Extracted from:

Learn to Program with Minecraft Plugins

Create Flying Creepers and Flaming Cows in Java

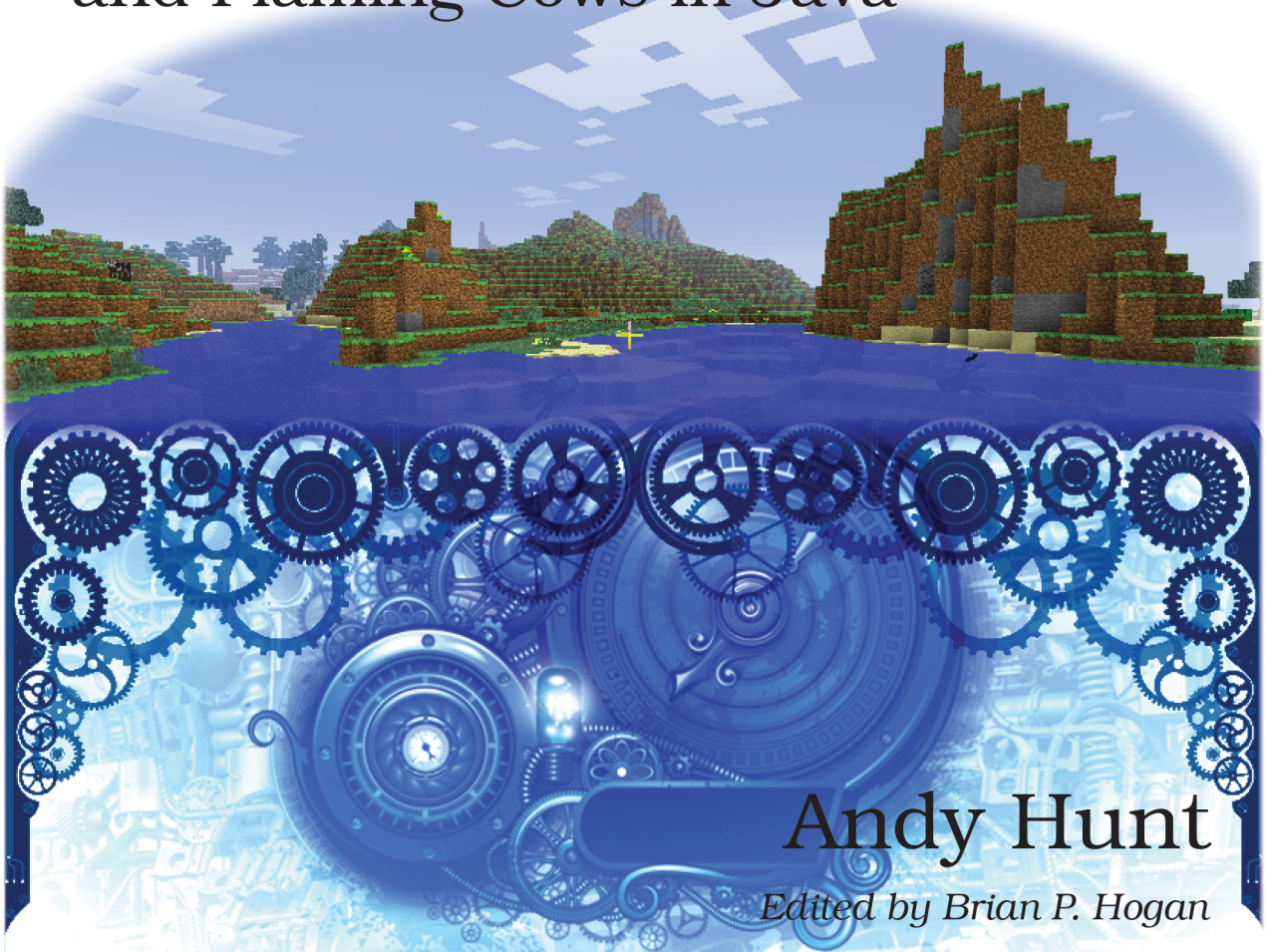
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Learn to Program with Minecraft Plugins

Create Flying Creepers and Flaming Cows in Java



Andy Hunt

Edited by Brian P. Hogan

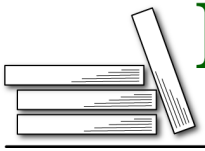
Learn to Program with Minecraft Plugins

Create Flying Creepers and Flaming Cows in Java

Andy Hunt

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Brian Hogan (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-937785-78-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—May 2014

CHAPTER 6

Add a Chat Command, Locations, and Targets

Your Toolbox

In this chapter we'll look at how plugins are constructed, and add these abilities to your toolbox:

- Add a command to a plugin by checking for that string in `onCommand` and adding the command code
- Add new command descriptions to `plugin.yml`
- Work with Minecraft coordinates (`Location`)
- Find nearby blocks or entities (`BlockIterator`)

How Does Minecraft Know About Your Plugin?

We've been using a bunch of objects in the Minecraft code. For example, you know that a player is represented as a `Player` object and the server is a `Server` object.

So it shouldn't be too surprising to realize that our plugins are, in fact, `Plugin` objects. Bukkit has kindly defined a basic "recipe," a basic `Plugin` class that it knows about. Our job, as plugin writers, is to provide our own plugin code that fits into that framework.

As we've seen, the first line of a plugin declares the plugin's name and then adds the magical phrase `extends JavaPlugin`:

```
import org.bukkit.plugin.java.JavaPlugin;

public class MyFavoritePlugin extends JavaPlugin {
```

That makes JavaPlugin a parent of your class MyFavoritePlugin, just like the examples in the last chapter.

The Minecraft server already knows how to work with a JavaPlugin, and since that's your plugin's parent, it now knows how to work with your plugin—even though your plugin didn't exist when Minecraft was created. It's counting on the fact that you'll write a couple of functions that it knows how to call.

In addition to the plugin code itself, Minecraft needs a configuration file for the plugin, named plugin.yml. You saw a description of this back on page ?, while we were building plugins the first time. It tells the server which commands your plugin will handle.

With that configuration file and your code, the Minecraft server can run your plugin just like any other part of the game.

Plugin: SkyCmd

We're going to create a brand-new plugin called SkyCmd. In it, we'll create a command named sky that will teleport all nearby creatures (not players) fifty blocks up into the air. Very handy at night with skeletons and creepers about.

In the SkyCmd directory, plugin.yml has an entry for our new command, /sky. Here's the whole source file to the plugin:

SkyCmd/src/skycmd/SkyCmd.java

```
package skycmd;

import java.util.logging.Logger;
import org.bukkit.Location;
import org.bukkit.command.Command;
import org.bukkit.command.CommandSender;
import org.bukkit.entity.Player;
import org.bukkit.plugin.Plugin;
import org.bukkit.plugin.java.JavaPlugin;
import org.bukkit.entity.Entity;
import java.util.List;

public class SkyCmd extends JavaPlugin {

  public boolean onCommand(CommandSender sender, Command command,
                           String commandLabel, String[] args) {
    if (commandLabel.equalsIgnoreCase("sky")) {
      if (sender instanceof Player) {
        Player me = (Player)sender;
        List<Entity> list = me.getNearbyEntities(50,50,50);
        for (Entity target : list) {
          if (!(target instanceof Player)) {
            Location loc = target.getLocation();
```

```
            double y = loc.getY();
            loc.setY(y+50);
            target.teleport(loc);
          }
        }
        return true;
      }
    }
    return false;
  }
}
```

Compare this to our original, very simple HelloWorld.java file. Notice right at the top, the package statement and later the public class statement now each refer to SkyCmd instead of HelloWorld.

Let's take a closer look at how a plugin handles a chat command like /sky.

Handle Chat Commands

When the player types a command, your onCommand method will be called.

The player's command is passed to you in a string, which we've named commandLabel. So the first thing you need to do is check and see if the command the player typed is the one you want. How do you check to see if strings are equal? Down on the line at ❶ we'll use the string's function equalsIgnoreCase to check if the player typed "sky". (Remember, you can't use == on strings; you have to use either equals or equalsIgnoreCase, and most of the time you want to ignore the case, so "sky" will match "Sky", "SKY", and even "sKy".)

If we got a match for "sky", then we'll execute this next code block, in between the braces—{ and }.

The next thing we need to check is a little awkward; it turns out that the CommandSender that gets passed to us here may not be a Player. It could be a Player object, but it could be a Console instead, or who knows what else. We want to make sure it's really a Player, so we'll check for that explicitly at ❷, using the Java keyword instanceof. This tests to see if the thing passed in is really a Player. If it is, then we're going to do the bulk of the command starting at ❸.

This begins with another bit of magic, just like we saw with parent/child recipes at the end of Chapter 5, *Plugins Have Objects*, on page ?. Now that we've confirmed the variable sender is really of type Player (not just a CommandSender or any other parent or child), we can convert it to the type Player, using a *cast* operator.

So the expression (Player) sender returns the variable sender, converted with a cast operator to the type Player so you can assign it to the variable me. It sounds messy, and it is a bit, but it's also something you can just copy and paste, as we'll be using this little recipe in almost every command plugin to get a Player object.

Now that we have a real Player object referenced by me, we can get the list of all nearby entities with me.getNearbyEntities(50,50,50);, which will get us all the entities within fifty blocks of us and return all of these entities in a List that we'll go through with a for loop.

We'll go over the details of lists in the next chapter, but first we'll look at how Location objects work. In this case, we're setting the variable target to each entry in the list of nearby entities as we go through the for loop. If the target is not a fellow player, then we want to fling it skyward, which we do by changing its location.

Location objects are important—that's how you get and set the coordinates of anything in Minecraft. Here's how we'll manipulate locations to fling the creatures up in the air.

Use Minecraft Coordinates

A Location stores three coordinates: x, y, and z, as the following figure shows.

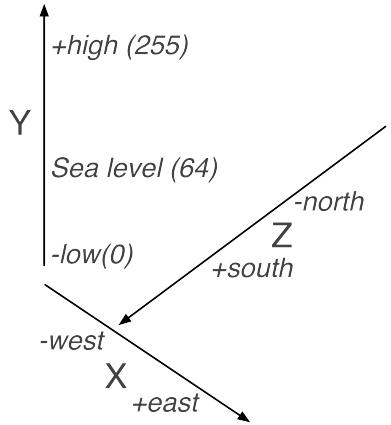


Figure 6—Minecraft coordinate system

The x value goes west (negative) to east (positive), the z coordinate goes north (negative) to south (positive), and y goes down (negative) to up (positive), with a y value of 0 being the bottom layer of bedrock, and 64 being sea level. That

means that to make a player or other entity fly up in the air, you need to add some to the y value.

We'll get each target's current y value from `loc` and save it as `y`. Next we'll change the value in `loc` by adding 50. Here's the fun part: by calling `target.teleport(loc)` we tell the target to teleport itself to this new location. Finally, we return `true` to indicate that this was our command and that we executed it successfully.

Notice that if the caller wasn't a `Player`, or if the command typed wasn't `"sky"`, we return `false`, indicating it wasn't our plugin's command.

Whew! That's a lot of stuff in a few lines of code. But give it a shot and compile and install it using `build.sh` just like we've been doing:

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

Stop and restart your server, and try out the new command `/sky` for fun. Make sure you are in survival mode instead of creative mode,[1] and wait for night to fall and the creepers to come....

1. In the Minecraft game, you can do this by typing `/gamemode c` for creative or `/gamemode s` for survival.