantityDropped` method to tell our `CopperBlock` to drop 9 ingots when it is mined. The following method declaration should go in your `CopperBlock` class.

```
public int quantityDropped(Random rand)
{
    return 9;
}
```

`rand` is a `Random` object that we could use to add randomness to our drops. For example, we could specify that we want a random number of ingots between 2 and 5 to drop when a block is broken by using the `nextInt` method to get a random number.

```
public int quantityDropped(Random rand)
{
    return 2 + rand.nextInt(3); //random between 2 and 5
}
```

# Making new crafting recipes

> The function calls that create new recipes will go in your main mod class

Now we need to make our new blocks and items useful by creating some recipes! There are two types of recipes we can create: shapeless and shaped. Shapeless recipes (such as making wooden planks from wood) don't require the items to be in any specific orientation to work. Shaped recipes (such as making a pickaxe or shovel) require blocks to be in specific locations for the recipe to function.

We simply use the `addShaplessRecipe` and `addShapedRecipe` function calls to register our recipes with the game. The calls will go in your main mod class, and you will need to import the `ItemStack` and `Items` classes to use them (IntelliJ will prompt you for them when you use them).

## Shapeless recipes

First, let's make a shapeless recipe.

```
GameRegistry.addShapelessRecipe(new ItemStack(Items.diamond, 64), new ItemStack(Blocks.dirt));
```

This recipe simply trades in a stack of 64 dirt blocks for 64 diamonds. The resulting item is the first argument in the function call, and any input items are the following arguments.


To add more items, simply add more ItemStacks within the parentheses.

```
GameRegistry.addShapelessRecipe(new ItemStack(Items.diamond, 64), new ItemStack(Blocks.dirt),
    new ItemStack(Blocks.dirt), new ItemStack(Blocks.dirt));
```


## Shaped recipes

Next, let's make a shaped recipe. Shaped recipes group the rows of items using letters to represent the type of block. For example, the recipe for TNT would be: `("xyx", "yxy", "xyx", 'x', new ItemStack(Materials.sulphur), 'y', new ItemStack(Blocks.sand))` Pay attention to the double-quotes for the letters representing the recipe shape and the single-quotes for the letters representing the items in the recipe.

> Note: when we leave out the size of the ItemStack, the game will assume a size of 1.

An example of a shaped recipe. You use three strings to represent the three rows of the crafting table and define which letters relate to which type of item. Spaces are used to represent empty crafting slots.

```
GameRegistry.addShapedRecipe(new ItemStack(Items.diamond), "xxx", "x x", "xxx", 'x',
    new ItemStack(Items.coal));
```


## Smelting recipes

Smelting recipes are very similar and only require a slightly different function call.

```
GameRegistry.addSmelting(Blocks.stone, new ItemStack(Blocks.stonebrick), 0.1f);
```

This time, the input item is on the left while the output is on the right. The number at the end specifies how much experience the player receives from the smelting.


On a side note, the *damage values* of items often hold extra information (also called data) about the block. For example, all the colors of wool are actually the same type of block. They're rendered differently based on the value of their damage value. We can use this data value in our recipes to alter what kinds of wool, clay, or wood are required.

```
// What do you think this smelting recipe does?
ItemStack woolStackBlack = new ItemStack(Blocks.wool);
ItemStack woolStackOrange = new ItemStack(Blocks.wool);
woolStackBlack.setItemDamage(15);
woolStackOrange.setItemDamage(1);
GameRegistry.addSmelting(woolStackBlack, woolStackOrange, 0.1f);
```


# Generate your ore

```
package utd.atvaccaro.coppermod.worldgen;

import utd.atvaccaro.coppermod.CopperMod;
import cpw.mods.fml.common.IWorldGenerator;
import net.minecraft.block.Block;
import net.minecraft.init.Blocks;
import net.minecraft.world.World;
import net.minecraft.world.chunk.IChunkProvider;
import net.minecraft.world.gen.feature.WorldGenMinable;

import java.util.Random;

/**
 * Created by atvaccaro on 8/20/14.
 */
public class OreManager implements IWorldGenerator {
    @Override
    public void generate(Random random, int chunkX, int chunkZ, World world,
        IChunkProvider chunkGenerator, IChunkProvider chunkProvider)
    {
        switch(world.provider.dimensionId)
        {
            case -1: generateNether(world, random, chunkX*16, chunkZ*16);
                break;

            case 0: generateSurface(world, random, chunkX*16, chunkZ*16);
                break;
```

```
        case 1: generateEnd(world, random, chunkX*16, chunkZ*16);
            break;
    }
}

private void generateEnd(World world, Random random, int x, int z)
{
    // add a call here to spawn in the End
}

private void generateNether(World world, Random random, int x, int z)
{
    // add a call here to spawn in the Nether
}

private void generateSurface(World world, Random random, int x, int z)
{
    this.addOreSpawn(CopperMod.copperOre, world, random, x, z, 16, 16, 16, 128, 15, 160);
}

/**
 * Adds an Ore Spawn to Minecraft. Simply register all Ores to spawn with this
  method in your Generation method in your IWorldGeneration extending Class
 *
 * @param block : The Block to spawn
 * @param world : The World to spawn in
 * @param random : A Random object for retrieving random positions within the
   world to spawn the Block
 * @param blockXPos : An int for passing the X-Coordinate for the Generation
   method
 * @param blockZPos : An int for passing the Z-Coordinate for the Generation
   method
 * @param maxX : An int for setting the maximum X-Coordinate values for spawning
   on the X-Axis on a Per-Chunk basis
 * @param maxZ : An int for setting the maximum Z-Coordinate values for spawning
   on the Z-Axis on a Per-Chunk basis
 * @param maxVeinSize : An int for setting the maximum size of a vein
 * @param chancesToSpawn : An int for the Number of chances available for the
   Block to spawn per-chunk
 * @param minY : An int for the minimum Y-Coordinate height at which this block
   may spawn
 * @param maxY : An int for the maximum Y-Coordinate height at which this block
   may spawn
 **/
public void addOreSpawn(Block block, World world, Random random, int blockXPos,
int blockZPos, int maxX, int maxZ, int maxVeinSize, int chancesToSpawn, int minY, int maxY)
{
    assert maxY > minY : "The maximum Y must be greater than the Minimum Y";
    assert maxX > 0 && maxX <= 16 :
    "addOreSpawn: The Maximum X must be greater than 0 and less than 16";
    assert minY > 0 : "addOreSpawn: The Minimum Y must be greater than 0";
    assert maxY < 256 && maxY > 0 :
    "addOreSpawn: The Maximum Y must be less than 256 but greater than 0";
    assert maxZ > 0 && maxZ <= 16 :
    "addOreSpawn: The Maximum Z must be greater than 0 and less than 16";

    int diffBtwnMinMaxY = maxY - minY;
    for (int i = 0; i < chancesToSpawn; i++)
    {
        int posX = blockXPos + random.nextInt(maxX);
```

```
            int posY = minY + random.nextInt(diffBtwnMinMaxY);
            int posZ = blockZPos + random.nextInt(maxZ);
            (new WorldGenMinable(block, maxVeinSize, Blocks.dirt)).generate(world,
             random, posX, posY, posZ);
            //third argument in WorldGenMinable determines replaced block; leave out
            //to use Blocks.stone by default
        }
    }
}
```

```
OreManager om = new OreManager();
GameRegistry.registerWorldGenerator(om, 0); //Integer determines when generation code
 runs (0 = normal)
//Use higher numbers to run later
```

# Creating foods

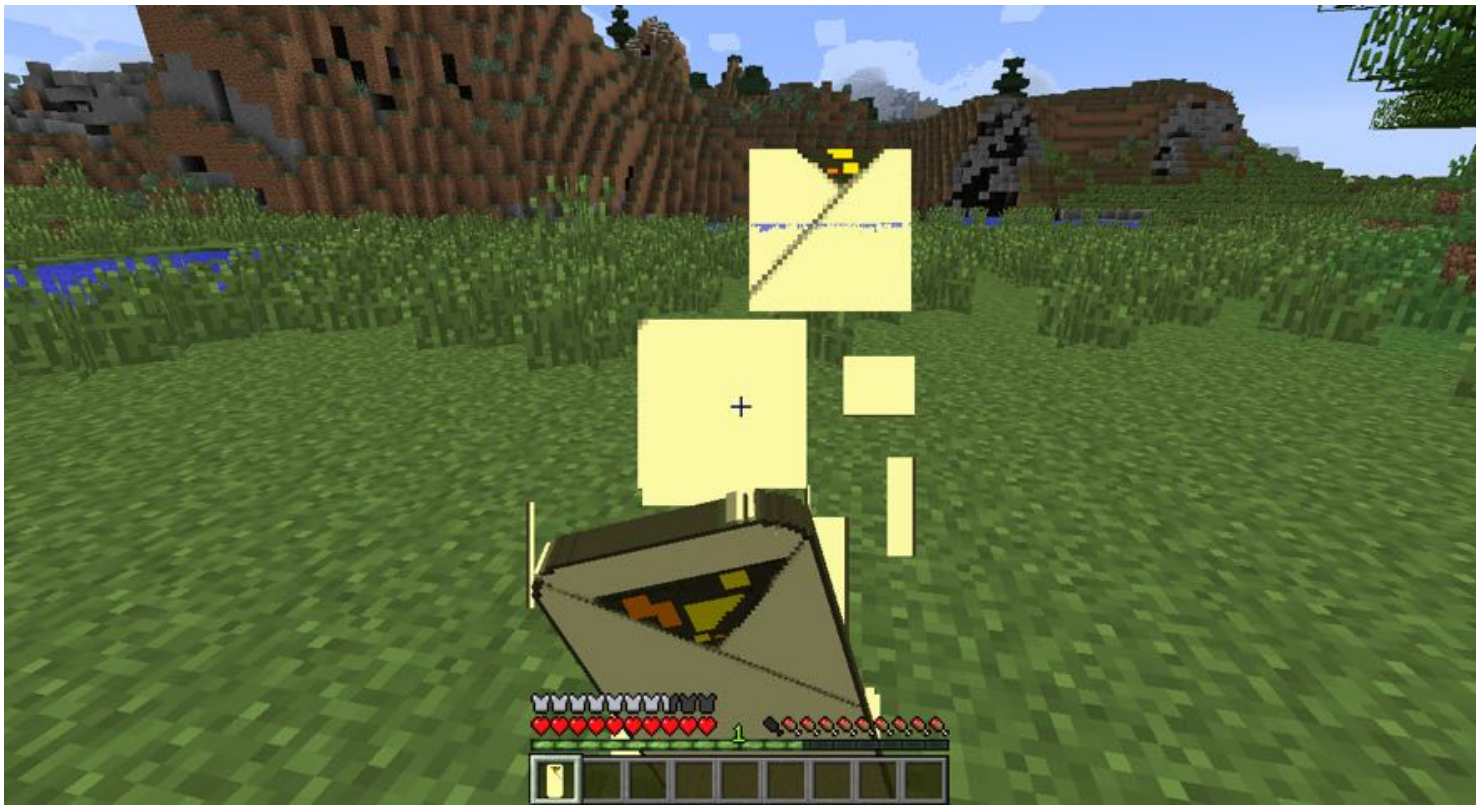
> In this section, you will learn how to create new foods and add potion effects to them

Any Minecraft player who wants to survive will need to learn about food and food resources. The foods in vanilla Minecraft tend to be straight-forward. You might cook raw beef into steak, or make cookies from cocoa and wheat. Creating your own food classes will let you create more complicated foods and even add potion effects when you eat something!

The `ItemFood` class is the parent class for all of the food items in Minecraft. Anything descended from `ItemFood` can be eaten and has some hunger or saturation values (the saturation value determines how long you can move until hunger starts ticking down again). For example, I wrote an `ItemSteakTaco` class to add tacos. I extended the `ItemFood` class and created a constructor that calls the superclass constructor.

```
package com.example.coppermod;

import net.minecraft.item.ItemFood;

public class ItemSteakTaco extends ItemFood {
    public ItemSteakTaco(int foodValue, float satmodifier, boolean isWolfsFavoriteMeat) {
        super(foodValue, satmodifier, isWolfsFavoriteMeat);
    }
}
```

The `foodValue` variable determines how much health is restored when the food is eaten, and the `satmodifier` variable determines how long until the player becomes hungry again. To give an idea of these values, `foodValue` is set to 8 and `satmodifier` is set to 0.8 for cooked porkchops. The boolean `isWolfsFavoriteMeat` simply tells whether or not the food is appealing to wolves (only true for some meats by default). After writing the class and creating my texture, I register the new item with the game (simultaneously giving the heal and saturation values).

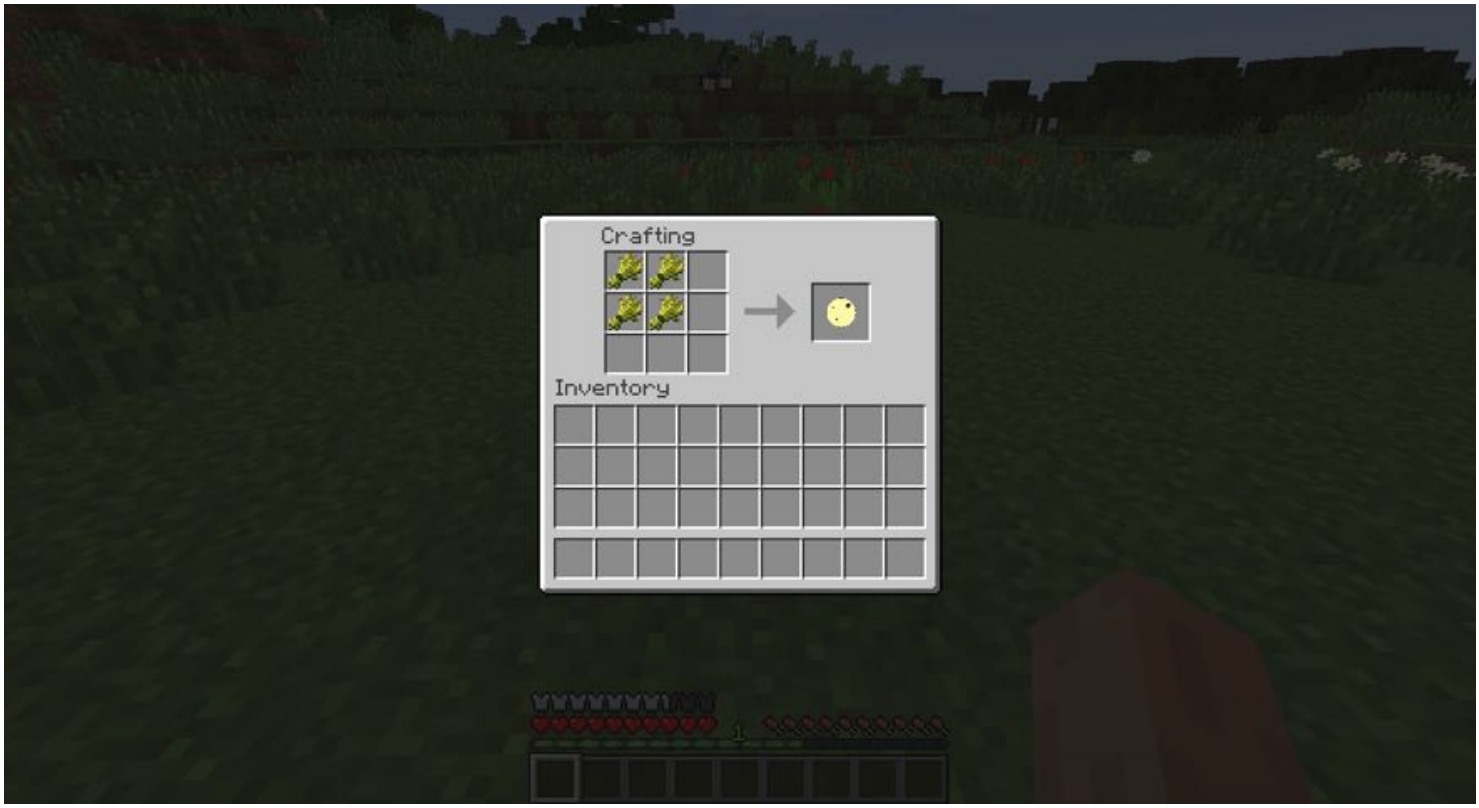


Tacos don't just appear in the wild, however. We also need to create the ingredients as well as recipes in order to make them. For this example, I will make an `ItemTortilla` class and then create the recipes necessary to make the tortilla and then the taco. `ItemTortilla` is also a food, but will not have as good healing and saturation properties.

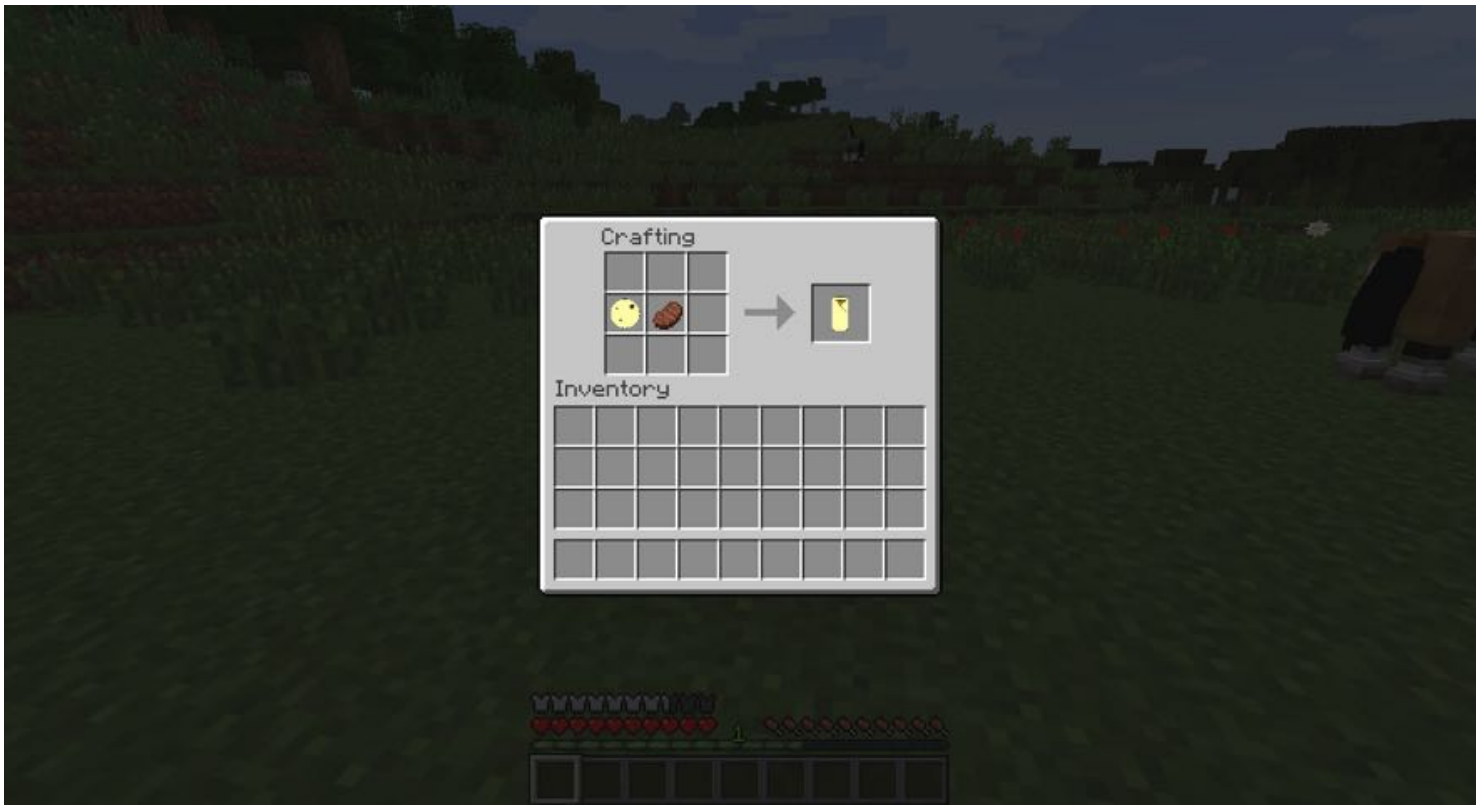
```
package com.example.coppermod.item;

import net.minecraft.creativetab.CreativeTabs;
import net.minecraft.item.ItemFood;

public class ItemTortilla extends ItemFood {
    public ItemTortilla(int foodValue, float satmodifier, boolean isWolfsFavoriteMeat) {
        super(foodValue, satmodifier, isWolfsFavoriteMeat);

        this.setCreativeTab(CreativeTabs.tabFood);
        this.setUnlocalizedName("itemTortilla");
        this.setTextureName(("coppermod:tortilla"));
    }
}
```

Making the crafting recipe for a food is just like for any other item. A recipe is required to make the tortilla from wheat, and then make the taco from tortilla and steak.

```
GameRegistry.addShapedRecipe(new ItemStack(itemTortilla), "xx ", "xx ", "   ", 'x', Items.wheat);
```



```
GameRegistry.addShapelessRecipe(new ItemStack(itemSteakTaco), itemTortilla, Items.cooked_beef);
```




Finally, we can quickly and easily add potion effects when our foods are eaten. If we override the `onFoodEaten` method and call the `setPotionEffect` method, we can give the player any potion effect for any duration. Look at the source code of the `Potion` class to know which ID numbers correspond to which potions.

```
protected void onFoodEaten(ItemStack itemstack, World world, EntityPlayer player)
{
    player.addPotionEffect(new PotionEffect(1, 20, 1));  //potion id, duration, amplifier
}
```

Now that you know the basics of creating foods, pick a couple of fairly complicated foods that require multiple steps and ingredients to make in real life. Some suggestions could be apple pie or a sandwich. Then, recreate the steps in Minecraft! Make new classes that extend `ItemFood` to make the ingredients and recipes (both crafting and smelting!) to simulate the cooking process.

# Creating crops

```
package utd.atvaccaro.coppermod.block;

import utd.atvaccaro.coppermod.CopperMod;
import cpw.mods.fml.relauncher.Side;
```

```java
import cpw.mods.fml.relauncher.SideOnly;
import net.minecraft.block.BlockCrops;
import net.minecraft.client.renderer.texture.IIconRegister;
import net.minecraft.item.Item;
import net.minecraft.item.ItemStack;
import net.minecraft.util.IIcon;
import net.minecraft.world.World;

import java.util.ArrayList;
import java.util.Random;

public class BlockBlueberry extends BlockCrops
{
    @SideOnly(Side.CLIENT)
    protected IIcon[] iIcon;

    public BlockBlueberry()
    {
        // Basic block setup
        setBlockName("blueberries");
        setBlockTextureName("coppermod:blueberries_stage_0");
    }

    /**
     * Returns the quantity of items to drop on block destruction.
     */
    @Override
    public int quantityDropped(int parMetadata, int parFortune, Random parRand)
    {
        return (parMetadata/2);
    }

    @Override
    public Item getItemDropped(int parMetadata, Random parRand, int parFortune)
    {
        // DEBUG
        return (CopperMod.blueberry);
    }

    @Override
    @SideOnly(Side.CLIENT)
    public IIcon getIcon(int side, int stage) { return iIcon[stage]; }

    @Override
    @SideOnly(Side.CLIENT)
    public void registerBlockIcons(IIconRegister parIIconRegister)
    {
        iIcon = new IIcon[8];
        // seems that crops like to have 8 growth icons, but okay to repeat actual texture if you wan
        // to make generic should loop to maxGrowthStage
        iIcon[0] = parIIconRegister.registerIcon("coppermod:blueberry_stage_0");
        iIcon[1] = parIIconRegister.registerIcon("coppermod:blueberry_stage_0");
        iIcon[2] = parIIconRegister.registerIcon("coppermod:blueberry_stage_1");
        iIcon[3] = parIIconRegister.registerIcon("coppermod:blueberry_stage_1");
        iIcon[4] = parIIconRegister.registerIcon("coppermod:blueberry_stage_2");
        iIcon[5] = parIIconRegister.registerIcon("coppermod:blueberry_stage_2");
        iIcon[6] = parIIconRegister.registerIcon("coppermod:blueberry_stage_2");
        iIcon[7] = parIIconRegister.registerIcon("coppermod:blueberry_stage_3");
    }

    @Override
```

```
    public ArrayList<ItemStack> getDrops(World w, int x, int y, int z, int meta, int fortune) {
        ArrayList<ItemStack> drops = new ArrayList<ItemStack>();
        drops.add(new ItemStack(CopperMod.blueberry));

        if (meta >= 7)
        {
            for (int i = 0; i < 3+fortune; ++i)
            {
                if(w.rand.nextInt(10) <= meta)
                {
                    drops.add(new ItemStack(CopperMod.blueberry));
                }
            }
        }
        return drops;
    }   //end getDrops
}
```

```
package utd.atvaccaro.coppermod.item;

import utd.atvaccaro.coppermod.CopperMod;
import net.minecraft.creativetab.CreativeTabs;
import net.minecraft.init.Blocks;
import net.minecraft.item.ItemSeedFood;

public class ItemBlueberry extends ItemSeedFood
{

    public ItemBlueberry()
    {
        super(1, 0.3F, CopperMod.blueberryBlock, Blocks.farmland);
        setUnlocalizedName("blueberry");
        setTextureName("coppermod:blueberry");
        setCreativeTab(CreativeTabs.tabFood);
    }
}
```

```
//CROPS
blueberryBlock = new BlockBlueberry();
GameRegistry.registerBlock(blueberryBlock, blueberryBlock.getUnlocalizedName());
blueberry = new ItemBlueberry();
GameRegistry.registerItem(blueberry, blueberry.getUnlocalizedName());
```

# Making new tools

## Custom ToolMaterial

First, we should go ahead and make our own `ToolMaterial`. A `ToolMaterial` is an object that holds all of the information about a material that can be made into tools. Minecraft already has them for materials such as wood and iron, but we need to make one for copper. We use the `EnumHelper` class to actually create our new material variable (this variable declaration goes in our main mod class).

```
public static final Item.ToolMaterial COPPER = EnumHelper.addToolMaterial("copper_tool", 2,
            150, 5.0F, 7.0F, 21); //Harvest level, durability, block damage, entity damage, enchantab
```

Note that the `addToolMaterial` method takes a total of six arguments. The first argument is a string that will be registered as the name of our material. The other five arguments are numbers that determine attributes of the material:

- Harvest level (0-3): Determines what types of blocks the tools can break properly (0 is wood, 3 is diamond).
- Durability (Diamond = 1562): The number of uses before the tool breaks.
- Block and Entity damages: Determine how much damage the tool material does to blocks and entities, respectively (these arguments are floats).
- Enchantability (Gold Armor = 25): How well a tool can be enchanted (higher number means better enchantments at lower levels).

Since we've made `COPPER` a static variable, we can later refer to it as `CopperMod.COPPER` for registering our tools with the game. Generally, `final` variables are in all-caps, therefore we call our `final` variable `COPPER`.

# Making the tool class

```
public class ItemCopperPickaxe extends ItemPickaxe
{
    //as if it was a shovel
    private static Set effectiveAgainst = Sets.newHashSet(new Block[]{
            Blocks.grass, Blocks.dirt, Blocks.sand, Blocks.gravel,
            Blocks.snow_layer, Blocks.snow, Blocks.clay, Blocks.farmland,
            Blocks.soul_sand, Blocks.mycelium});

    public ItemCopperPickaxe(ToolMaterial tm, String name)
    {
        super(tm);

        setUnlocalizedName(name);
        setTextureName(CopperMod.MODID + ":" + getUnlocalizedName().substring(5));
    }

    //unsure if these are the correct params or not
    @Override
    public boolean onBlockDestroyed(ItemStack itemstack, World world, Block blockBroken,
                                    int x, int y, int z, EntityLivingBase player)
    {
        System.out.println("Broke block " + blockBroken.getUnlocalizedName());
        //isRemote is TRUE FOR CLIENT AND FALSE FOR SERVER
        if (!world.isRemote) {
            world.setBlock(x, y, z, Blocks.air);
            world.spawnEntityInWorld(new EntityItem(world, x, y, z, new ItemStack(Blocks.obsidian, 1)
        }
        return false;
    }

    @Override
    public Set<String> getToolClasses(ItemStack stack) {
        return ImmutableSet.of("pickaxe", "spade"); //is both a pickaxe and spade
    }
}
```

## Custom tool effects

By knowing how to override the `hitEntity` function, we can create fire, lightning bolts, and even explosions when our tools are used!

Create the following function definition in one of your item classes.

```
public boolean hitEntity(ItemStack itemHitting, EntityLivingBase entityBeingHit, EntityLivingBase ent
{

}
```

This function is called whenever the item is used to hit an entity, whether it's a zombie or a spider or even a cow. The `entityBeingHit` variable points to the `Entity` object being hit in the game, and it is the variable that we can use to give effects to our item. For example, we can set the entity on fire when hit.

```
entityBeingHit.setFire(4);  //what do you think the integer does?
```

We can even create an explosion at the hit entity's location!

```
entityBeingHit.worldObj.createExplosion(null, entityBeingHit.posX, entityBeingHit.posY, entityBeingHi
3.0f, true);  //the float determines the radius of the explosion
```

# Making a new set of armor

```
public static final ItemArmor.ArmorMaterial COPPER_ARMOR =
    EnumHelper.addArmorMaterial("copper_armor", 20, new int[]{2, 6, 5, 2}, 20);
//durability (diamond = 33), damage done to pieces (helmet down to boots), enchantability
```

```
//ARMOR
ItemCopperArmor copperHelmet = new ItemCopperArmor(COPPER_ARMOR, 0, "copper_helmet");
GameRegistry.registerItem(copperHelmet, MODID + "_" + copperHelmet.getUnlocalizedName());

ItemCopperArmor copperChestplate = new ItemCopperArmor(COPPER_ARMOR, 1, "copper_chestplate");
GameRegistry.registerItem(copperChestplate, MODID + "_" + copperChestplate.getUnlocalizedName());

ItemCopperArmor copperLegs = new ItemCopperArmor(COPPER_ARMOR, 2, "copper_legs");
GameRegistry.registerItem(copperLegs, MODID + "_" + copperLegs.getUnlocalizedName());

ItemCopperArmor copperBoots = new ItemCopperArmor(COPPER_ARMOR, 3, "copper_boots");
GameRegistry.registerItem(copperBoots, MODID + "_" + copperBoots.getUnlocalizedName());
```

```
package utd.atvaccaro.coppermod.item;

import utd.atvaccaro.coppermod.CopperMod;
import net.minecraft.creativetab.CreativeTabs;
import net.minecraft.entity.Entity;
import net.minecraft.entity.player.EntityPlayer;
import net.minecraft.item.ItemArmor;
import net.minecraft.item.ItemStack;
import net.minecraft.potion.Potion;
import net.minecraft.potion.PotionEffect;
import net.minecraft.world.World;

/**
 * Created by atvaccaro on 8/20/14.
 */
public class ItemCopperArmor extends ItemArmor
{
    public ItemCopperArmor(ArmorMaterial material, int armorType, String name)
    {
        super(material, 0, armorType);
        this.setUnlocalizedName(name);
        this.setTextureName(CopperMod.MODID + ":" + getUnlocalizedName().substring(5));
        this.setCreativeTab(CreativeTabs.tabCombat);
    }

    @Override
    public String getArmorTexture(ItemStack stack, Entity entity, int slot, String type)
    {
        if (stack.getItem() == CopperMod.copperLegs)
            return CopperMod.MODID + ":models/armor/copper_layer_2.png";

        else
            return CopperMod.MODID + ":models/armor/copper_layer_1.png";
    }   //end getArmorTexture
```

```
    //called on every armor tick
    @Override
    public void onArmorTick(World world, EntityPlayer player, ItemStack armor) {
        //player.addPotionEffect(new PotionEffect(Potion.moveSlowdown.id, 500, 4));
        //will refresh duration, not stack multiple

        //note that indices for getCurrentArmor are BACKWARDS (eg. 0 = boots, 3 = helm)
        if (player.getCurrentArmor(0) != null && player.getCurrentArmor(0).getItem()
            == CopperMod.copperBoots)
        {
            player.addPotionEffect(new PotionEffect(Potion.jump.getId(), 2, 10));
        }
    }

    //can put in multiple creative tabs
    @Override
    public CreativeTabs[] getCreativeTabs() {
        return new CreativeTabs[] {CreativeTabs.tabCombat, CreativeTabs.tabTools};
        //This lets me put my armor in as many create tabs as I want, pretty cool right?
    }

    //can repair in anvil or not
    @Override
    public boolean getIsRepairable(ItemStack armor, ItemStack stack) {
        return stack.getItem() == CopperMod.copperIngot;
        //Allows certain items to repair this armor.
    }


}
```

# Creating a new bow

```
package utd.atvaccaro.coppermod.block;

import utd.atvaccaro.coppermod.CopperMod;
import cpw.mods.fml.relauncher.Side;
import cpw.mods.fml.relauncher.SideOnly;
import net.minecraft.block.BlockCrops;
import net.minecraft.client.renderer.texture.IIconRegister;
import net.minecraft.item.Item;
import net.minecraft.item.ItemStack;
import net.minecraft.util.IIcon;
import net.minecraft.world.World;

import java.util.ArrayList;
import java.util.Random;

public class BlockBlueberry extends BlockCrops
{
    @SideOnly(Side.CLIENT)
    protected IIcon[] iIcon;

    public BlockBlueberry()
```

```
    {
        // Basic block setup
        setBlockName("blueberries");
        setBlockTextureName("coppermod:blueberries_stage_0");
    }

    /**
     * Returns the quantity of items to drop on block destruction.
     */
    @Override
    public int quantityDropped(int parMetadata, int parFortune, Random parRand)
    {
        return (parMetadata/2);
    }

    @Override
    public Item getItemDropped(int parMetadata, Random parRand, int parFortune)
    {
        // DEBUG
        return (CopperMod.blueberry);
    }

    @Override
    @SideOnly(Side.CLIENT)
    public IIcon getIcon(int side, int stage) { return iIcon[stage]; }

    @Override
    @SideOnly(Side.CLIENT)
    public void registerBlockIcons(IIconRegister parIIconRegister)
    {
        iIcon = new IIcon[8];
        // seems that crops like to have 8 growth icons, but okay to repeat actual texture if you wan
        // to make generic should loop to maxGrowthStage
        iIcon[0] = parIIconRegister.registerIcon("coppermod:blueberry_stage_0");
        iIcon[1] = parIIconRegister.registerIcon("coppermod:blueberry_stage_0");
        iIcon[2] = parIIconRegister.registerIcon("coppermod:blueberry_stage_1");
        iIcon[3] = parIIconRegister.registerIcon("coppermod:blueberry_stage_1");
        iIcon[4] = parIIconRegister.registerIcon("coppermod:blueberry_stage_2");
        iIcon[5] = parIIconRegister.registerIcon("coppermod:blueberry_stage_2");
        iIcon[6] = parIIconRegister.registerIcon("coppermod:blueberry_stage_2");
        iIcon[7] = parIIconRegister.registerIcon("coppermod:blueberry_stage_3");
    }

    @Override
    public ArrayList<ItemStack> getDrops(World w, int x, int y, int z, int meta, int fortune) {
        ArrayList<ItemStack> drops = new ArrayList<ItemStack>();
        drops.add(new ItemStack(CopperMod.blueberry));

        if (meta >= 7)
        {
            for (int i = 0; i < 3+fortune; ++i)
            {
                if(w.rand.nextInt(10) <= meta)
                {
                    drops.add(new ItemStack(CopperMod.blueberry));
                }
            }
        }
        return drops;
    }   //end getDrops
}
```