Extracted from:

Learn to Program with Minecraft Plugins, 2nd Edition

Create Flaming Cows in Java Using CanaryMod

The Pragmatic Bookshelf

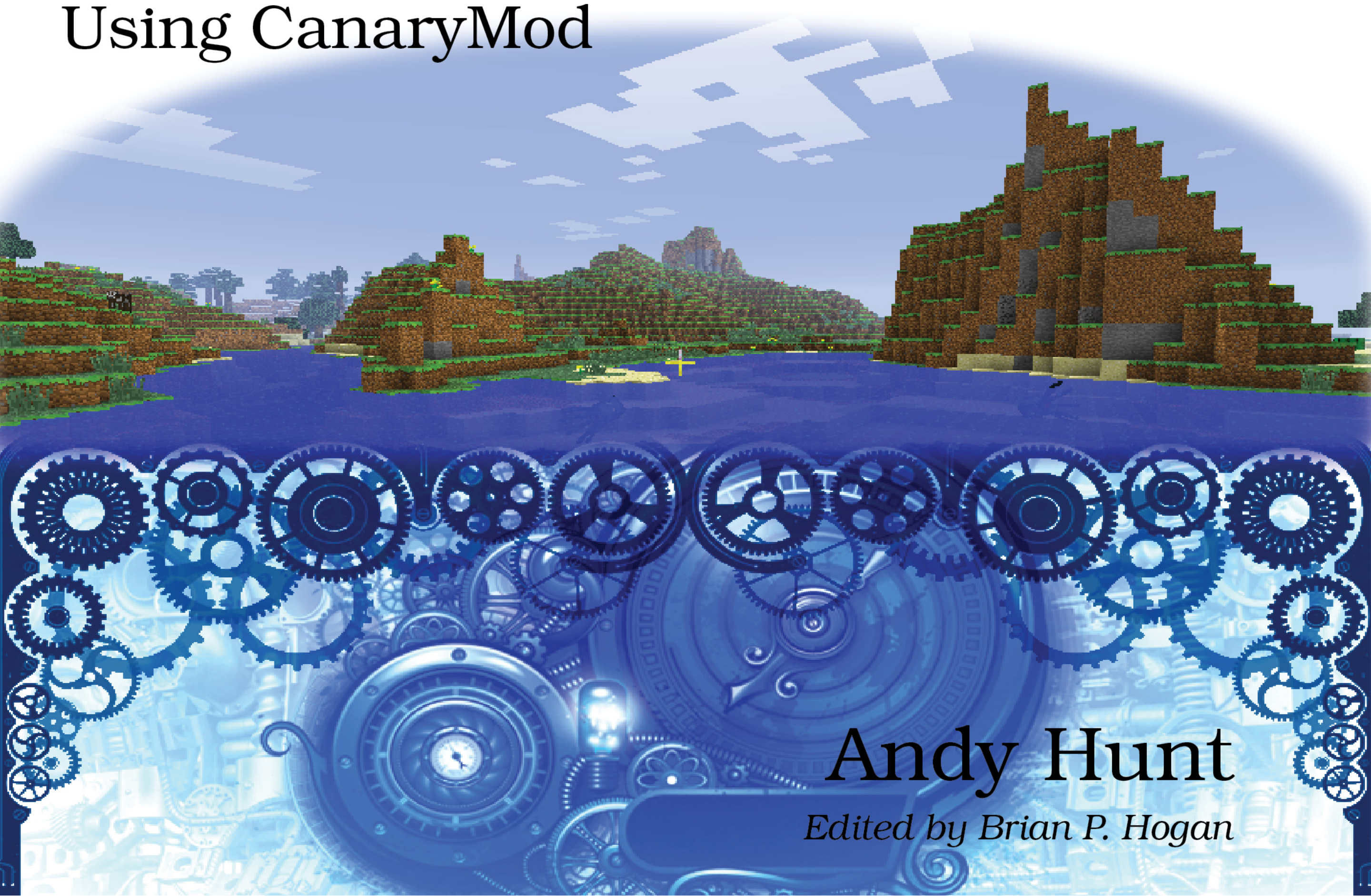
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

2ND ed.
now with CanaryMod

Learn to Program with Minecraft Plugins

Create Flaming Cows in Java Using CanaryMod



Andy Hunt

Edited by Brian P. Hogan

Learn to Program with Minecraft Plugins, 2nd Edition

Create Flaming Cows in Java Using CanaryMod

Andy Hunt

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Brian Hogan (editor)
Potomac Indexing, LLC (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

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

Printed in the United States of America.
ISBN-13: 978-1-941222-94-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—November 2014

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.

CHAPTER 9

Modify, Spawn, and Listen in Minecraft

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 Canary documentation under net.canarymod.api.world.blocks.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:

1. https://ci.visualillusionsent.net/job/CanaryLib/javadoc

- 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 BlockType this block is made of.
- rightClick(Player player) simulates a right-click on the block. Useful for forcing changes to blocks like levers, buttons, and doors.

Let's play with some blocks, Minecraft style.

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 net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import com.pragprog.ahmine.ez.EZPlugin;

public class Stuck extends EZPlugin {

  @Command(aliases = { "stuck" },
            description = "Trap a player in cage of blocks",
            permissions = { "" },
            min = 2,
            toolTip = "/stuck name")
  public void stuckCommand(MessageReceiver caller, String[] args) {
    Player victim = Canary.getServer().getPlayer(args[1]);
    if (victim != null) {
      stuck(victim);
    }
  }
```

In the @Command spec, we're setting the minimum number of arguments to 2. That way we don't have to write any code to check it ourselves; the system will do it automatically. Then in stuckCommand itself, 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 body and return without doing anything.

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

Here's the beginning of the stuck function:

Stuck/src/stuck/Stuck.java
```
public void stuck(Player player) {
  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.teleportTo(loc);
```

The first thing we'll do inside of the stuck function is get 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 Figure 3, *Trapping a player in blocks*, on page 8.

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 values:

Stuck/src/stuck/Stuck.java
```
int[][] offsets = {
 //x,  y,  z
  {0,  -1, 0},
  {0,  2,  0},
  {1,  0,  0},
  {1,  1,  0},
```

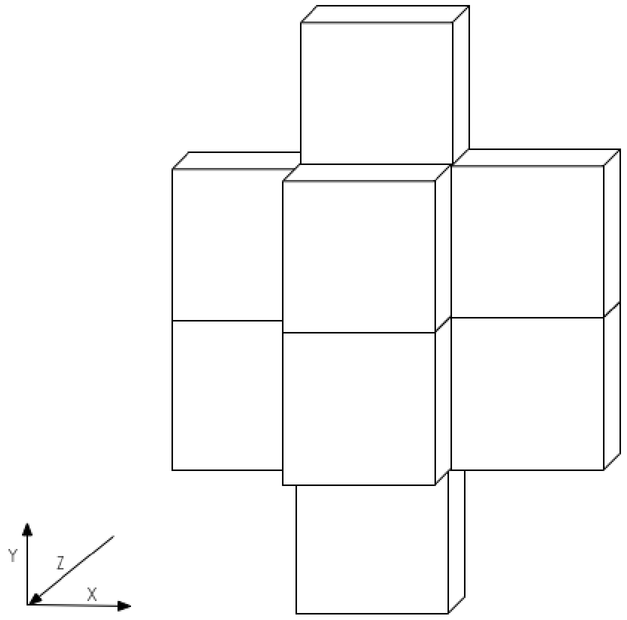


Figure 3—Trapping a player in blocks

```
  {-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, we can more easily add extra blocks to the list if we ever want to (remember, we're adding the playerX, playerY, and playerZ offsets from the preceding code):

Stuck/src/stuck/Stuck.java

```java
for(int i = 0; i < offsets.length; i++) {
  int x = offsets[i][0];
  int y = offsets[i][1];
  int z = offsets[i][2];
  setBlockAt(new Location(x+playerX, y+playerY, z+playerZ),
    BlockType.Stone);
}
```

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 a Microsoft 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 x, y, and z values 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 BlockType.Stone. All the different block types are listed in the documentation for net.canarymod.api.world.blocks.BlockType. You could, for instance, remove a block without breaking it—you'd set the block's material to BlockType.Air, like we did back with the array towers.

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

Stuck/src/stuck/Stuck.java
```
package stuck;

import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import com.pragprog.ahmine.ez.EZPlugin;

public class Stuck extends EZPlugin {

  @Command(aliases = { "stuck" },
            description = "Trap a player in cage of blocks",
            permissions = { "" },
            min = 2,
            toolTip = "/stuck name")
  public void stuckCommand(MessageReceiver caller, String[] args) {
    Player victim = Canary.getServer().getPlayer(args[1]);
    if (victim != null) {
      stuck(victim);
    }
  }
  public void stuck(Player player) {
    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.teleportTo(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];
      setBlockAt(new Location(x+playerX, y+playerY, z+playerZ),
        BlockType.Stone);
    }
  }
}
```

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 three blocks wide, three blocks deep, and four 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 block type and perhaps add some additional information (like on a sign), but entities are more complicated.

To start off, all entities have the capabilities described in net.canarymod.api.entity.Entity. Each Entity object includes the following useful functions:

getLocation()	Return the Location of the entity
setFireTicks(int ticks)	Set time to keep an entity on fire
setRider(Entity rider)	Set the entity's rider
spawn()	Spawn this kind of Entity into the world
teleport(Location location)	Teleport the entity to a new location

Then, depending further on the type of the entity, you might have other cool functions to play with. Living entities (net.canarymod.api.entity.living.EntityLiving), for example, have the following extra functions that other nonliving entities don't have:

getItemInHand()	Return the item this entity is holding.
setAttackTarget(LivingBase livingbase)	Set this entity's attack target.
getHealth()	Return a double of this entity's health. It can be zero (dead) up to the amount returned by getMaxHealth().
setHealth(double health)	Set the health. Zero is dead, without causing damage.
kill()	Kill this entity, causing damage (and loot drops, etc.).

You may have noticed that not all these functions are declared in EntityLiving 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 Ageable, an EntityAnimal, an EntityLiving object, and, of course, an Entity.

That means it uses functions from all these different parents. For example, because Cow inherits from Ageable, you get functions where you can alter a Cow property to change its growing age.

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 set and get 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 our EZPlugin helper instead of writing out all the code directly. It's not that the code is particularly complicated or hard to understand; it's just that these couple of lines of code will always be the same, so it makes sense to use a helper function. That way you only need to use the one-line helper function call, instead of using several lines of duplicated code. Let's take a close look at what those helper functions actually do.

Here's the method we've been using to spawn cows, squid, and such:

EZPlugin/src/com/pragprog/ahmine/ez/EZPlugin.java

```java
public static EntityLiving spawnEntityLiving(Location loc, EntityType type) {
  EntityFactory factory = Canary.factory().getEntityFactory();
  EntityLiving thing = factory.newEntityLiving(type, loc);
  thing.spawn();
  return thing;
}
```

It's a little messy, maybe, but it's a common Java pattern. The idea is that first you obtain a factory object, in this case, an EntityFactory. The factory works as the name implies; it generates things for you. Here, it's generating new EntityLiving objects. But just creating a new object (even a Cow object) isn't enough to make it exist in the Minecraft world. You need to tell the object to spawn itself.

Spawning particles is a little easier:

EZPlugin/src/com/pragprog/ahmine/ez/EZPlugin.java

```java
public static void spawnParticle(Location loc, Particle.Type type) {
  loc.getWorld().spawnParticle(new Particle(loc.getX(),
    loc.getY(), loc.getZ(), type));
}
```

Here we just need to use the spawnParticle function in World, and pass it a new Particle initialized with individual coordinates x, y, and z. Again, it's not complicated, but a helper function literally "helps" us keep code cleaner and easier to read.

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

The function we've been using to set block types is also straightforward:

EZPlugin/src/com/pragprog/ahmine/ez/EZPlugin.java
```
public static void setBlockAt(Location loc, BlockType type) {
  loc.getWorld().setBlockAt(loc, type);
}
```

Although a Block has its own setType, that doesn't make the change in the world like we want. So instead, we use the World's setBlockAt() function. By always using the helper function, we're sure to always use the correct version.

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. Notice I'm not using the helper functions to spawn this time; instead I'm using the factory calls directly, as I'm using a slightly different version of spawn.

FlyingCreeper/src/flyingcreeper/FlyingCreeper.java
```
Location loc = me.getLocation();
loc.setY(loc.getY() + 5);

EntityFactory factory = Canary.factory().getEntityFactory();
EntityLiving bat = factory.newEntityLiving(EntityType.BAT, loc);
EntityLiving creep = factory.newEntityLiving(EntityType.CREEPER, loc);
bat.spawn(creep);

PotionFactory potfact = Canary.factory().getPotionFactory();
PotionEffect potion =
  potfact.newPotionEffect(PotionEffectType.INVISIBILITY,
          Integer.MAX_VALUE, 1);
bat.addPotionEffect(potion);
```

All Entity objects can have riders. 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 spawning the bat with a creeper rider (the argument to spawn).

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:

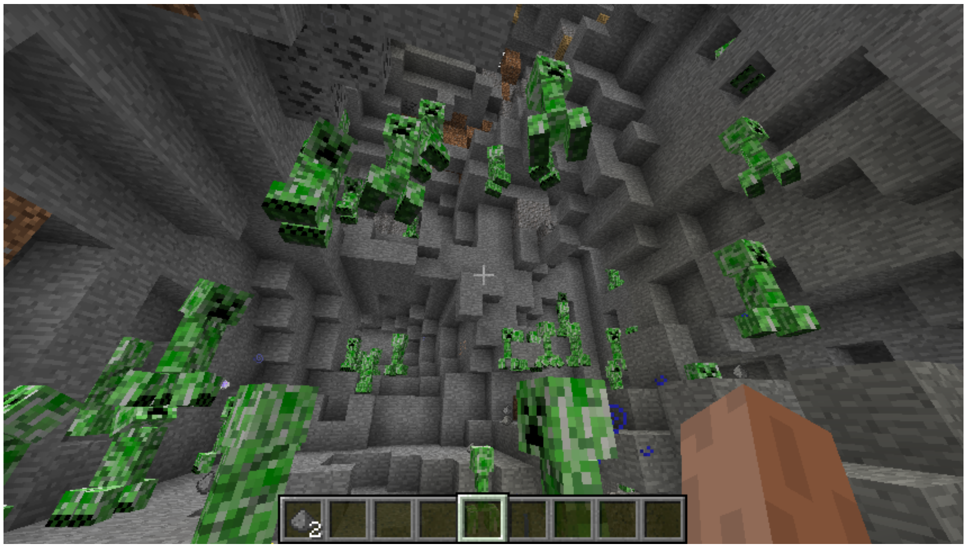
```
PotionEffect (PotionEffectType type, int duration, int amplifier)
```

3. All potion effect types are listed at https://ci.visualillusionsent.net/job/CanaryLib/javadoc/net/canarymod/api/potion/PotionType.html.

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.

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.