

Extracted from:

# Learn to Program with Minecraft Plugins

Create Flying Creepers and Flaming Cows in Java

This PDF file contains pages extracted from *Learn to Program with Minecraft Plugins*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2014 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

# Learn to Program with Minecraft Plugins

Create Flying Creepers  
and Flaming Cows in Java



# Learn to Program with Minecraft Plugins

Create Flying Creepers and Flaming Cows in Java

Andy Hunt

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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>.

Minecraft is ®, ™, and © 2009–2014 Mojang/Notch.

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](mailto:rights@pragprog.com).

Copyright © 2014 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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

## Organize Instructions into Functions

So now that you can store all kinds of data in variables, next you need to learn how to write instructions to do fun actions with all that data, from printing messages to flinging flaming cows in Minecraft.

As you've seen, you can also tell Java to do things. In Java, you organize lines of code (instructions) inside a pair of curly braces, like { and }. You give that section of code a name, and those instructions will be run in order, one line after another. We call that a *function* (sometimes we'll call it a *method*; for now they mean mostly the same thing).

Why do we bother with functions at all? Couldn't we just have one big list of instructions and be done? Well yes, we could, but it can get very confusing that way.

Think of a list of instructions and ingredients to make a cake with frosting:

- Blend together and bake
- Flour
- Butter
- Sugar
- Milk
- Eggs
- Vanilla
- Cocoa powder
- Confectioner's sugar
- Butter
- Milk
- Mix together and spread on cake

Which part of the list is for the cake itself, and which is for the frosting? Maybe the frosting part starts at the cocoa powder. Then again, maybe it's a chocolate cake base with a vanilla frosting. The point is, it's hard to tell. It might work as is, but if you need to figure out what's going wrong it will be very hard. And if you need to make any changes, it will be harder still. Suppose you have some strange relatives who want their cake to have an orange-apricot glaze instead of chocolate frosting (I did mention they were strange). Where do you go in and make the changes?

Instead of one big list, suppose we had broken it up into two steps like this, where each one lists the ingredients and steps for just that part of the cake-making process:

- `makeChocolateCake`
- `makeVanillaFrosting`

Oh, now it's easy to see. If there's a problem with the cake, you know where to look. If you want to do a different icing, you can easily change it to this:

- `makeChocolateCake`
- `makeOrangeApricotGlaze`

That's pretty much the idea behind functions. They are a way to gather instructions and data together into groups that make sense. But functions have an extra fun ability: you can use the same function (list of instructions) with slightly different data. For example, you could have one function named `makeFrosting` and call it with different flavorings:<sup>4</sup>

```
makeFrosting(flavor)
    sugar
    butter
    mix in "flavor"
    spread on cake
```

Then you could use that same function, passing in slightly different data as needed:

```
makeFrosting(vanilla)
makeFrosting(chocolate)
```

That's why we use functions: to make long lists of instructions (code) easier to read and understand, and to reuse sets of instructions with slightly different data.

You could say functions make programming a piece a cake. But back to Minecraft.

## Defining Functions in Java

Every bit of code we write in Java will be in a function; that's how Java works. We've seen functions already, right from the very first plugin.

Back in the `HelloWorld` plugin, we declared a bunch of functions that Minecraft calls when the game is running: the short ones for `onEnable` and `onDisable`, and the main one for `onCommand`.

---

4. When you write out an idea that's code-like but isn't really a programming language, we call it *pseudo-code*. Just in case you see that term somewhere, now you know what it means.

We call these particular functions in a plugin the *entry points*. These are the functions that the Minecraft server will call when it needs to. You can provide code, if necessary, for all of these—or just for some, depending on your plugin.

<b>onLoad</b>	Called when the server loads up the plugin, but before it's enabled
<b>onEnable</b>	Called when the server enables this plugin
<b>onDisable</b>	Called when the server disables this plugin or shuts down
<b>onCommand</b>	Called when the user types in a command in the Minecraft chat with a slash, "/"

In our usual `onCommand`, we're calling other functions. Here's the section from `HelloWorld`:

```
public boolean onCommand(CommandSender sender, Command command,
                        String commandLabel, String[] args) {
    if (commandLabel.equalsIgnoreCase("hello")) {
        String msg = "[Server] That'sss a very niccce EVERYTHING you have there...";
        getServer().broadcastMessage(msg);
        return true;
    }
    return false;
}
```

There's a call to `equalsIgnoreCase()`, a call to `getServer()`, and a call to `broadcastMessage()`.

Java knows you're calling a function because of the parentheses after the name of the function. It will expect that someone defined a function based on the name and it will give that function your message. We call the stuff you pass to functions *arguments*. When arguments are given to a function, the function knows them as *parameters*. We say the values are *passed in* or the function is *called with* these values. All these words and phrases are referring to the same concept.

For example, the `getServer()` function doesn't take any arguments. You still use the parentheses characters, ( and ), so that Java knows it's a function. That `getServer()` call returns something (I'm guessing it's a `Server`), and we're calling the `Server`'s `broadcastMessage()` function, passing in a string argument named `msg`. With me so far?

You can define a function yourself. Here's an example that defines a new function named `castIntoBlackHole`. Watch closely, because you'll be doing this on your own next.

```
public static void castIntoBlackHole(String playerName)
{
    // Do something interesting with the player here...
}
```

There is a bit more noise here than in the cake example. Let's see what all this stuff means.

- *public* means that any other part of the program can use it, which for now you want to be the case.
- *static* means you can call this function all by itself (not like a plugin; we'll see the difference and what that means in the next chapter).
- *void* means this function is going to run a couple of instructions, but not give you any data back—it won't "return" any values to the caller.
- *castIntoBlackHole* is a name we just made up; it is the name of the function, and the () characters indicate that it is a function and will take the arguments we've listed. You always need the parentheses, even if the function doesn't take any arguments.

In this case, it takes one argument we named *playerName*, which it expects to be a String. For each argument your function accepts, you need to specify both a variable name and its type. Your function can take multiple arguments; you use a comma to separate each pair made up of the type and variable (like we did back in the `onCommand` in `HelloWorld`).

The braces, { }, are where the code for this function goes. You can put as much code in a function as you want, but a good rule of thumb is to not make it any longer than maybe 30 lines. Shorter is always better; if you find yourself writing very long functions, you will want to break those up into several smaller functions to help make the code easier to read.

Here's an example of a function that returns a value; it will triple any number you give it:

```
public static double multiplyByThree(double amt)
{
    double result = amt * 3.0;

    return result;
}
```

This function calculates a result and uses the `return` keyword to return that value to the caller. You would call the `multiplyByThree` function and assign the returned value to variables like this:

```
double myResult = multiplyByThree(10.0);

double myOtherResult = multiplyByThree(1.25);
```

Now `myResult` will be 30.0, and `myOtherResult` will be 3.75.

## Try This Yourself

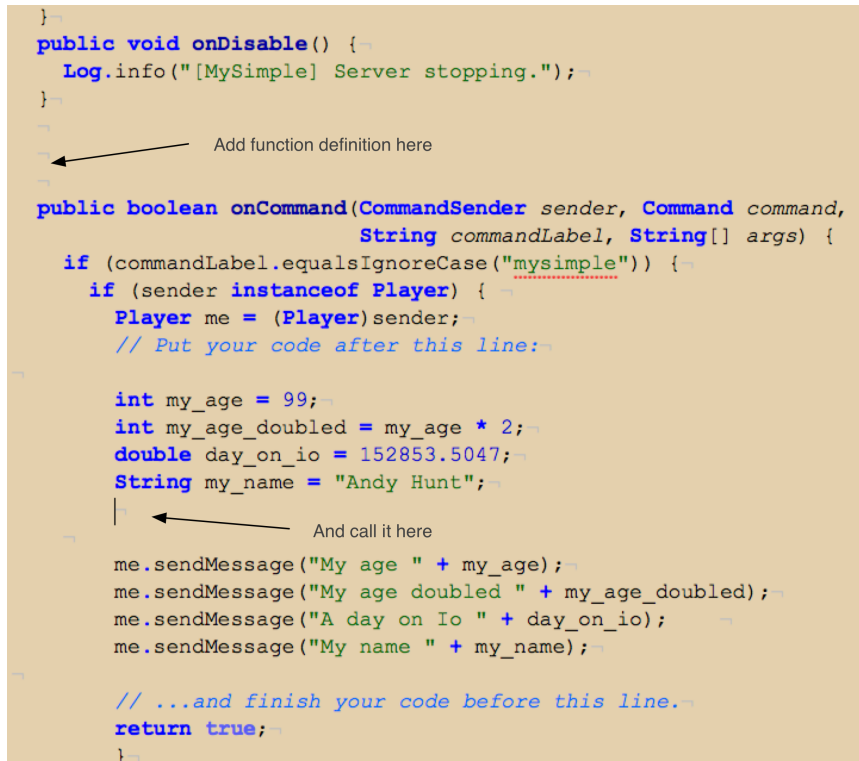
You're going to write a function named `howlong()` to calculate how many seconds you've been alive:

```
public static int howlong(int years) {
    // Write this function...
}
```

The function will take a number of years and return a number of seconds. We'll cheat a bit to make this easy, and convert years to seconds. (See the footnote if you need a hint.)<sup>5</sup>

You'll add this new function to the Simple plugin, and call the function to print out its value just like we did with your name and age.

Define the function where the top arrow is pointing:



```

}
public void onDisable() {
    Log.info("[MySimple] Server stopping.");
}

// Add function definition here

public boolean onCommand(CommandSender sender, Command command,
    String commandLabel, String[] args) {
    if (commandLabel.equalsIgnoreCase("mysimple")) {
        if (sender instanceof Player) {
            Player me = (Player)sender;
            // Put your code after this line:

            int my_age = 99;
            int my_age_doubled = my_age * 2;
            double day_on_io = 152853.5047;
            String my_name = "Andy Hunt";

            // And call it here

            me.sendMessage("My age " + my_age);
            me.sendMessage("My age doubled " + my_age_doubled);
            me.sendMessage("A day on Io " + day_on_io);
            me.sendMessage("My name " + my_name);

            // ...and finish your code before this line.
            return true;
        }
    }
}

```

5. In other words, multiply the number of years by the number of days in a year, multiplied by the number of hours in a day, multiplied by the number of minutes in an hour, and finally by the number of seconds in a minute.

And add the call to the function `howlong` down where my cursor is, at the second arrow. Assign it to an extra-big integer (a `long`) and pass in an age (I'll use 10 here) like this:

```
long secondsOld = howlong(10);
```

Then print it out to the player just like the rest of the `sendMessage()` calls do.

If I compile and install it with `./build.sh`, stop the server and restart it (or reload the server), and then run the `/simple` command in Minecraft, my test with 10 years gets me 315,360,000 seconds:

```
$ cd Desktop
$ cd Simple
$ ./build.sh
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```



Did you get the same answer? You can see the full source code that I put together at `code/Simple2/src/simple2/Simple2.java`.

Note that there are a couple of different ways to accomplish even this simple function. There usually isn't just one "correct" way to write code.

That's a good start, but there's more to Java than just variables and functions. The Java language has certain special *keywords* that you can use to direct how and when to run various bits of code. We've seen some of these already, including `public` and `static`, which describe the code. Now we'll look at keywords, including `if`, `for`, and `while`, that let you control how code is run.