Extracted from:

Learn to Program with Minecraft Plugins

Create Flying Creepers and Flaming Cows in Java

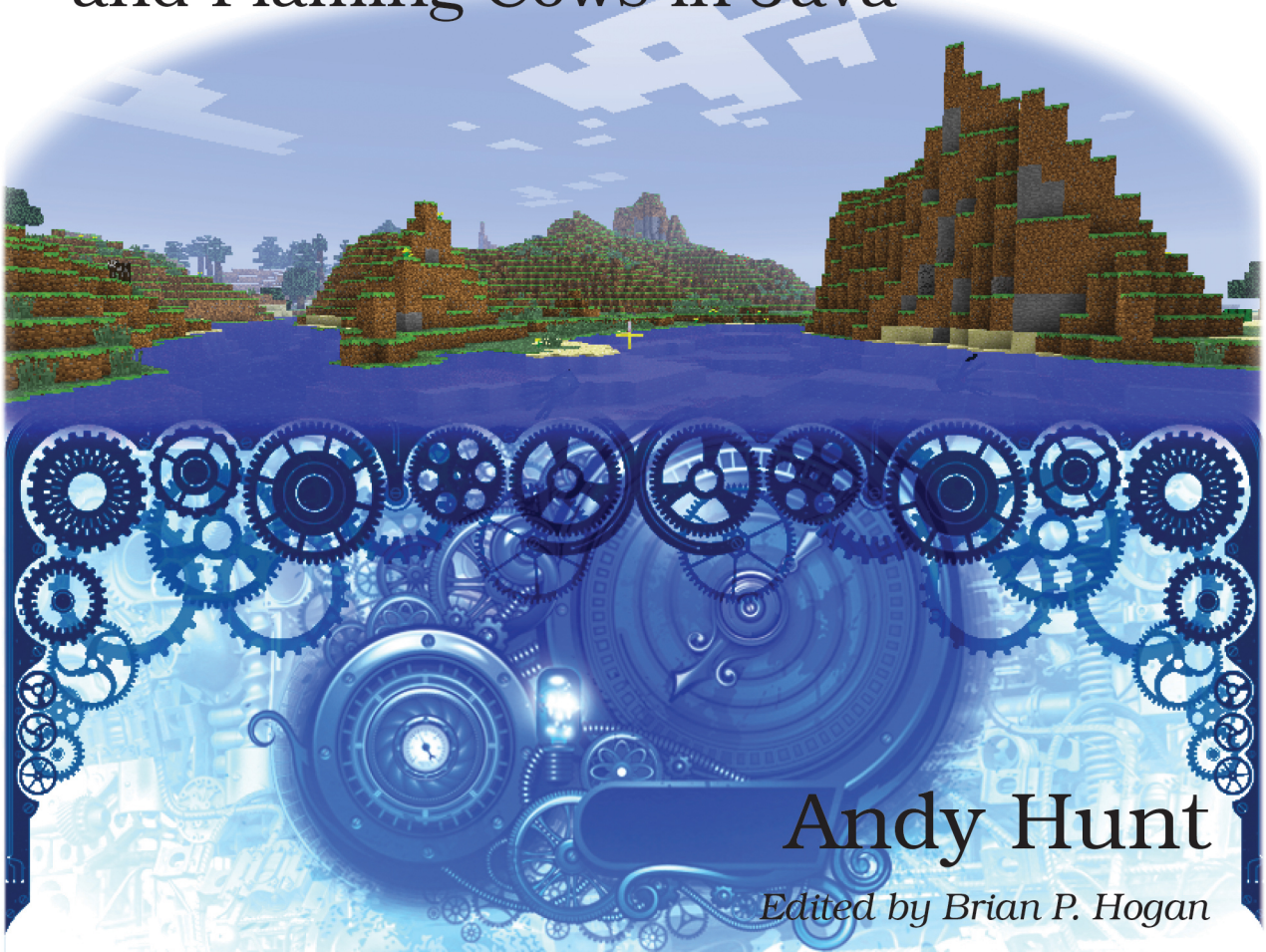
The Pragmatic Bookshelf

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

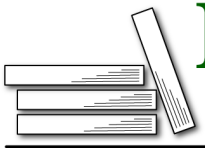
Learn to Program with Minecraft Plugins

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 9

Modify, Spawn, and Listen in Minecraft

Your Toolbox

With this chapter you'll add these abilities to your toolbox:

- Modify blocks in the world
- Modify and spawn new entities
- Listen for and react to game events
- Manage plugin permissions

This is exciting! Now you have most of the basic tools you need; you can alter the world and react to in-game events.

Now we're going to go beyond issuing simple commands and dropping squid bombs, and look at a wider range of things you can do in Minecraft. By the end of this chapter, you'll be able to affect behavior in the game without having to issue *any* commands at all.

All you have to do is listen—you'll see how, by learning about Minecraft events. We'll listen for events, act on them, and even schedule our own events to fire sometime in the future.

From your plugin code, you can change existing blocks and entities, and you can spawn new ones. We'll look at exactly how to do that:

- Modify existing blocks: change things like location, properties, and contents
- Modify existing entities: change properties on a `Player`
- Spawn new entities and blocks

We've done some of this already—we've changed a `Player`'s location, and we've spawned more than a few `Squids`. Let's take a closer look at what else you can

do with the basic elements in the Minecraft world, and then we'll see how you can react to in-game events to affect those elements and create new ones.

Modify Blocks

The basic recipe for a block object in Minecraft is listed in the Bukkit documentation under org.bukkit.block.Block.[1]

There are many interesting functions in a Block, and we won't cover them all, but here are a few of the most useful and interesting things you can do to a block:

- getLocation() returns the Location for this block. Only one block can exist at any location in the world, and every location contains a block, even if it's just air.
- getType() returns the Material this block is made of.
- setType(Material type) sets a new Material for this block.
- getDrops() returns a list of items (actually a Collection of ItemStack lists) that would fall if this block were destroyed.
- breakNaturally() breaks the block and spawns any items, just as if a player had done the breaking. Returns true if the block was successfully destroyed, and false if it wasn't.

Let's play with that a bit.

Plugin: Stuck

Let's look at a plugin that will encase a player in solid rock (the full plugin is in code/Stuck). When you issue the command stuck with a player's name, that player will suddenly be encased in a pile of blocks. (If you're alone on the server, your player name might be the wonderfully descriptive name "player.")

We'll start by looking at pieces of this plugin, and then put it all together.

All the interesting parts are in a separate helper function, named stuck. The main part of the plugin should look pretty familiar by now:

Stuck/src/stuck/Stuck.java

```
package stuck;

import org.bukkit.Location;
import org.bukkit.Material;
import org.bukkit.World;
```

1. http://jd.bukkit.org/rb/doxygen/d9/d48/interfaceorg_1_1bukkit_1_1block_1_1Block.html

```
import org.bukkit.block.Block;
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;

public class Stuck extends JavaPlugin {
  public boolean onCommand(CommandSender sender, Command command,
                           String commandLabel, String[] args) {
    if (commandLabel.equalsIgnoreCase("Stuck")) {
      if(args.length == 1) {
        Player player = getServer().getPlayer(args[0]);
        if (player != null) { return stuck(player); }
      } else {
        sender.sendMessage("Usage: /stuck playerName");
      }
    }
    return false;
  }
```

In onCommand we'll try to get the named player, which may or may not work. If it doesn't work (if there's no player online with that name), we'll fall out of the if/then/else stuff and end up returning false from onCommand.

If it does work (that is, if we found the player), then we'll go ahead and call stuck, passing in the player object we got.

Here's the beginning of the stuck function:

Stuck/src/stuck/Stuck.java

```
public boolean stuck(Player player) {
  World world = player.getWorld();
  Location loc = player.getLocation();
  int playerX = (int) loc.getX();
  int playerY = (int) loc.getY();
  int playerZ = (int) loc.getZ();
  loc.setX(playerX + 0.5); loc.setY(playerY); loc.setZ(playerZ + 0.5);
  player.teleport(loc);
```

The first thing we'll do inside of the stuck function is get the correct world and the player's current location in loc. Over the next few lines, we'll set up to teleport the player to the center of the block he or she is stuck in right now. That makes it easier to plunk blocks down all around the player.

And how are we going to do that, exactly? Well, we know that a player takes up two blocks. The location we got for the player is really where the character's legs and feet are. The block on top of that (y+1) is the player's head and chest. So we want a bunch of blocks, arranged like a stack of two blocks on all four

sides of the player, plus a block underneath and one on top. That should be ten blocks in all, as you can see in the figure.

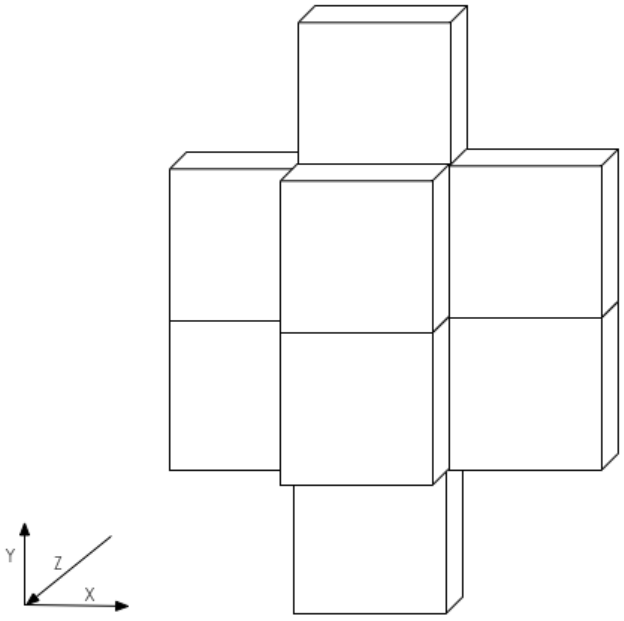


Figure 8—Trapping a player in blocks

We know where each of those blocks goes, based on the player's location. So we've got a case where we need ten sets of coordinates, each one offset from the player's base block. We need a list of lists.

And that's what you'll see next. It's an int array of ten elements, and each element is an int array of three offsets, one each for x, y, and z value:

Stuck/src/stuck/Stuck.java

```
int[][] offsets = {
 //x,  y,  z
  {0,  -1, 0},
  {0,  2,  0},
  {1,  0,  0},
  {1,  1,  0},
  {-1, 0,  0},
  {-1, 1,  0},
  {0,  0,  1},
  {0,  1,  1},
  {0,  0, -1},
  {0,  1, -1},
};
```

We'll use a simple for loop to go through this list of offsets. The first element in the list is indexed at 0, and we'll go up to (but not including) the length of the list. By using the length of the offsets list instead of sticking in a fixed number, like 10, it makes it easier if we ever want to add extra blocks to the list (remember we're adding the playerX, playerY, and playerZ offsets from the preceding code):

Stuck/src/stuck/Stuck.java

```
    for(int i = 0; i < offsets.length; i++) {
      int x = offsets[i][0];
      int y = offsets[i][1];
      int z = offsets[i][2];
      Block b = world.getBlockAt(x + playerX, y + playerY, z + playerZ);
      b.setType(Material.STONE);
    }
    return true;
  }
}
```

So here we are, going through the list of offsets. At each list index (which is in i), we need to pick out the three elements x, y, and z. In each of the small arrays, x is first at index 0. The Java syntax lets you work with arrays of arrays by writing both indexes, with the big list first. Think of this set of numbers as a table or a matrix, with rows and columns, like you might find in an Excel spreadsheet. You specify indexes in "row-major order," which just means the row comes first, then the column. For each trip through the loop, we'll pick out an x, y, and z value from the list. That's the location of a block we want to turn to stone.

We get the block at that location we want—in this case, by adding the x, y, and z offset to the player's location (playerX, playerY, and playerZ from the code). With the block in hand, simply set its material to stone by using the constant Material.STONE. All the material types are listed in the documentation for org.bukkit.Material. You could, for instance, remove a block without breaking it—you'd set the block's material to Material.AIR.

Here's the code for the full plugin, all together:

Stuck/src/stuck/Stuck.java

```
package stuck;

import org.bukkit.Location;
import org.bukkit.Material;
import org.bukkit.World;
import org.bukkit.block.Block;
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;

public class Stuck extends JavaPlugin {
  public boolean onCommand(CommandSender sender, Command command,
                           String commandLabel, String[] args) {
    if (commandLabel.equalsIgnoreCase("Stuck")) {
      if(args.length == 1) {
        Player player = getServer().getPlayer(args[0]);
        if (player != null) { return stuck(player); }
      } else {
        sender.sendMessage("Usage: /stuck playerName");
      }
    }
    return false;
  }
  public boolean stuck(Player player) {
    World world = player.getWorld();
    Location loc = player.getLocation();
    int playerX = (int) loc.getX();
    int playerY = (int) loc.getY();
    int playerZ = (int) loc.getZ();
    loc.setX(playerX + 0.5); loc.setY(playerY); loc.setZ(playerZ + 0.5);
    player.teleport(loc);

    int[][] offsets = {
     //x,  y,  z
      {0,  -1, 0},
      {0,  2,  0},
      {1,  0,  0},
      {1,  1,  0},
      {-1, 0,  0},
      {-1, 1,  0},
      {0,  0,  1},
      {0,  1,  1},
      {0,  0, -1},
      {0,  1, -1},
    };

    for(int i = 0; i < offsets.length; i++) {
      int x = offsets[i][0];
      int y = offsets[i][1];
      int z = offsets[i][2];
      Block b = world.getBlockAt(x + playerX, y + playerY, z + playerZ);
      b.setType(Material.STONE);
    }
    return true;
  }
}
```

Compile and deploy the Stuck plugin and give it a try. What happens if the player is standing on the ground or up in the air? What does it look like from the player's point of view, inside the blocks?

Try This Yourself

In the Stuck plugin, we've encased a player in the minimum number of blocks needed to enclose the player. But from the outside, it makes kind of a weird-looking shape.

So here's what you need to do: add extra blocks so that the player is encased in a solid rectangle, measuring 3 blocks wide, 3 blocks deep, and 4 blocks tall. Add the extra blocks to the list of block offsets, then recompile and deploy and see if you've put the extra blocks in the right places. From the outside, it should look like a solid, rectangular block.

Modify Entities

Entities, as you might expect, are quite different from blocks. For one thing, there are many more kinds of entities, and they have different kinds of abilities (and functions for us). With blocks, all you have to do is change the ID and perhaps add some additional information, but entities are more complicated.

To start off, all entities have the capabilities described in org.bukkit.entity.Entity, which include the following useful functions:

- getLocation() returns the Location of the entity.
- setVelocity(Vector velocity) sets its velocity.
- teleport(Location location) teleports the entity to a new location. It returns true if it was successful, and false otherwise.
- teleport(Entity destination) teleports the entity to the location of the *destination* entity. Way to crash a party.

Then, depending on the type of the entity, you might have other cool functions to play with. Living entities (org.bukkit.entity.LivingEntity), for example, have the following extra functions:

- launchProjectile(Projectile) lets you throw an egg, fire an arrow, toss a snowball, or launch other kinds of Projectiles. It returns the launched Projectile.
- getLastDamage() returns the amount of damage (as an int) just inflicted on this entity.

- setLastDamage(int damage) sets the damage value of the last damaging event. You could make it a lot or a little.
- damage(int amount) inflicts damage on this entity.
- getHealth() returns a double of this entity's health. It can be zero (dead) up to the amount returned by getMaxHealth().
- setHealth(double health) sets the health. Zero is dead.

You may have noticed that not all these functions are declared in LivingEntity itself. This is where Java gets a little messy. The familiar entity objects incorporate a lot of different parent recipes. For instance, a Cow is an Animals, but also an Ageable, a Creature, a LivingEntity, a Damageable object, and, of course, an Entity.

That means it uses functions from all these different parents. Because Cow inherits from Ageable, you get functions where you can alter a Cow property to change its age—make it a baby or an adult—let it breed or not, and so on.

A Player, on the other hand, does not use Ageable, so you can't turn players into babies, even if they're acting like them. Instead, a Player has a whole different set of functions available, including functions to change the weather, the time of day, and the player's experience level, food level, inventory, and so on.

Spawn Entities

You can use several functions to spawn different entities and creatures, as well as game objects—like an Ender Pearl.[2] To create new things in the world, we'll use functions defined in org.bukkit.World. There are a few useful ways to spawn things, depending on what you're making and how you want to specify it:

- spawn(Location location, Class whatToSpawn) is what we used to spawn squid previously. You can get the class to pass in by using the class function on the class name itself, like this: Cow.class.
- spawnArrow(Location location, Vector velocity, 0.6, 12) returns a new Arrow. The hard-coded numbers 0.6 and 12 seem to make an arrow that flies nicely.
- spawnCreature(Location loc, CreatureType type) returns a LivingEntity.
- spawnEntity(Location loc, EntityType type) is used for nonliving entities; it returns an Entity.

2. In survival mode, right-clicking on an Ender Pearl will transport you to where it lands.

- spawnFallingBlock(Location location, Material material, byte data) means "watch out below." What you pass in for data depends on the material.
- strikeLightning(Location loc) lets you hurl lightning bolts, in case you fancy being Zeus.

Plugin: FlyingCreeper

Here's a plugin that shows spawning two entities: a bat and a creeper. We'll make the creeper ride the bat, and then turn the bat invisible using a potion effect. The result is a nightmarish, terrifying, flying creeper.

Here are the guts of the plugin:

FlyingCreeper/src/flyingcreeper/FlyingCreeper.java

```
Location loc = player.getLocation();
loc.setY(loc.getY() + 5);
Bat bat = player.getWorld().spawn(loc, Bat.class);
Creeper creeper = player.getWorld().spawn(loc, Creeper.class);
bat.setPassenger(creeper);
PotionEffect potion = new PotionEffect(
        PotionEffectType.INVISIBILITY,
        Integer.MAX_VALUE,
        1);
bat.addPotionEffect(potion);
```

Notice we're using the first version of spawn that we just saw, where we pass in a location and a class—in this case, Bat.class and Creeper.class.

All Entity objects have a setPassenger() function. In theory, you could even ride primed TNT. But I wouldn't advise it. Here we're going to have the creeper ride the bat by setting the bat's passenger to the creeper.

Next we need to turn the bat invisible to make the flying creeper look more convincing. Fortunately, all LivingEntity objects can use potion effects.[3] We'll create a new potion effect, which lets us specify the effect's type, duration, and magnitude:

```
PotionEffect (PotionEffectType type, int duration, int amplifier)
```

In this case, the type is PotionEffectType.INVISIBILITY and we want it to last forever, so we'll make the duration the largest possible value we can: Integer.MAX_VALUE. There is no integer larger. The magnitude doesn't really matter in this case, as you can't be any "more invisible," so we'll just use a 1.

3. All potion effect types are listed at http://jd.bukkit.org/rb/doxygen/d3/d70/classorg_1_1bukkit_1_1potion_1_1PotionEffectType.html.

Finally, we add that new potion to the bat, and it's invisible.

Congratulations! You are now the proud owner of flying creepers. Good luck, and stay low.

For extra credit, you could go back and modify the `SquidBomb` to generate a ton of invisible creepers instead of squid. That'd be fun.

We'll see some more examples of modifying and spawning entities in the next section, once we see how to listen for game events.