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

Praise for This Book

A handful of boilerplate and about five lines of custom code, and suddenly, exploding arrows! All of a sudden, doing something cool in the context of a fully realized 3D game engine is very easy. And that's how programming has to be learned…in easy, bite-sized chunks.

➤ **Carl Cravens**
Linux system architect and Minecraft dad

Learn to Program with Minecraft Plugins explains things very well—no programming experience required. It's very helpful for new programmers. And so far, it's been an excellent vehicle for some quality father-son bonding.

➤ **Mel Riffe, Minecraft dad, and Noah Riffe, age 12**

Phenomenal. Approachable and simple, without talking down to the audience. I could see anyone at any age reading this.

➤ **David Bock, age 44**

Well, first off, this is a wonderful book. The way it is written is amazing. I am learning BASH and Java! I've tried to learn BASH, but all the other books I've found are just way too hard. Not only that, but modding Minecraft while running a server? EPIC! Thank you for writing this book!

➤ **Jack H. Age 13**

I really liked making the server plugins. My favorite was the cow shooter.

➤ **Jonathan Knowles, age 13**

Go, you—this book is awesome!

➤ **Stina Qvarnström**
Developer, Bool Noridc AB, Sweden

Sssssssssss

➤ **A creeper**

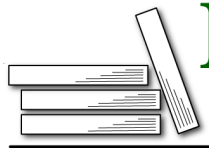
Learn to Program with Minecraft Plugins, 2nd Edition

Create Flaming Cows in Java Using CanaryMod

Andy Hunt

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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)
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

Contents

Acknowledgments

A very special thanks to my son Stuart for suggesting this book and answering a lot of my dumb questions about Minecraft, and to the rest of my family for putting up with me as I disappeared under headphones and typed away in an imaginary world.

Thanks to my editor Brian Hogan, managing editor Susannah Pfalzer, and production manager Janet Furlow, and everyone at the Pragmatic Bookshelf for helping to get this second edition finished in record time.

Thanks to second edition tech reviewers, including Said Eloudrhiri, Ingo Haumann, Jack H., Dan Kacenjar, Andrés N. Robalino, and Zachary Thomas. Extra special thanks to Joshua McKinnon for his detailed and thoughtful review.

Special thanks to the crew at CanaryMod, especially Jason Jones, for their support.

Start Here

Welcome!

Thanks for taking the time to pick up this book. I hope you'll find it a quick read and have a lot of fun along the way. If you've never written a program before, don't worry. We'll take it slow and start at the very beginning. No experience required.

Everyone loves Minecraft. I think a big reason for its success is that you get to participate in making the game. You get a chance to build; to create. Whether it's a quick shelter in survival mode or a huge Redstone simulation or your very own castle, you get to create.

But sometimes the Minecraft game's built-in capabilities aren't enough. You want to do something more. You want to shoot flaming cows or encase an opponent in a cage of solid rock. For these and many other extra abilities, you need to add features to the game itself.

Applications on your computer or phone are written in a special kind of text we call *programming languages*. They're not as huge or hard to learn as natural languages, such as English or Spanish or Chinese, but they are different from the language you use to write and talk in every day.

There are many, many programming languages in use today. Some are very popular and not very powerful. Some are used only by a handful of people but are incredibly powerful and difficult to master.

Minecraft is written in the Java programming language. Java is moderately powerful, but it also has some hard and confusing parts. We'll focus on the basics and sidestep the difficult stuff.

This is a fast-and-loose book to get you programming in Java quickly. You'll learn enough of the Java programming language to create your own Minecraft plugins and accomplish common tasks in Java.

We'll take a look at setting up your own Minecraft server, keeping cloud-based backups of the code you write, and we'll take a peek at a few advanced coding techniques.

Who This Book Is For

This book is aimed at readers who have no experience programming, but who do have some experience playing Minecraft. If you aren't familiar with Minecraft, there are plenty of videos and books to help you get started. But I'm guessing you're already pretty adept at and enthusiastic about Minecraft, and you want to learn the programming end.

Readers younger than 8 or 9, or readers of any age who are having trouble understanding programming in Java, might want to take a short detour and experiment with a friendlier programming language first. Scratch and the newer version, Snap!,[1,2] are great little languages that help you learn the basic concepts of programming. They show how programming elements fit together visually. Once you get the hang of that, then coming back to a text-based language like Java can make a lot more sense.

Otherwise, you just need a modern computer running Windows, Mac OS X, or Linux, and we'll go from there.

Getting Started

Minecraft is designed as a "client-server" application. That means it's split into two parts.

First, there's the client, which is the application you run on your desktop or laptop computer. The client renders images of the Minecraft world and accepts your commands to move and act in the game.

Second, there's the server, which keeps track of everything in the game, including all the players who are connected, their inventories, what they've built, where they are, and so on. Most of the time the server is running on some faraway machine in another part of the country. But it could be running on your laptop or desktop computer as well.

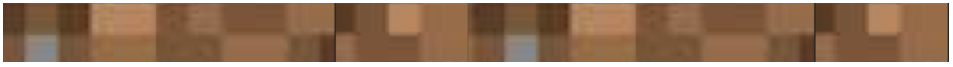
The client and the server talk to each over the network, the same way you use a web browser to connect to servers and play games or see pictures of cats.

1. http://scratch.mit.edu
2. http://snap.berkeley.edu

To add or change functionality in the Minecraft game, you have to add to or change the Java program on the server. That's what you'll learn to do in this short book: program in Java by writing Java instructions (which we call "source code," or just "code" or the "program") to create *plugins* for the Minecraft server. A plugin is just a piece of code that you add to an existing program.[3]

Before we get started with plugins, you need to set up a local Minecraft server for testing, and install the Java programming language and a couple of other applications. We'll go over all of that in the first few chapters. Installing stuff isn't very much fun by itself. In fact, it's boring as dirt. But we'll try to get through the dirt pile as fast as we can.

In fact, to help you keep track of your progress, a "Your Growing Toolbox" sidebar at the end of each chapter shows a progress bar. You'll start from just plain dirt:



And finish up with 100% grass:

Some chapters will go faster than others and some will add more than others, but you *will* keep making progress.

Swimming in the Deep End

Because we've got a lot to cover in a small space, I'm going to try to show you things first, maybe even use them first, and then explain them in detail a little later. Sometimes that can feel like you've been thrown into the deep end of the pool. When you see something that doesn't make sense yet, don't worry about it. Just let it wash over you; the explanation will be along shortly.

In many cases, you can use something successfully even if you don't really understand how it all works. I can turn a desk lamp on and use it without understanding how electricity is generated. I could even build my own desk lamp without understanding how to build a big power-station generator. I just need to know how my part fits in.

We'll focus on how your part fits in as much as possible.

3. Some folks also write Minecraft *mods*, which take place in the graphical client, but we won't cover those here.

Getting Help

There's a forum on the book's website for questions, updates, and tips. Just point your browser to https://pragprog.com/book/ahmine2 and click on the Discuss tab.

On that same home page, there's a link for Source Code where you can download all the source code from this book.[4]

Please download that to your Desktop now—you'll use your computer's Desktop for most of our work, but more on that in a bit. Meanwhile, start downloading.

I'll wait.

The download is an archive file created using zip, so you need to unzip it on your Desktop. You can use unzip from the command line (for Mac OS X or Linux), or for Windows you can use WinZip or the free 7-Zip.[5]

Got it unpacked? Great!

As we go along, you'll learn how to use new tools and learn new ways to use those tools. We'll keep track of the new stuff you learn with that "toolbox" at the end of each chapter. By the end of the book you'll have enough in your toolbox that you'll be able to design and code your own plugins from scratch!

Conventions

Code or commands that I'm showing you as examples look like this:

```
$ I've typed all of this as an example for you.
```

For code or commands that you're supposed to type in, I'll show it with a shaded background like this:

```
$ you type this part here (but not the dollar-sign prompt)
```

Things in *italics* are placeholders; you don't type them in directly, so something like this:

```
me.chat( string msg );
```

means you would replace the italic part, like this:

```
me.chat("Creepers are coming.");
```

Now let's see how to work this thing.

4. The exact link is https://media.pragprog.com/titles/ahmine2/code/ahmine2-code.zip.
5. Available from http://www.winzip.com or http://www.7-zip.org

In this chapter you'll learn about the shell, where you can type commands to your computer. You'll add these topics to your mental toolbox:

- How to open a command shell and type commands
- How files and directories make up the file system
- How to navigate around directories in the file system

CHAPTER 1

Command Your Computer

One of the earliest and greatest computer games that created a world for you to explore was Colossal Cave Adventure,[1] way back in 1976. It was a purely text-based adventure—there were no images or graphics. You typed instructions to the game using simple sentences, commanding it to "go north" or "take axe" or "kill troll" as needed. And it did as you asked, even killing the troll with your bare hands.

Today, text commands are still in games—even Minecraft has text commands. You've probably typed commands in Minecraft's chat window using a "/" character.

You're going to issue commands to your computer to build plugins and work with files in very much the same way, using the *command line*.

The command line is a powerful tool that lets you work on your local computer as well as on distant computers in the cloud. In fact, we'll cover how to do exactly that later on in the book, in Appendix 4, *How to Install a Cloud Server*, on page 229.

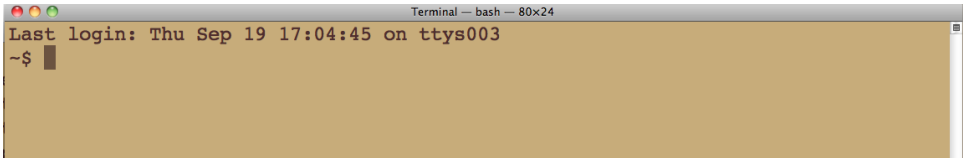
You can even use the command-line processor to write programs; it contains a full programming language by itself, separate from Java. I've done a little of that for you, with a *script* (an executable list of commands) that helps you build and install plugins. We'll use that as we go along.

If you're already familiar with using the command line, feel free to skip to the end of this chapter on page 12.

1. http://en.wikipedia.org/wiki/Colossal_Cave_Adventure

Use the Command Line

The command line looks something like this on my computer. Yours may use different colors and fonts, and you can usually set these to your liking. I apparently prefer black letters on a tan background:



You access the command line a little differently depending on the kind of operating system you're running, which will be one of these:

Windows comes with a very bare-bones command line. You get to it by running `cmd.exe`. If your version of Windows has a "Start" command, then you might be able to select Start -> Run and enter `cmd.exe` or just use the search box that comes up to find and run `cmd.exe`. But I don't recommend using `cmd.exe` by itself; see the instructions in the box on page 4.

On Mac OS X, open the command line via Applications -> Utilities -> Terminal.

If you're running Linux, you likely already know how to get to a command-line shell. But for the sake of completeness, and because it's called different things, try any of these: open a shell, start a Konsole, or right-click on the Desktop to open a Terminal.

(Fortunately, all of these pesky differences between Windows, Mac, and Linux disappear once you're writing Java code: Java runs the same on each platform.)

Once you have your command-line application up and running, you're ready to type commands into the command-line processor, or *shell*, as I'll call it. You'll be using a few simple commands that I'll show you as we go along.

Each shell prints out a short message indicating it's ready for you to type something in. But instead of a straightforward prompt like "Ready for you to type, master," most command prompts are a little more cryptic.

Windows will show something like `C:\>`. Linux and Mac systems might show `$` or `%`. Any of these prompts might include additional information, like your name, the computer's name, or a directory name. Since that will be different for everyone, I'm going to choose the simplest one for the examples in this book and show the command prompt as a `$`. Whatever your prompt looks like, that's where you type in commands.

DO NOT TYPE IN THE DOLLAR SIGN ($). I'll show it to indicate where you type, but you *don't* type the $:

```
$ you type this part here, but not the dollar sign
```

When you're done typing in a command and want the computer to run it, press the Enter (or Return) key on your keyboard. That's the easy part. Next you just need to learn a few basic commands to type in!

You can list the files and directories with the *directory listing* command:

```
$ ls
```

To change your current directory, you use the *change directory* command. For example, to change the directory to the Desktop, you'd type

```
$ cd Desktop
```

and be transported to the Desktop (or any other directory whose name you enter).

For instance, when your shell first opens up, you'll be in some sort of default directory or folder (sometimes called a *home* directory). You can see what files are there by typing the command `ls` (short for "list files") and pressing Return. (On stock Windows using `cmd.exe`, you'd have to type `dir` (short for "directory listing") instead. Also, while the rest of the world uses a "/" in directory names, Windows uses a "\." These and other tiresome differences are why I recommend using the POSIX-standard bash shell for Windows as described in the box on page 4.)

You'll see a listing of a bunch of files. Type `cd Desktop`, and you'll be in your Desktop directory. Type the command `ls`, and you'll see a list of all the files on the Desktop. Now you're working at the command line! Let's delve into that a little bit more.

Move Around in File Directories

Normally when you want to look at files on your computer's hard drive, you use graphical programs like Explorer on Windows or the Finder on Mac. Either way, the idea is to show you the files, folders/directories, and applications/programs on the computer so you can navigate around and run things.

We're doing the same thing here, but in a more powerful way and without graphics. If you're already familiar with doing this, please feel free to skip to the end of this chapter for a small treat. Otherwise, read on.

Install BusyBox on Windows

The Mac OS X and Linux environments are based on Unix, which has a very rich command-line environment, including a full-featured shell. There's a published standard for that environment, called POSIX, which includes commands and language features. The POSIX standard shell is really an entire programming language by itself, and you can write shell scripts to do basic tasks on the computer for you.

Windows, however, isn't as sophisticated. The default command processor doesn't do much at all, the commands have different names, and directory and file names are specified differently.

So in order to keep things consistent for everybody, I recommend that Windows users download BusyBox. It installs a more professional, POSIX-compliant standard shell and common commands—the same ones that Mac OS X, Linux, and the rest of the world uses.

To install the command shell and basic commands for Windows, download this file to your Desktop: ftp://ftp.tigress.co.uk/public/gpl/6.0.0/busybox/busybox.exe

Once it's downloaded to your Desktop, rename the file from busybox.exe to sh.exe. Then open a cmd.exe window and type this:

```
C:\> cd Desktop
C:\> C:\Windows\system32\cmd.exe /c sh.exe -l
```

Now you'll see a new shell with a dollar-sign prompt. That's where we'll run commands and do our work. Notice an important detail: this is running sh.exe with a -l flag (which you would pronounce as "minus ell"), which tells it to act like a "login shell" where you can type commands.

For convenience's sake, you can make a batch file on your Desktop to launch this command shell for you. To do that, create a text file and save it to your Desktop. Name the file shell.bat and type this line of text in the file:

```
C:\Windows\system32\cmd.exe /c sh.exe -l
```

Save the file, and now you can double-click on shell.bat, and you'll have a POSIX-compatible shell. With that done, you can access the commands we'll use (things like ls, mv, cp, and pwd), and you'll specify directory names with "/" instead of the Windows "\" character, so everything will work the same on Windows as on Mac and Linux.

You'll also be able to employ the command scripts that we'll use to build and install plugins.

The Busybox home page at http://intgat.tigress.co.uk/rmy/busybox/index.html has more information on BusyBox for Windows.

The collection of files and folders on your computer is called the *file system*. At any point in time a Finder (or Explorer) window on your Desktop or in your command shell is sitting in some current directory. On your Desktop you might have several folder windows open, each looking at a different folder (directory) on your disk.

Each command-line shell window you have open works the same way: for each window, there is a *current directory*. Some systems are set up to show the current directory in your prompt. But you can always find out where you are by typing the command `pwd` (which stands for "print working directory"):

```
$ pwd
/Users/andy/Desktop
```

Try this now: open a fresh shell, and before doing anything else, type `pwd` at the prompt (shown here as a `$`—yours may be different):

```
$ pwd
```

That will print out your *home directory*, which is where every one of your shells will start from.

In each shell, all the commands you run will run with this particular idea of a current directory (or current working directory). Many programs you use will look here, in the current directory, to run, open, and save files.

If you haven't yet, download to the Desktop the files that came with this book,[2] and unzip the archive there.[3] That will unpack to a directory named `code`, which contains all the examples from this book.

Under `code` there are a bunch of plugin directories, one for each plugin in this book. We'll start off looking at the `HelloWorld` plugin files. Under that directory are a few other files and subdirectories.

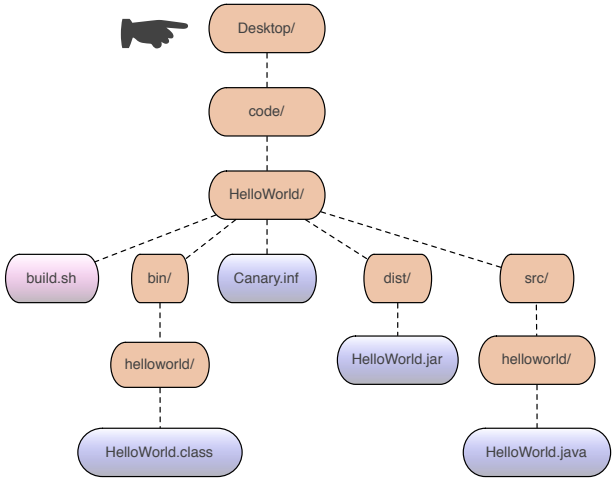
The diagrams on the next page show how you can navigate through these directories.

2. http://media.pragprog.com/titles/ahmine2/code/ahmine2-code.zip
3. Use `unzip` at the command line, or on Windows use WinZip or 7-Zip.

Start with this:

```
$ cd Desktop
```

And you'll be here:

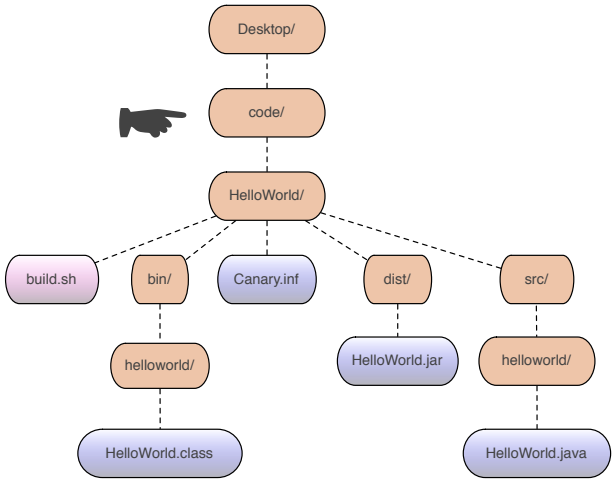


(If that didn't work, try `cd ~/Desktop`, or check the end of this chapter for hints.)

Next go down into the code directory by typing this:

```
$ cd code
```

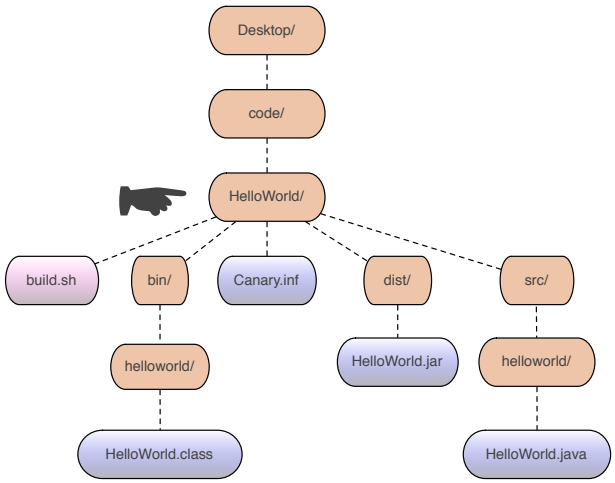
Now you're here:



Go down into HelloWorld:

```
$ cd HelloWorld
```

And now you're here:



Now list the files there. You'll see this:

```
$ ls
Canary.inf  bin/  build.sh  dist/  src/
```

My system (Mac OS X) is configured to show directories with a slash at the end. (If yours isn't, try typing ls -F.) So here I have two files and three directories in my current directory. File types are often represented by the last part of the file name—its suffix. Here I have a shell script with an .sh suffix, and a config file with an .inf suffix. Down in the src/ directory there's a helloworld subdirectory; there I have a Java source file with a .java suffix (more about these kinds of files as we go along).

To go down into the src directory, type cd src, and you're there.

```
$ cd src
$ ls
helloworld
```

Then down one more into helloworld:

```
$ cd helloworld
$ ls
HelloWorld.java
```

In the src/helloworld directory, there's the HelloWorld.java file—the "guts" of our first plugin.

But now you're down in HelloWorld/src/helloworld. How do you get back up to HelloWorld? To go up just one level, you'd type this:

```
$ cd ..
```

Or to go up two levels, type this:

```
$ cd ../..
```

That's two periods (or dots), which is pronounced "up one." One dot by itself means the current directory, which isn't very useful with cd, but we will use "." with other commands—especially when copying files.

But now suppose you need to visit a directory that isn't just in the current directory or up one. How do you get there? Suppose I'm somewhere completely different, like /home/minecraft, and I want to go to /Users/andy/Desktop/code/HelloWorld.

I'd do something like this:

```
$ cd /Users/andy/Desktop/code/HelloWorld
```

The leading slash makes the difference. Earlier, when you typed cd src, the cd command looked for src right under the current directory. But if instead you typed cd /src it would look for a directory named src under a directory named "/"—which we call the *root*.

Root is the topmost directory on your system. It's above your code, above your Desktop, above everything. Somewhere under root are your home directory and Desktop. In my case, that's /Users/andy/Desktop. I could get there the slow way by typing this sequence of commands:

```
$ cd /
$ cd Users
$ cd andy
$ cd Desktop
```

But we'll see a much easier way in just a moment. And speaking of shortcuts, you don't even need to spell out each directory name fully, like D-e-s-k-t-o-p.

On most systems, there's a nice keyboard shortcut to save you from typing out long names—the Tab key. If you type in the first few letters of a long name and then press the Tab key, it will autocomplete to the long name. Suppose I'm in my code directory:

```
$ ls
Adventure          CreeperCow     LocationSnapshot  SquidBomb
ArrayAddMoreBlocks EZPlugin       MySimple          SquidBombConfig
ArrayOfBlocks      FireBow        NameCow           Stuck
BackCmd            FlyingCreeper  NamedSigns        install
BackCmdSave        HashPlay       PlayerStuff       mkplugin.sh
```

```
BuildAHouse         HashPlayClamp  PortingGuide.txt  numbers
CakeTower           HelloWorld     Simple            runtime
CanaryLinks.txt     LavaVision     Simple2
CowShooter          ListPlay       SkyCmd
```

If I type `cd Ad` Tab (or enough letters to be unique) it will autocomplete to

```
$ cd Adventure/
```

and I can just press Return. For short names it might not look like much, but if you have a long directory name like RumpelstiltskinReincarnationSpellPlugin, typing `Ru` Tab begins to look mighty appealing.

Copy and Paste

Sometimes you might want to copy a bit of text and paste it at the command line. For instance, you might want to copy a line from this book and paste it in.

Copy and paste at the command line can be a little different from copying in an application like Mail or a web browser. You still click and drag the mouse to select text to begin with.

On Linux, you can use Ctrl-C to copy and Ctrl-V to paste in most applications. At the command line, you may need to add a Shift key, so copy would be Ctrl-Shift-C.

On Mac you use ⌘C and ⌘V to copy and paste.

On Windows the command-line window is slightly different.

First, you need to enable QuickEdit. Right-click the top bar of the command window and select Properties. On the Options tab, in the Edit Options section, check QuickEdit Mode to turn it on.

To use: after selecting text, you need to press Enter to copy, then right-click or Ctrl-V to paste.

That's just for the command prompt window. Everywhere else on the system (your editor, etc.) it's Ctrl-C and Ctrl-V as usual.

Try This Yourself

Let's make some directories and files using the command line. You're going to make your own copy of a plugin, with directories and all.

Start off in your Desktop directory (make sure you're there by typing `pwd`) and make a new directory called `myplugins` using the `mkdir` command.

```
$ cd Desktop
$ pwd
/Users/andy/Desktop
$ mkdir myplugins
```

List out the files on your Desktop (using ls), and you'll see all the files you have there as well as the new directory, myplugins. Let's go down into myplugins and do some work.

```
$ cd myplugins
```

Do a pwd to confirm you're in the myplugins directory.

If you do an ls here you won't see anything—we haven't made any files there yet. Let's fix that by making the directory structure, which will be the same as for the HelloWorld plugin. Start by making a directory named for the plugin itself:

```
$ mkdir HelloWorld
```

And (you guessed it!) cd down into HelloWorld.

```
$ cd HelloWorld
```

Now you can make a few directories that you'll need: src, src/helloworld, bin, and dist. Go ahead make those directories here now:

```
$ mkdir src
$ mkdir src/helloworld
$ mkdir bin
$ mkdir dist
```

Use ls to make sure they are there.

```
$ ls
bin/ dist/ src/
```

```
$ ls src
helloworld/
```

Now you need three files here, which you can copy from the book's example code. You can drag and drop using your regular graphical windows, or use the copy command, cp:

```
$ cp ~/Desktop/code/HelloWorld/build.sh .
```

The tilde character (~) is shorthand for "my home directory." And hey—we got to use the single dot! The whole command line means "copy this file to the current directory."

You'll need this file too, so copy that over while you're here.

```
$ cp ~/Desktop/code/HelloWorld/Canary.inf .
```

Now you've created the directories and supporting files that you'll need for a plugin.

Files on Your Desktop

We'll do all of our work on the Desktop because that's the easiest place for you to find files, and it's the same across Windows, Mac, and Linux.

Out in the real world you probably wouldn't want to clutter up your Desktop with each new project you work on. But until you get more comfortable moving things around and setting things up, stick to the Desktop.

If It Doesn't Work

One area where you might run into problems is if your home directory contains spaces. For instance, if you're on Windows and your name is "John Smith," typing in a command using the tilde, like this:

```
$ cp ~/Desktop/code/HelloWorld/build.sh .
```

makes it look like you typed this:

```
$ cp C:/Users/John Smith/Desktop/code/HelloWorld/build.sh .
```

The computer interprets that as saying "copy C:/Users/John and Smith/Desktop/code/HelloWorld/build.sh" to the current directory. You'll get the error that there is "no such file or directory."

You have two workarounds: you can use a relative path, by typing ".." for parent directories, so you go "up two" and down into code:

```
$ cp ../../code/HelloWorld/build.sh .
```

Or just type it in by hand, using quotation marks around the file name:

```
$ cp "C:/Users/John Smith/Desktop/code/HelloWorld/build.sh" .
```

Another problem you might run into is not being in the directory that you think you are. When in doubt, you can always do a pwd command to print your current working directory:

```
$ pwd
/Users/andy/Desktop
```

Here I'm in my Desktop directory, which is where we'll be starting off for most of our work.

Start at the Desktop

On most systems, you should be able to type cd Desktop to get to your Desktop. If that doesn't work, you may need to type cd ~/Desktop—using the tilde shortcut—or you might need to spell the whole thing out, as in cd /Users/andy/Desktop.

However you accomplish it, when I say "start at your Desktop" or just `cd Desktop`, that's what you'll always need to do, whether it's from anywhere, like this:

```
$ cd Desktop
```

from your home directory first:

```
$ cd
$ cd Desktop
```

using a tilde for the home directory:

```
$ cd ~/Desktop
```

explicitly typing the name of your home directory:

```
$ cd /Users/andy/Desktop
```

or doing that with quotes because you have spaces in the name:

```
$ cd /Users/"John Smith"/Desktop
```

No matter what, it will always be shown here as just

```
$ cd Desktop
```

And Now for Some Fun

In this book's downloaded code, in your `Desktop/code` directory, there's a special subdirectory named `Adventure`. Using the command line, `cd` there and have a look at those files and directories.

You can use `ls` to list the directories as we've done here. To take a quick look at text files (named `.txt`), you can use the `cat` command.

Start at your Desktop (however that works for you, as described in the last section).

```
$ cd Desktop
$ cd code
$ cd Adventure
$ cat README.txt
These are some files to make exploring the file system a little more fun.
```

Do an `ls` and see what else is there and explore a bit in the subdirectories. See what treasures—and what dangers—you find.

Common Commands

Here are some of the most common commands you'll use at the shell:

java	Run Java classes and Java archives (jars) as a program
javac	Compile Java source code to class files
cd	Change the directory
pwd	Print the working (current) directory
ls	List files in the current directory
cat	Display the contents of a file
echo	Display text; also display environment variables using the $ prefix
mkdir	Make a new directory
cp	Copy a file
mv	Move a file
rm	Remove (delete) a file permanently (Use this with extreme caution; this is not the same as the Trash, and there is no "undo.")
chmod	Change file permissions (including read, write, and execute)
.	(single dot) Means the current directory
..	(two dots) Means the parent directory
~	(tilde) Means your home directory

Next Up

Next you need a way to type in the Java source code. You'll need Java itself, and an application to put it all together for you. We'll install all of that in the next chapter, and then get to building plugins.

Your Growing Toolbox

You now know how to:

- Use the command-line shell

In this chapter you'll add several new applications to your computer, and add these tools to your toolbox:

- The Java language compiler, javac, and the runtime application, java
- The Minecraft graphical client
- The Minecraft server that we'll be modifying

CHAPTER 2

Add an Editor and Java

To write programs in Java, you need something to write with: some way of editing Java text files. While you could use a bare-bones text editor like Notepad or TextEdit, that's a really painful way to do it.

And you can't use Microsoft Word or another office-style word processor. Those applications aren't designed for programming, and they don't store files in a format that Java can use—Word files are filled with fonts, colors, sizes, and all sorts of formatting information.

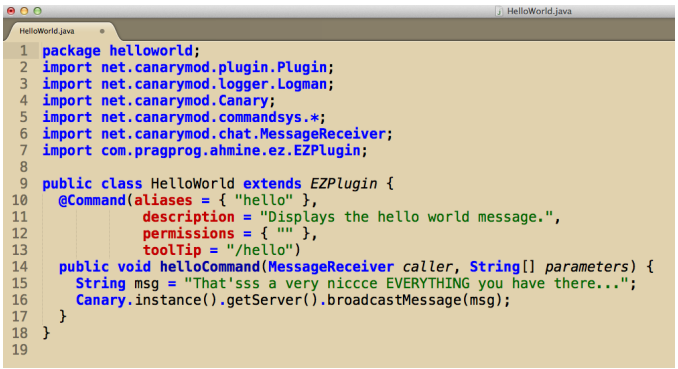
What you need is a text editor that's designed for programming. I've got a good one for you, which we'll see next. You'll also need to install Java to build and run plugins, and you'll need Minecraft, of course. With all that installed, you'll be set to build a plugin in the next chapter.

This chapter may contain some new (and possibly cryptic) commands and potentially confusing concepts. It's okay if you don't know what these mean and don't understand fully right now; these aren't things you'll run into day-to-day when writing plugins; they're just a few necessary evils you need to get everything installed.

Let's start with the editor.

Install an Editor to Write Code

You write plugins and programs by typing in kinda-English words and some funny punctuation. Most things you write will be longer than a text message but shorter than an essay in English class. You need a proper editor to type in your programs. My suggestion is an editor called *Sublime Text*, shown in the following screenshot.



Sublime Text is available as a download and runs on Windows, Mac, and Linux.[1] You can try the free evaluation (there's no time limit, but it will nag you every so often) or cough up some bucks and purchase it. If you have a different editor that you read about or prefer, that's fine—you can use that. On Windows, for example, you might want to look at the free editor Notepad++.[2] The choice of an editor is very much a personal one. There is no "right" answer, size, shape, or color. But you do need an editor that has the right features.

One great advantage of an editor built for programmers is a feature called *syntax highlighting*. The editor knows about different parts of the Java language and will make language elements like functions, variables, strings, and keywords show up in different colors and fonts (as in the example in the figure). That kind of visual support can be really helpful when you're first learning what all those different things mean. We'll get to exploring all of that in a chapter or two, but first, choose your soon-to-be-favorite editor and install it. (You might also look into using IntelliJ IDEA or Eclipse. These are full development environments that include an editor and build support, but they can be complicated and hard to manage for beginners.)

So fire up Sublime Text (or some other suitable code editor) and give it a whirl. If you've used anything similar to Word, this will seem at least a little familiar: there's a menu bar at the top, and under the File menu you'll see useful choices like New, Open, and Save.

Go ahead and type in some text. Maybe write a two-line short horror story or something ("the reflection in the mirror blinked"). Click and drag to select text, and press Delete to remove it. For now it's just a plain old text editor, so we don't need to try anything too fancy.

1. http://www.sublimetext.com
2. http://notepad-plus-plus.org/download/

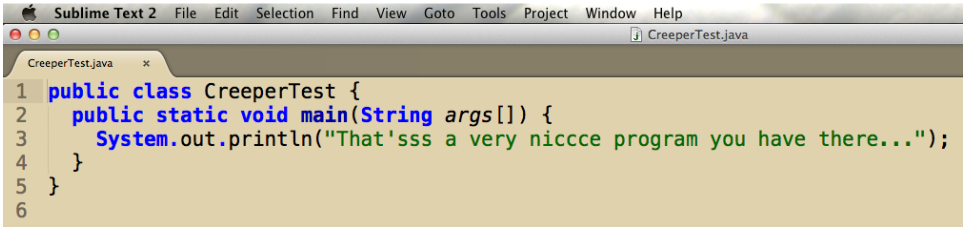
Try This Yourself

Let's make sure you can create a file, edit it, and save it. You're not quite ready for a full plugin yet, so let's start with a simple test file.

In the editor's menu, start with File -> New File to create an empty text window that you can type into.

Before you start typing, save the file by name. Select the menu item File -> Save As and save the file as CreeperTest.java on your Desktop. Just like that—with a capital C and T, and lowercase other letters. Now that you've saved the file, your editor will know you're going to type in Java code.

Type in the following text, just like I did:



Be sure to copy the text *exactly*, starting with "public class" through to the last "}"—including all punctuation, spelling, capitalization, and spacing—but not the line numbers, since those are part of the editor. Use the Delete key (sometimes labeled "backspace") to erase any mistakes.

Now select File -> Save to save the file with your text.

Ta-da! Now you have a file with Java source code.

And the computer has no idea what to do with it. It's just a file of text; as far as your computer knows, it could be a recipe for snacks or an essay or a list of game cheats. We need to fix that next.

What you need now is Java itself.

Windows Paths

A reminder: when we're using the bash shell from BusyBox, all directories use a forward slash ("/"). But the rest of the Windows system will still use backslashes ("\"). So what Windows and the installation programs call this:

```
C:\Users\yourname\Desktop\server
```

we'll call C:/Users/yourname/Desktop/server

Install the Java Programming Language

"Java" isn't just one application—it's actually several. There's the Java compiler, javac, the Java program runner, java, and the archiver utility, jar. javac takes your text file and creates a magical pile of gunk it calls a *class file*. You run the java command and tell it to use that pile of gunk, and your program springs to life. We'll look at that process in more detail as we go along, but that's the gist of it.

You might have Java installed already. Try running the Java compiler, javac, and see if it's there:

```
$ javac -version
javac 1.7.0
```

Yup, it's installed on my machine, version 1.7. If it was not installed, I'd get a message like this:

```
$ javac -version
bash: javac: command not found
```

You might also want to check that your java is indeed the same version as your javac (it should be, but things happen):

```
$ java -version
java version "1.7.0_67"
Java(TM) SE Runtime Environment (build 1.7.0_67-b01)
Java HotSpot(TM) 64-Bit Server VM (build 24.65-b04, mixed mode)
```

If you need to install Java, you'll want the Java Development Kit Standard Edition (JDK SE) version 7 or later, which you can download for different operating systems.[3] Depending on your operating system and vendor, you may or may not have Java installed yet, you might only have java and not javac, or you might have a slightly different version. Again, you'll need Java JDK 1.7 or later to work with the examples in this book.

No matter which operating system's installer you use, I heartily recommend you accept all the default answers to any installer questions, especially for the install location. Java and its associated programs can get more than a little quirky and fussy if things aren't where they expect.

So go download the Java Development Kit (JDK) installer and follow its instructions.

3. http://www.oracle.com/technetwork/java/javase/downloads/index.html

Depending on your platform, you might need to add the JDK's bin directory to your "path." We'll look at what that means and how you do it in the next section.

But first let's try a little exercise and see if it works.

Try This Yourself

With the JDK installed, you can now run that test program you typed in.

Again, this is just a simple program to make sure Java is happy. It's not a Minecraft plugin yet, but it's a start. As soon as you get this working, you'll be ready to start building Minecraft plugins.

cd to your Desktop and make sure the CreeperTest.java file is there:

```
$ cd Desktop
$ ls
CreeperTest.java
```

Now run the Java compiler on CreeperTest.java by executing the javac (Java compiler) command.

```
$ javac CreeperTest.java
$ ls
CreeperTest.class CreeperTest.java
```

The javac program first checks to make sure it understands everything you typed in the CreeperTest.java file. If there are no mistakes, then it creates the binary class file.

But the odds are that most of the time *there will be mistakes.* Don't worry if you get a ton of error messages: you didn't break your computer. Programming languages like Java are notoriously picky about capitalization, punctuation, and all the other things we type in that file.

Remain calm. (That, by the way, is generally good advice when working with computers.)

Try to decipher the message the computer gives you, and double-check all the code against our example from a moment ago. We'll walk through several likely problems next. There's also some advice in Appendix 1, *How to Read Error Messages*, on page 211.If all else fails, don't forget this book has a website where other folks are wrestling with this stuff too![4]

Once javac happily finishes its work without errors, you're ready to run.

4. This book's discussion forum is at https://forums.pragprog.com/forums/382.

You can call Java to run your program, giving it the name part only (not the .class suffix):

```
$ java CreeperTest
That'sss a very niccce program you have there...
```

Hey, it worked! Congratulations—you've compiled and run your first piece of code. Skip ahead to the Minecraft installation section that starts on page 23.

Otherwise, let's see what might have gone wrong.

If the Java Command Is Not Found

The most likely error you'll see is "javac not found" or "java not found," which means that even though you installed Java, your shell couldn't find the java.exe or javac.exe applications. Here's what's going on:

The commands you've used so far are either built in or were installed by the BusyBox installer. But when you install Java, the computer may not know where you put it.

When you go to type in the command javac, the computer needs to find an executable named "javac" (on Windows it's javac.exe; on Mac/Linux, it's just javac). There are a couple of standard places that it knows to look. On Windows, that might include a directory like C:\Windows\System32, and on Mac/Linux there could be several directories, like /usr/bin, /usr/local/bin, and so on.

Because the system has its own commands that you shouldn't mess with, and because you want to add your own commands to run, it turns out there are a lot of places the computer needs to look! You can tell it exactly where to look with a list of directory names. We call that your *search path*, or *path* for short.

You can see what's in the path for your shell window by typing this:

```
$ echo $PATH
```

For the shell/command-line processor to find a command to execute, it must be in a directory that's in your search path. So you'll need to add Java's bin directory to your path.

That directory's location depends on your operating system and what installer you used. A typical location for Windows would be C:\Program Files\Java\jdk1.7.0_67\bin (your version numbers may be a little different).

On Mac, it's probably installed in /usr/bin or /usr/local/bin, both of which are already in your path. But it might be installed someplace else completely, like in /opt/local/java.

In any case, once you've located the install directory for Java, you'll see a bin directory in there. In that bin directory you'll see java, javac, jar, and a whole bunch of other things. You'll need to add the full path for that bin directory to your shell's PATH.

The PATH is a list of directories, separated by a colon (semicolon on Windows). To change your PATH for the bash shell that we're using, follow these steps:[5]

1. In the shell, navigate to your home directory by typing cd, by itself. Confirm the full path of your home directory by typing pwd.
2. Using your text editor, create or edit the bash startup file in your home directory. Normally this will be a file named .profile or .bash_profile in your home directory (note the leading dot). On Windows using BusyBox, you have to use .profile. Otherwise you should use .bash_profile. ls won't normally show files with a leading dot, but ls -a will show it if the file already exists. You may need to create it from scratch, and that's okay.
3. Add a line to the file to modify the PATH setting, adding Java's bin directory, separated by a colon (:)—or a semicolon (;) on Windows.

For example, on Linux or Mac, if my JDK was installed in /opt/local/java, I'd add a line to .bash_profile that said

```
export PATH="$PATH:/opt/local/java/bin:"
```

On Windows, you need to change the backslashes to slashes and use semicolons instead of colons, so if Java's installed in C:\Program Files\Java\jdk1.7.0_67\bin (which is Windows style) you add a line to .profile that says this, in POSIX style:

```
export PATH="$PATH;C:/Program Files/Java/jdk1.7.0_67/bin;"
```

Save and close the file, then close and reopen your command-line windows to pick up the new settings.

Seriously—you have to close all your open command-line windows and reopen them for this to take effect.

To check your path and see if your new setting worked, type this:

```
$ echo $PATH
```

You should see your new entry that includes Java's bin.

5. If you run into trouble with this method, especially with Windows, take a look at http://www.java.com/en/download/help/path.xml.

Other Reasons It Might Not Work

Here are some other things that might go wrong even if the PATH is set correctly:

Make sure you are in the right directory; type ls and check that the file CreeperTest.java is right there.

Make sure you're typing javac CreeperTest.java (with the .java part). Otherwise you might see a truly confusing error message like this one:

```
error: Class names, 'CreeperTest', are only accepted if
annotation processing is explicitly requested
```

If the javac command reports some kind of "syntax" or "not found" or "not defined" error, that means it doesn't understand the text in the CreeperTest.java file, so you may have mistyped something. These kinds of errors might look something like this:

```
CreeperTest.java:1: class, interface, or enum expected
```

Or you might see some other error message. The number in between the colons (:1:) is the line number where the typo is located.

If you can't find the typo, grab a fresh copy of the file from this book's downloaded source code, at code/install/CreeperTest.java, and try that.

If the java command can't find CreeperTest.class, make sure the javac command ran okay and that it produced a .class file successfully. You should be in that same directory when running java.

If you see this error

```
Exception in thread "main" java.lang.NoClassDefFoundError: CreeperTest/class
```

you may have accidentally typed java CreeperTest.class (with the .class part at the end) instead of java CreeperTest (no suffix). To recap, these are the commands to compile and then run:

```
$ javac CreeperTest.java
$ java CreeperTest
```

That is, you must specify the .java suffix when compiling, but do not type in the .class part when running with java.

Also, check to see if there's a setting for CLASSPATH (which is just like PATH, but for Java classes).

```
$ echo $CLASSPATH

$
```

It should be blank. If it's not, make sure it at least includes a single dot (".") to include the current directory.

If all else fails, don't be afraid to ask around for help. Programming is most often a team effort.

Phew! That was the hardest part.

Once you have Java working, you need to install the Minecraft parts.

Install the Minecraft Client and Server

Minecraft is a client-server system, so you'll need both parts: the desktop graphical client that you use to play the game, and the server process that you connect to, where we'll add plugins.

Install the Minecraft Graphical Client

You probably have this part already, but if not, download the Minecraft installer for your computer from http://minecraft.net. Follow the installer instructions to install it, but don't run it just yet.

When you play Minecraft, this "client" is the application that you run. It connects to your paid account at http://minecraft.net.

There are also clients for Apple's iOS and Android devices, including smartphones, but they are a dead end. As I'm writing this, they don't connect to normal Minecraft servers, so they can't use our custom plugins.

The client handles the graphics and sound, and lets you type chat commands in the game. But right now it has no local game to connect to—guess we better go grab and download that server code.

Install the CanaryMod Server

Now for the fun part. You'll be adding plugins to your own server, so you'll need CanaryMod, a special Minecraft server from the fine folks at CanaryMod.net that's designed to use plugins.

Make a directory named server on your Desktop (that means it would be located in a directory named something like /Users/yourname/Desktop/server):

```
$ cd Desktop
$ mkdir server
```

We'll call this your server directory, and that's where you'll install the server parts.

The CanaryMod project comes in a single jar file: CanaryMod.jar, which is used to run the game server itself, and also what we use to develop new plugins.

Hop on the Web and visit the CanaryMod project's release page at http://canarymod.net/releases. You'll see a listing of their releases, grouped by the corresponding Minecraft version number. I'm using the latest release, which right now is for Minecraft 1.7.10. You'll want to download the latest jar file. I've also included a copy in the downloads for this book for your convenience.

Each jar may be named with extra version numbers (something like CanaryMod-1.7.10-1.1.0.jar; your numbers will probably be larger). Download the file to your Desktop/server directory, and rename it to just CanaryMod.jar.

You can do that from the shell with the mv (move file) command and the wildcard character (*), which matches all the numbers so you don't have to type them out:

```
$ cd server
$ pwd
/Users/andy/Desktop/server
$ ls
CanaryMod-1.7.10-1.1.0.jar
$ mv CanaryMod*.jar CanaryMod.jar
$ ls
CanaryMod.jar
```

The example code for this book contains a subdirectory named runtime. In the runtime directory you'll see a startup script named start_minecraft. Copy it to your server directory using the copy command, cp. You're still in that directory, so you can just copy using ".." to refer to the Desktop, and "." to refer to your current directory, server:

```
$ cp ../code/runtime/start_minecraft .
```

We'll use that start_minecraft script to start up and run the server. It's going to run the Java command, passing in an option to make sure we have enough memory to run (the magical-looking -Xmx1024M), and then passing in the jar file to run, CanaryMod.jar:

```
java -Xmx1024M -jar CanaryMod.jar
```

The first time the Minecraft server runs, it creates a whole bunch of files, including the default World, and then exits. Go ahead and run it (still in the server directory):

```
$ ./start_minecraft
```

Note I used "./" as part of the command name. That will run the command from the current directory. If the current directory (".") is in your path, you won't need to use the "./" sequence.

If you get an error that reads ./start_minecraft: Permission denied, then you'll need to type in the following line to make the file executable:

```
$ chmod +x start_minecraft
```

Once the server launches, Minecraft will spew a bunch of text out to your terminal. Here's what that looks like on my machine; your directory names, timestamps, and version numbers will be different, but should look something like this:

```
$ cd Desktop
$ cd server
$ ./start_minecraft
Please wait while the libraries initialize...
Starting: CanaryMod 1.7.10-1.1.0
Registered xml Database
Found 24 plugins; total: 24
[10:48:32] [CanaryMod] [INFO]: Starting: CanaryMod 1.7.10-1.1.0
. . .
[INFO]: You need to agree to the EULA in order to run the server.
Go to eula.txt for more info.
...
```

There's a lot of spew in the middle there that I left out, but you get the idea.

Somewhere in the middle of the spew, it mentions accepting a license agreement. You need to edit the file eula.txt that it just made in the server directory. If you agree to Mojang's End-User License Agreement (EULA) change the line containing eula=false to eula=true, and save the file. Now you can start the server again with ./start_minecraft. You'll get similar spew this time, but now its waiting for you to type a command at the > prompt.

One more thing before we get started: you want to grant yourself operator (op) privileges. To do that, just type the op command with your Minecraft user name at the server's prompt:[6]

```
> op AndyHunt
[14:32:36] [CanaryMod] [INFO]: [SERVER] Opped AndyHunt
>
```

6. Some earlier versions of Canary would report an error from the op command, even though it actually worked. Just ignore the error.

Start-up errors

In some versions of CanaryMod, you might see an error on startup that says something like `Error on line 1: Premature end of file. net.canarymod.database.exceptions.DatabaseWriteException.` While somewhat scary looking, it just means that the server tripped over itself getting everything set up. Just delete the files in `Desktop/server/db` and try again.

Without operator privileges, you won't be able to break blocks or place anything in the game, so don't skip this step!

If you are planning on inviting friends to play on your server, you can either op each of them as well, or you can give all visitors permissions to build:

```
> groupmod permission add visitors canary.world.build
> [14:33:39] [CanaryMod] [INFO] [MESSAGE]: Permission added
```

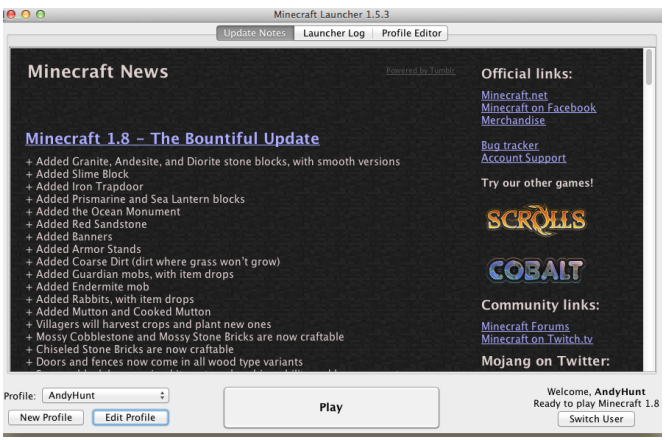
When you're ready, you can stop your Minecraft server at any time by typing the command `stop`, like this:

```
>stop
[10:50:10] [CanaryMod] [INFO] [NOTICE]: Console issued a manual shutdown
[10:50:10] [net.minecraft.server.MinecraftServer] [INFO]: Stopping server
...
[10:50:10] [CanaryMod] [INFO]: Disabling Plugins ...
$
```

And you'll be back to the command-line prompt again.

Start your server back up and leave it running, and we'll try to connect to it from the client.

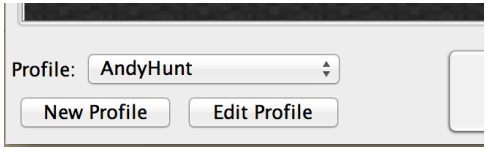
Next, start up your Minecraft client and log in using your Minecraft user name and password. With any luck, you'll see a launcher.



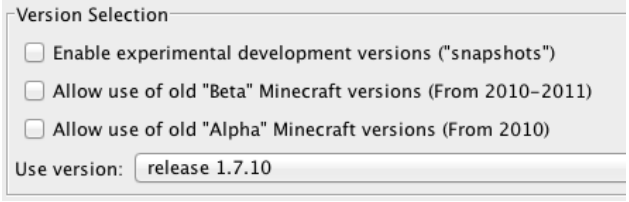
Note the version number in the lower right-hand corner, saying it's ready to play with a version 1.8 server.

When you start up the Minecraft client, you can tell it what version to use. This needs to match the version of Canary you downloaded. As I write this the latest version of the Canary server is 1.7.10, but the client that you just installed defaults to Minecraft 1.8, which CanaryMod doesn't support. That won't work. You'll need to tell the Minecraft client to use the correct version.

In the Minecraft startup program, click on Edit Profile in the lower left.

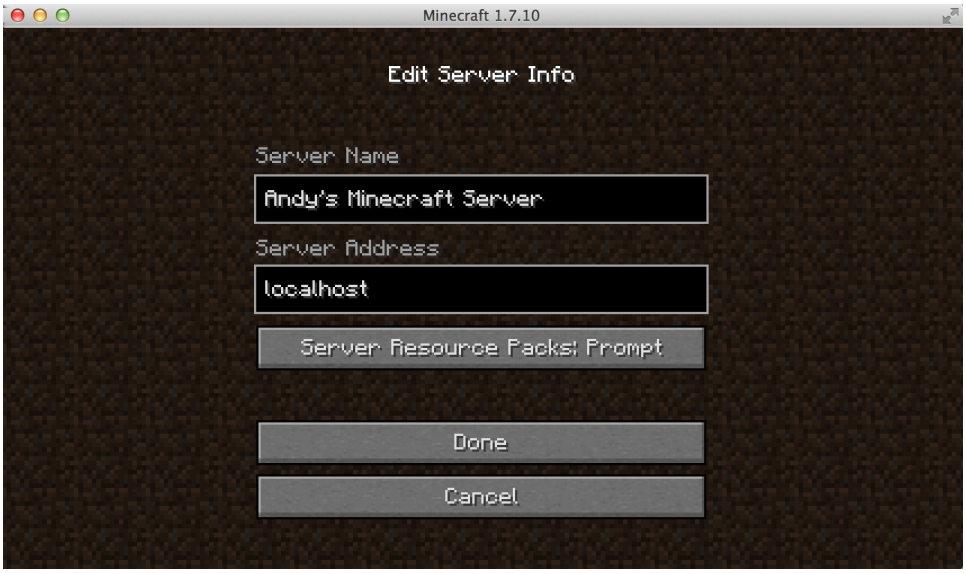


Look for the Use Version setting in the dialog box that pops up. Change the option to use the current version of your Minecraft server, and click Save Profile.



By the time you read this, the version numbers might be completely different. Whatever the versions are, you'll need to check the version in the client and make sure it matches the server's version. Go ahead and click the Play button.

Now, click Multiplayer. Next, you need to add your local server. Click Add Server and type a name for your server. Use localhost for the server address, as shown here:



Your server will show up on the pick list (along with any other servers you regularly connect to), as shown in the following image:

Select your server from the list, then click Join Server. Welcome to the world!

If you get disconnected right away, you may have a version mismatch; double-check which version of the client you're using.

Once connected, you're in the Minecraft world on your own local Minecraft server. Congratulations!

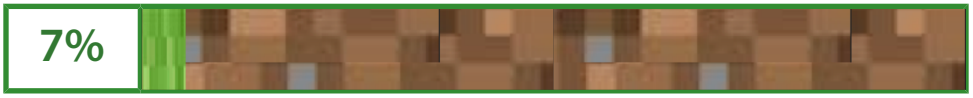
You can take this time to do a little setup in the Minecraft world. Maybe build yourself a nice house before the creepers come....

Next Up

That's a great start, and now you have a full-fledged Minecraft server running on your own computer.

In the next chapter we'll roll up our sleeves and compile and install our first real plugin.

Your Growing Toolbox



You now know how to:

- Use the command-line shell
- Build with Java, `javac`
- Run a Minecraft server

After completing this chapter you'll know how to

- Compile Java source code to .class files, pack them in a jar, and install them on a Minecraft server
- Run your local server with a new plugin
- Connect to your local server

CHAPTER 3

Build and Install a Plugin

Now that you have the tools installed, we'll build a simple, basic plugin. It won't do much as plugins go, but it will make sure you can build and run your own plugins, and it will act as starting point (or skeleton) for all the plugins we'll write in this book.

So how do your typed-in instructions end up running on a Minecraft server? Here's how the whole process works.

You type Java language instructions (we call that "source code") and save them into a text file, and then the Java compiler, javac, reads your text file and converts it into something the computer can run.

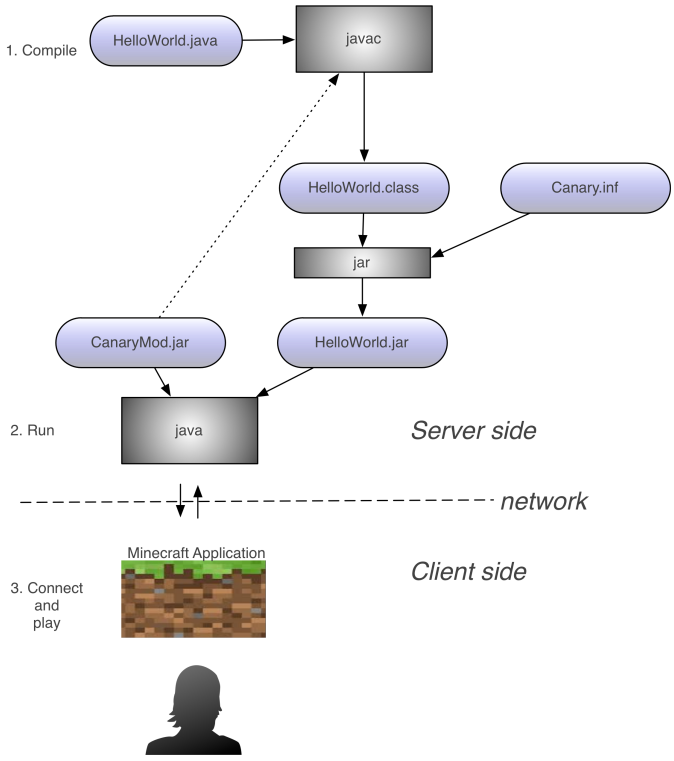
You went through this process already with the simple CreeperTest.java program you typed in previously.

For the source code you type into a file named CreeperTest.java you'll get a *binary* (not text) file named CreeperTest.class. A binary file is just a file of numbers —it makes sense to the computer, but not to humans.

Because a typical program might use lots and lots of class files, you usually archive a bunch of class files into a jar file, and Java runs the code from the jar.

Java (the java program itself) reads class files and jar files to create a running process on the computer. With Minecraft, this will be the server process that your Minecraft clients connect to. For now, the only client will be you.

The following figure shows how these parts all fit together. The javac compiler takes your Java source code, and definitions it finds in CanaryMod.jar, and produces a class file. That class file gets packed up with the Canary.inf file into a jar that is your plugin. Then at runtime, Java starts the server from CanaryMod.jar and loads your plugin from its jar.



In the Java world, you have to place all these files in specific places for this all to work. We made a directory structure like that earlier, all ready for your version of the HelloWorld plugin. I've also got a complete plugin all set up for you in the HelloWorld directory in the code for this book, which you downloaded to Desktop/code/HelloWorld.

So in Desktop/code/HelloWorld, you'll find a directory tree for the source code, under src. You'll also see a bin directory where the compiled class files are created, and a dist directory where the class file and configuration files are packed together into a jar file. When you're ready to share your plugin with others, you'll give them the jar.

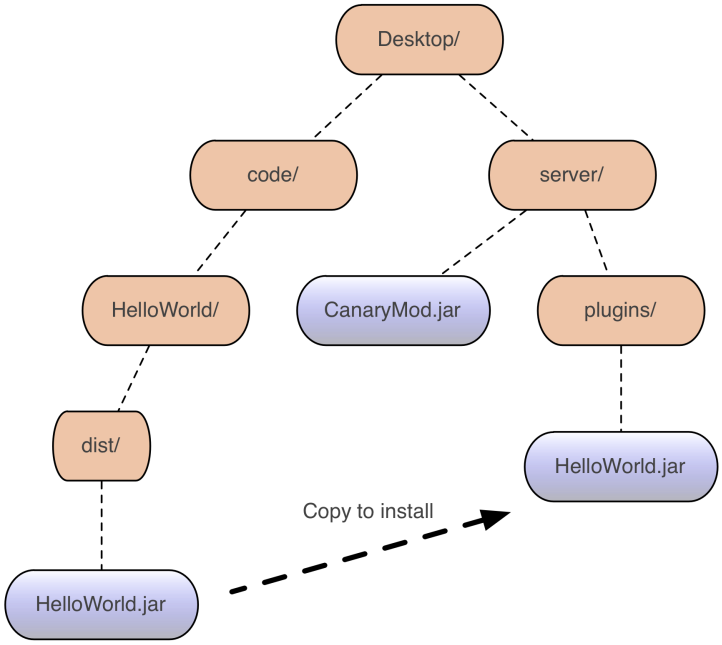
HelloWorld is one development directory. You'll probably have one of these for each plugin you develop, each with its own src, bin, dist, and so on.

Then over in your server directory at Desktop/server, you have the Minecraft server files, including CanaryMod.jar, which contains all the bits you need to run the game, as well as the parts we're using to develop code in the Minecraft worlds.

Also in server, there's a directory for plugins that the Minecraft server will use when it runs, and the lib directory (we'll use that at the end of this chapter for our EZPlugin library).

When working on code in the development directory, the last step once you're ready to test it out in a server is to copy the jar file up to the server's plugin directory (see the following figure).

We'll see how to do that automatically in just a second.



Java tends to use paths and configuration files to specify where all these files and directories live. It can get a little tricky at times, as there are a lot of moving parts, and it's frustrating when Java can't find some critical file that is sitting right there in front of you. Just because *you* know where a file is doesn't mean Java knows.

Now let's look at the source code, then cover how to build and install it.

Plugin: HelloWorld

It's a long-held tradition in the programming world to start off with a simple test program that prints out the message "Hello, World." So we'll start off by building and running an existing plugin that does that in Minecraft—except we'll send out a slightly more interesting message.

Here is the Java source code for our HelloWorld plugin, which is already typed in for you in the file ~/Desktop/code/HelloWorld/src/helloworld/HelloWorld.java. There's a lot of weird stuff in here. (If you haven't downloaded the code for this book to your Desktop yet, grab it now from http://media.pragprog.com/titles/ahmine2/code/ahmine2-code.zip. You can use unzip from the command line to unpack the archive and create all the files.)

HelloWorld/src/helloworld/HelloWorld.java

```
1 package helloworld;
2 import net.canarymod.plugin.Plugin;
  import net.canarymod.logger.Logman;
  import net.canarymod.Canary;
  import net.canarymod.commandsys.*;
  import net.canarymod.chat.MessageReceiver;

3 public class HelloWorld extends Plugin implements CommandListener {

    public static Logman logger;

    public HelloWorld() {
      logger = getLogman();
    }

    @Override
    public boolean enable() {
      logger.info("Starting up");
      try {
        Canary.commands().registerCommands(this, this, false);
      } catch (CommandDependencyException e) {
        logger.error("Duplicate command name");
      }
      return true;
    }

    @Override
    public void disable() {
    }

4   @Command(aliases = { "hello" },
             description = "Displays the hello world message.",
             permissions = { "" },
             toolTip = "/hello")
5   public void helloCommand(MessageReceiver caller, String[] parameters) {
      String msg = "That'sss a very niccce EVERYTHING you have there...";
      Canary.instance().getServer().broadcastMessage(msg);
    }
  }
```

Don't be discouraged if this looks like space-alien speech or Elvish right now. We'll make sense out of it over the next several chapters. Instead focus on what *is* familiar: there are some English words in there, like "import" and "public" and "return," and what might be sentences or statements of some kind, which all end with semicolons (";"). There are also some strange characters like "{" and "}" that seem to be important.

What does it all mean? Well, this plugin implements a user command, /hello, which will broadcast the traditional creeper greeting, "That'sss a very niccce EVERYTHING you have there..." to all online players in Minecraft.

Notice that the name of this plugin is declared as public class HelloWorld on the line at ❸. That's the same name as the file name that contains this code: HelloWorld.java. This piece of code is also set up to be in a *package*—that is, a group of related files—using the same name on the line at ❶. The package name is all lowercase, and it's also the directory name where our Java source code file lives, under src/ in helloworld/HelloWorld.java.

It's important that the names in all of these places match; a typo on one of them can lead to strange errors.

You use import statements (you'll see these beginning on the line at ❷) to get access to other things that you need in your plugin, like parts of the Canary library and other Java libraries. If you forget to include an import for something you need, you'll get an error that says "cannot find symbol" because Java doesn't know what you mean. For your convenience, I've included a list of all the imports we're using in Appendix 7, *Common Imports*, on page 253.

The code for our plugin starts at ❸, and there's this funny @Command annotation (a kind of a tag, not actual code) at ❹ that describes the command itself. Finally, the code for that command starts at ❺.

We'll look at all that and more later, but first, we need to let the server know that we've got a plugin for it to load.

Configure with Canary.inf

This source code alone isn't enough; you also need a configuration file so that Minecraft can find and launch your plugin. The configuration file is named Canary.inf and looks like this:

HelloWorld/Canary.inf

```
main-class = helloworld.HelloWorld
name = HelloWorld
author = AndyHunt, Learn to Program with Minecraft Plugins
version = 1.0
```

Here's a description of what this file needs. Don't worry much about the details yet—it will make more sense as we get further into the book.

main-class
: Name of the package and class that Java will run to start this plugin (*package.classname*).

name
: Name of the plugin—in this case, `HelloWorld`.

author
: Name of the author (that's you).

version
: Version number of your plugin. Start low, and increment the number each time you release a new version to the world.

Build and Install with build.sh

The commands you've been typing in your terminal window can also be saved into a file; that way you can run them over and over again without having to retype them each time. We call this a *shell script*, and it's another way to program the computer.

Building a plugin is only a little more complicated than compiling a single Java file as we did last chapter, but even so, it involves a lot of commands we don't want to have to type out every time.

To make it easier, I've made a shell script for you named `build.sh` that will do the three main steps:[1]

1. Use `javac` to compile the `.java` source to `.class` files.
2. Use `jar` to archive the class files, manifest, and configuration file.
3. Copy the jar file to the server.

After that, you'll need to stop and restart the server so it can pick up the changes.

The build script needs to know where your server directory is located. At the very top of the script, it says this:

```
MCSERVER=$HOME/Desktop/server
```

1. For larger projects, folks use tools like Ant, Maven, or Gradle when the build becomes more complex and has to manage dependencies among many parts. But that's overkill for our needs here.

For most people that should just work if your server directory is located on your Desktop. If that doesn't work, you'll need to edit the build.sh file and change that directory so that the MCSERVER= points to your local Minecraft server directory. That MCSERVER= setting is how the script knows where to find the server.

Change your current directory to the HelloWorld directory. From there, run the build.sh script (this is what we'll do for each plugin from now on):

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

(Remember, no matter how you actually get to your Desktop, I'll just show it as cd Desktop from now on for reference. You may need to do a cd ~/Desktop or start at your home directory and go down, depending on your particular system.)

You should see results that look a lot like this:

```
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

Check to see that the file really was installed in the server directory:

```
$ cd Desktop
$ cd server/plugins
$ ls
HelloWorld.jar
```

Yup, the jar file is in the server directory. Success!

If you get errors, here are a few things to check:

- If you get the error ./build.sh: Permission denied you might need to type chmod +x build.sh to give the script executable permission.
- If you're seeing syntax errors, make sure you are using a fresh copy of the files downloaded from this book's website, with no local modifications.
- If everything compiles okay but you get an error trying to copy the jar file, make sure the server directory is correct.

If the script is having trouble finding the server directory, edit build.sh and change the MCSERVER= directory name to the correct location of your Minecraft server.[2]

2. Make sure it's spelled correctly and starts with a "/" character so that you have the full path to your server starting at the root directory.

(If you have to change it here, you may need to do the same thing for each new plugin's build script as we go along).

Once it compiles and installs, you are excellent! Now you have a compiled plugin, ready for the Minecraft server to use.

If your server is still running, it won't know about this new plugin. You have to stop it and then restart it.

```
> stop
[14:36:24] [net.minecraft.server.MinecraftServer] [INFO]: Stopping server
[14:36:24] [net.minecraft.server.MinecraftServer] [INFO]: Saving players
[14:36:24] [net.minecraft.server.MinecraftServer] [INFO]: Saving worlds
[14:36:24] [net.minecraft.server.MinecraftServer] [INFO]: Saving chunks
           for level 'default'/Overworld
[14:36:24] [CanaryMod] [INFO]: Disabling Plugins ...
$ ./start_minecraft
08:47:32 [INFO] [HelloWorld] Loading HelloWorld v0.1
08:47:32 [INFO] [HelloWorld] Loaded.
08:47:32 [INFO] [HelloWorld] Enabling HelloWorld v0.1
08:47:32 [INFO] [HelloWorld] Starting up.
08:47:32 [INFO] Server permissions file permissions.yml is empty, ignoring it
08:47:32 [INFO] CONSOLE: Reload complete.
>
```

And there's the startup message from our new HelloWorld plugin. If you don't see any message from HelloWorld starting up, then your Minecraft server can't find it. Make sure the HelloWorld.jar file is in the server's plugins directory, stop the server, and try starting it up again.

Once you're connected and in the Minecraft world, you can test out your fine new command from the client chat window. In the Minecraft game, just start typing /hello and see what happens. As soon as you type the "/" character, you'll see that you're typing in a chat window at the bottom of the screen. Press Return and...

...you should see our message appear in the server log console and in the game window. This is what it looks like in the server console:

```
14:47:58 [INFO]: Command used by AndyHunt: /hello
14:47:58 [INFO]: That'sss a very niccce EVERYTHING you have there...
```

And here's what it looks like in the game window:

Getting Around in Minecraft

In case you aren't familiar with the Minecraft graphical user interface (GUI), here are a couple of quick tips. You can check out some of the many YouTube tutorials or official docs for more.

The keys `W`, `A`, `S`, and `D` move you forward, left, backward, and right, respectively. Use the `Spacebar` to jump.

Use the mouse to control the direction you are facing.

You "hit" things with your left mouse button—for example, to strike with a sword or dig with a pickaxe or shovel.

You "use" items with your right mouse button—for example, placing an item or opening a door.

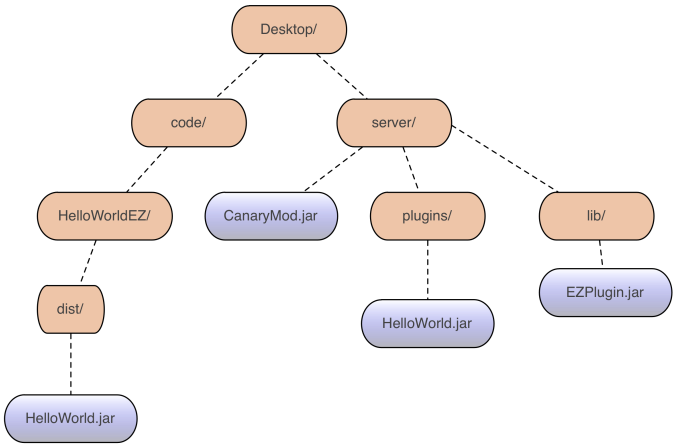
You type commands to Minecraft with a leading "/" character. You can change the game to creative mode with `/gamemode c`, and back to survival mode with `/gamemode s`.

Using EZPlugin

As we go through the book, we'll talk more about the Java language, and explain what all the bits and pieces in `HelloWorld.java` actually do. But about half of this code isn't really important to us right now. What's more, it's always going to be exactly the same for each plugin we use. We're going to move it out of the way so you won't have to keep looking at it in every single plugin that we work with.

In the downloaded code, I've included a special library called EZPlugin. I moved all the stuff that's going to be the same for all plugins into EZPlugin.java. That's going to make our next plugins much smaller and easier to read.

All of the other plugins (past HelloWorld) depend on EZPlugin, so we'll need to build it before we go on. The build process will install it in the server's lib (short for "library") directory:



At your command line, change your current directory to the EZPlugin directory. From there, run the build_lib.sh script. So for me, I can start anywhere and go to my home directory, then I can cd down into Desktop, then code, then EZPlugin:

```
$ cd
$ cd Desktop
$ cd code/EZPlugin
$ ./build_lib.sh
```

You should see results that look a lot like this:

```
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

Check to see that the file really was installed in the server's lib directory:

```
$ cd Desktop
$ cd server/lib
$ ls
EZPlugin.jar
```

Once EZPlugin.jar is there, you won't need to run build_lib.sh again. All the other plugins can use it now.

Next, get rid of the first version of HelloWorld.jar, in the server's plugins directory:

```
$ cd ../plugins
$ ls
HelloWorld.jar
$ rm HelloWorld.jar
$ ls
$
```

Now have a look at a much simpler version of HelloWorld, which uses EZPlugin, and is located in code/HelloWorldEZ/src/helloworld/HelloWorld.java:

HelloWorldEZ/src/helloworld/HelloWorld.java

```
package helloworld;
import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.chat.MessageReceiver;
import com.pragprog.ahmine.ez.EZPlugin;

public class HelloWorld extends EZPlugin {
  @Command(aliases = { "hello" },
           description = "Displays the hello world message.",
           permissions = { "" },
           toolTip = "/hello")
  public void helloCommand(MessageReceiver caller, String[] parameters) {
    String msg = "That'sss a very niccce EVERYTHING you have there...";
    Canary.instance().getServer().broadcastMessage(msg);
  }
}
```

Notice that this just has a bunch of imports at the top, and then the command business—the part we're actually interested in—down at the bottom. This will be the skeleton for all our upcoming plugins.

Go ahead and make sure you can build it, and that it can find the EZPlugin library:

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

You should see the usual successful output:

```
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

If not, double-check that EZPlugin.jar is in Desktop/server/lib, and go back into EZPlugin.jar and rebuild it if needed.

Next Up

Congratulations! You just compiled and installed a plugin from source code, installed it on your local server, connected, and tested it out! You then built the `EZPlugin` library and installed it as well.

With that out of the way, we'll spend the next few chapters taking a deeper look at all the source code that makes a plugin, and see what makes Java tick so you can make your own plugins.

Your Growing Toolbox

You now know how to:

- Use the command-line shell
- Build with Java, `javac`
- Run a Minecraft server
- Deploy a plugin
- Connect to a local server

In this chapter we'll take care of a lot of the basics of writing Java code. With these tools in your toolbox, you'll know the following:

- What the funny characters mean in Java
- How to use *variables* to hold number and character string values
- How to declare and use *functions*: lists of Java instructions
- How to control code with if, for, and while statements

CHAPTER 4

Plugins Have Variables, Functions, and Keywords

As you may have noticed, there's a lot of text in the plugin source code that I haven't explained yet. Let's dig deeper into Java and take a look at what all that text means—it's the raw material of programming plugins.

As you saw in Chapter 3, *Build and Install a Plugin*, on page 31, a program in Java is just a text file. Look at the text in HelloWorld.java: there's a lot of sort-of English words, and some strange-looking punctuation. All of it means something, and the Java compiler is mighty picky when it's reading your text. First of all, spelling counts. Entity is not the same as Entitee. Player is not the same as player, so uppercase and lowercase matter, too. There are times when you use uppercase, and times you need to use lowercase.

Each of those odd characters you see has a special meaning, and they're used for different things. Here are some examples (don't worry much about the details yet):

`//` Comment, used to leave a note for yourself or others. Java will ignore anything you type from after the double slash to the end of the line.

`/* longer comment */` Used for longer comments, either on one line or spanning multiple lines.

`( )` Parentheses, used when calling a function and passing data to it, like this: `System.out.println("Hello, Creeper");`.

`[ ]` Brackets, used for choosing one item from a list of items, like this: `second = myArray[1];`.

{} Braces, used to mark the beginning and end of a section of code.

. Dot, used to select a part of something. Most often you use this to select a function that's part of an object, like `System.out.println()` or `player.getLocation()`.

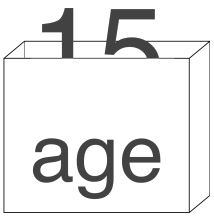
; Semicolon, used to mark the end of a code statement. Leaving the semicolon off at the end of a line of code is a surefire way to make Java angry and have it spew hundreds of error messages at you.

Some words you type are special, and you have to use them the way Java says to. Other words you can make up yourself, and could be anything from `ImmortalPlayer` to `rathead`. So how does this all fit together?

To create a program or a plugin in Java, you use all these bits of text to create two different kinds of things: data and instructions. That's all there is to any computer program or to a Minecraft plugin. It's all just data and instructions. Here's how we work with them.

First I'll show you how you declare and use these things, and then you'll need to try it yourself in your very first plugin.

Keep Track of Data with Variables



Data are facts—things like your age or the color of your bicycle. To work with these facts in Java, you hold them in *variables*. A variable is a holder of data. Think of it like a small box you can put things in.

In Minecraft, we use variables to keep track of things like players in the game, a player's current health and location, and all the other data we need to know. (In fact, you're going to add some variables to a piece of code in just a minute—a plugin that will build you a house.)

A variable is the box that holds that data. You can put a label on the box, but that's just a label for your convenience—it doesn't affect what's inside.

Here's how to use variables. For this example, we'll make a variable called `age`, and tell Java to set the value of that variable to 15, either in two steps or all in one step:

```
int age;
age = 15;
```

or

```
int age = 15;
```

The important part here is that you're telling Java the *type* of the variable you want to make—in this case, a whole number (no fractional part or decimal point). A whole number is known as an integer, which is abbreviated in Java as `int`.

There are lots of types in addition to `int`, including `float` and `double`, which hold fractional numbers (with decimal points); `String`, which holds a sequence of characters such as `"Hello, World!"`; types you can create yourself, such as `NunChuck` or `CowBell`; and types that Minecraft defines, such as `Player` and `Location`. Any time you declare a variable in Java, you need to specify the type for that variable. Java can't guess; you need to tell it.

You can create the variable and assign its value in one step, or create the variable first and then assign it later. In either case you can put a new value in that variable any time you want.

This is okay:

```
int age = 15;
age = 39;
age = 21;
```

However, you can't declare it a second time. This won't work:

```
int age = 15;
int age = 21;
// Error!
```

Instead, only declare it as `int age` once, then use `age =` to change the value if needed.

Now, just because you "labeled the box" *age* doesn't mean it really has to hold an age. There's nothing to stop you from putting some other number in that box:

```
int age = 2048;
```

That's perfectly legal Java code, as 2048 is a perfectly reasonable number. It might be okay for the age of a historical relic, but it's not a realistic age for a person. Setting that age for a person would be stupid, but there are no laws against stupidity. That means it's up to you to give variables names that make sense. Renaming this variable to something like `ageInYears` might make more sense, to avoid problems like the famous crash of the Mars climate orbiter, where one set of programmers used metric units and the other used Imperial units. Oops.[1]

1. http://mars.jpl.nasa.gov/msp98/news/mco990930.html

Java does try to keep you from mixing apples and oranges. Although it doesn't know that you're using an int as a "human age," it does know the difference between things like an int, a float, a String, and so on. And it won't let you mix up those types. If you've declared that a variable should hold an integer, you can't try to store a String in it. This will throw an error:

```
int age = "Old enough to know better";  // Error!
```

We tried to store a string of characters in an int. That won't work, and Java will complain. You can't go the other way either. Here we're trying to store an integer value (42) into a variable declared to hold a String:

```
String answer = 42;   // Error!
```

But for common types such as numbers and strings, there are ways you *can* convert things back and forth as needed, if it makes sense. For instance, suppose you read a numeric value from something the user typed, and it was given to you as a String named str. You can make it an int like this:

```
String str = "1066";
int value = Integer.parseInt(str);
// value is now set to 1066 as a number
```

So you can convert, but you have to do it yourself; Java won't guess for you. There's a list of common type conversions, including ones we haven't covered here, in Appendix 5, *Cheat Sheets*, on page 241.

Let's try that out.

Plugin: BuildAHouse

I've got a plugin already set up for you; all you need to do is declare some variables and you can give the /buildahouse command.

First make your way to the downloaded code, and into the BuildAHouse plugin:

```
$ cd Desktop
$ cd code/BuildAHouse/src/buildahouse
$ ls
BuildAHouse.java MyHouse.java
```

You're going to edit the file MyHouse.java, which is one small part of this whole plugin (don't look at the rest yet!). Right now it looks like this:

BuildAHouse/src/buildahouse/MyHouse.java

```
package buildahouse;
public class MyHouse {
  public static void build_me() {
    // Declare width
    // Set width to the number of blocks
    // Declare height
    // Set height to the number of blocks

    BuildAHouse.buildMyHouse(width, height);
  }
}
```

If you try to compile and install that with ./build.sh like we did with HelloWorld, you'll get two errors:

```
$ cd Desktop
$ cd code/BuildAHouse
$ ./build.sh
Compiling with javac...
src/buildahouse/MyHouse.java:10: cannot find symbol
symbol  : variable width
location: class buildahouse.MyHouse
    BuildAHouse.buildMyHouse(width, height);
                             ^
src/buildahouse/MyHouse.java:10: cannot find symbol
symbol  : variable height
location: class buildahouse.MyHouse
    BuildAHouse.buildMyHouse(width, height);
                                    ^
2 errors
```

And that's your first mission: declare and set an int variable named width and an int variable named height, and set them to something reasonable for a house, perhaps no smaller than 5 blocks high and 5 blocks wide. Or maybe 10×10 if you're feeling spacious.

Delete those comment lines and replace them with your two variables for width and height. Save the file, and then go ahead and run build.sh again:

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

Stop and restart your server, then connect (or reconnect) your Minecraft client. Pick a nice-looking spot in the Minecraft landscape, and type the command /buildahouse.

Bam! You are now inside your brand-new, creeper-proof house, which was built to the exact dimensions you specified with your `width` and `height` variables. A right-click will open the door, by the way.

Different Kinds of Numbers

Java makes a distinction between *integer* whole numbers (with no decimal point, like 8) and *floating-point* numbers with a fractional part (with a decimal point), like 10.125.

For plain old whole numbers, you use an `int`, like we've seen.

Floating-point numbers can be `float` or `double`. A `double` is larger and can store numbers much more precisely, but at the cost of needing more space and power to manage. But computers are fast and have plenty of storage these days, so almost everyone just uses `double` any time they need a double-precision, floating-point, fractional number.

When you type a number with a decimal point in Java, it assumes you've typed a `double`:

```
3.1415 // assumed to be a double
```

But if you really need the number to be a `float`, you have to stick the letter `f` on the number:

```
3.1415f // now it's a float, not a double.
```

We'll need to use floats in just a minute to play a sound effect, because that's what Canary requires to set the volume and pitch of the sound. And in general you might need a floating-point number more often than you think.

For example, let's look at a simple division problem. In Java, you write division using the "/" character (instead of ÷), so to divide the number 5 in half you'd write `5 / 2`. Depending on how you do the division, though, the answer might surprise you.

- 5 / 2 is 2 (just 2, nothing more)
- But 5 / 2.0 is 2.5, as you would expect
- 5.0 / 2.0 is also 2.5

Why is 5 divided by 2 equal to only 2? On a math quiz, that would be wrong. But we're not dividing real numbers here; we're dividing int numbers, so the result is another whole number; another int. There are no fractions at all.

If you want the answer to include fractions, then at least one of the numbers involved has to have a fractional part. That's why 5 divided by 2.0 (note the extra .0) gives us the real answer of 2.5 (two and a half). This time, we're using an int (5) and a double (2.0).

Sometimes you won't care about fractional parts or remainders (leftovers). If you're calculating something that doesn't *have* fractional parts, then all-int math is just fine. Half of a Player or a Cow doesn't make sense (unless you're making hamburger). But if you need fractional answers, then at least one of the numbers involved has to be a double or a float.

Here's a handy list of several of the common math operators you might need:

Addition:	+	Subtraction:	-
Multiplication:	*	Division:	/

They work just as you'd expect, but there are some handy shortcuts.

```
int health = 50;
health = health + 10;
```

is the same as

```
int health = 50;
health += 10; // Same thing
```

Both result in health being set to 60.

You can use expressions like += and -= to change the value of a variable without having to repeat its name. Hey, less typing. I like it.[2]

If you're just adding 1 or subtracting 1, then there's an even easier way of typing it:

```
int health = 50;

health--; // Subtracts 1 from health
health++; // Adds 1 to health
```

In case you want to get really fancy and use more advanced math, including trig functions like sine and cosine, constants like pi, and that sort of thing, Java has libraries with all of that ready for you to use.

Strings of Characters

There's more to the world than numbers, though. While your government or school may know you as a string of numbers like #132-54-7843, your friends call you by a name that's a string of characters, like "Jack" or "Jill" or "Notch."

In Java, you use a String type to handle strings of characters. We'll use strings in Minecraft a lot, for names of players, names of files, messages—any kind of text data that can change value, like a number does.

To specify a string in code literally, you put it inside double quotes, which looks "like this". We'll use strings in our plugins and look more at what you can do with strings as we go along.

You can add strings together using a plus sign:

```
String first = "Jack";
String middle = "D.";
String last = "Ripper";

String name = first + " " + middle + " " + last;
// Now name will be "Jack D. Ripper"
```

Note that strings are your *data*, which is different from the characters we use to give Java instructions (our program *code*).

Try This Yourself

It's time to try out some of these ideas: we'll make a simple plugin from scratch.

2. The best programming languages make you type the fewest characters to get something done while still making sense.

In the BuildAHouse plugin, I had a bunch of code that you didn't see that actually did the house-building, and all you had to do was declare a few variables. But now you're going to make *an entire plugin from scratch*, all by yourself.

To keep things a little on the simple side, at first you're just going to print out some values. In fact, let's call this plugin Simple.

First you need to make a Simple directory for the new plugin, with a src/ subdirectory, a src/simple subdirectory, and with a Canary.inf, Manifest.txt, and build.sh, just like we had in HelloWorldEZ.

I already provided a Simple directory for your reference down in Desktop/code, but don't look at that unless you get stuck. You're going to make your *own* Simple right under Desktop.

Since you might be doing this a lot, I've made a shell script named mkplugin.sh that will get you started.

In your Desktop directory, run the mkplugin.sh with the name of the plugin you want to create, and it will make the directories underneath your current directory and start you off with bare-bones Java code:

```
$ cd Desktop
$ code/mkplugin.sh Simple
$ cd Simple
$ ls
Canary.inf  Manifest.txt bin  build.sh  dist  src
$ cd src
$ ls
simple/
$ cd simple
$ ls
Simple.java
```

The file Simple.java lives up to its name; it doesn't actually *do* anything.

You'll be adding some code in between where it says Put your code after this line and ...and finish your code before this line, as shown at the bottom of the following listing.

Open this file (Desktop/Simple/src/simple/Simple.java) in your text editor and get ready to type.

Plugin: Simple

Simple/src/simple/Simple.java

```
package simple;
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 com.pragprog.ahmine.ez.EZPlugin;

public class Simple extends EZPlugin {

  @Command(aliases = { "simple" },
           description = "Displays simple variable assignments",
           permissions = { "" },
           toolTip = "/simple")
  public void simpleCommand(MessageReceiver caller, String[] parameters) {
    if (caller instanceof Player) {
      Player me = (Player)caller;
      // Put your code after this line:

      // ...and finish your code before this line.
    }
  }
}
```

Don't worry about all that extra program text yet. We'll talk about that more as we go along. For now just put new code on the lines as shown, and it will work fine. Here's what you're going to do:

First, add two import statements at the top of the file, after the other `import` lines at ❶. Type in the following:

```
import net.canarymod.api.world.effects.SoundEffect;
import net.canarymod.api.world.position.Location;
```

Next, after where it says `// Put your code after this line:` at ❷, do this:

1. Create an integer variable named `myAge` and set it to whatever your age is.
2. Make another integer variable named `twiceMyAge` and set it equal to `myAge` multiplied by 2.
3. Create a float variable named `volume` and set it equal to `0.1`.
4. Create a float variable named `pitch` and set it equal to `1.0`.

5. Create a double (floating-point) variable named “dayOnIo” and set it to 152853.5047. That’s how many seconds a day lasts on Jupiter’s moon, Io.[3]

6. Create a string named myName and set it to your name.

7. Display each of these values by sending a chat message to the player, using me.chat(*string msg*). For example, to display your name, type this:

```
me.chat("My name is " + myName);
```

State each message with a string (like "My name is ") and the plus sign. Then add your variable. Don’t forget the semicolon at the end of each statement.

Finally, add lines to play a sound effect at your location, using the float values you just declared:

```
Location loc = me.getLocation();
playSound(loc, SoundEffect.Type.GHAST_SCREAM, volume, pitch);
```

Save the file and then run the build.sh in the Simple directory. (Make sure to stop your server if you’ve left it running; some operating systems will throw errors if you try to install fresh jars while the server is still running.)

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

Stop and restart your server, then connect from the Minecraft client. Run your new command, /simple, and marvel at the messages on your console (and the sound effect!).

Once that works, try changing the values for volume and pitch. Crank volume up to 1.0f for a terrifying scream. For pitch, try a very low value, like 0.1f for more of a growl, and higher (10.0f?) for a piercing shriek.

After you type the new value in the Java source code, don’t forget to take these steps:

1. Save the file.
2. Compile and install with ./build.sh.
3. Stop and restart the server.
4. Reconnect your Minecraft client.
5. Type /simple in the client and enjoy the display and sound effect.

3. Io is pretty cool. It has more than 400 active volcanoes. See http://en.wikipedia.org/wiki/Io_(moon).

Here's what mine looks like:

(No, I'm not *really* 99, but I'm not going to tell you my real age, or my bank account number, or....)

If you need to see my code for hints, take a look at `code/MySimple/src/mysimple/MySimple.java`.

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 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:

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

Then you could use that 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.

We snuck something else into this example. Our makeFrosting function isn't real Java code. When you write out an idea that's codelike but isn't really a programming language, we call it *pseudo-code*. Programmers use pseudo-code the same way artists sketch a picture before starting to paint—it helps them to see the big picture.

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 function that Minecraft calls when the game is running: helloCommand.

We call these kinds of functions in a plugin the *entry points*. These are the functions that the Minecraft server will call when it needs to. You provide the code, and the game will call it when needed.

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

```
public void helloCommand(MessageReceiver caller, String[] parameters) {
  String msg = "That'sss a very niccce EVERYTHING you have there...";
  Canary.instance().getServer().broadcastMessage(msg);
}
```

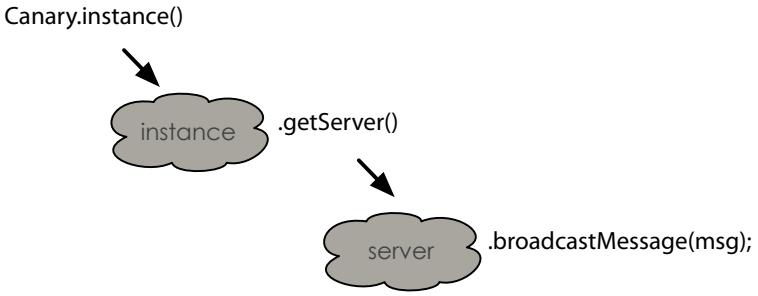
There's a call to something named instance(), a call to getServer(), and a call to broadcastMessage().

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 (it's actually a Server). Once we have that, we can then call any of the Server's functions. In this case, we're calling the Server's broadcastMessage() function, passing in a string argument named msg. Take a look at this line in action:

```
Canary.instance().getServer().broadcastMessage(msg);
```



To make it clearer, you could break up this chain of function calls into separate pieces, which would look something like this:

```
instance = Canary.instance();
server = instance.getServer();
server.broadcastMessage(msg);
```

See? It's just a set of function calls. 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 helloCommand in HelloWorld).

In between 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 long howlong(int years) {
  // Write this function...
}
```

The function will take a number of years and return a number of seconds as a long (an extra-big int). We'll cheat a bit to make this easy, and convert years to seconds. (See the footnote if you need a hint.)[4]

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:

```
17  import net.canarymod.api.world.position.Location;
18  import com.pragprog.ahmine.ez.EZPlugin;
19
20  public class MySimple extends EZPlugin {
21                                          Add function definition here
22    @Command(aliases = { "mysimple" },
23             description = "Displays Andy's simple var
24             permissions = { "" },
25             toolTip = "/mysimple")
26    public void mysimpleCommand(MessageReceiver caller,
27      if (caller instanceof Player) {
28        Player me = (Player)caller;
29        // Put your code after this line:
30
31        int myAge = 99;
32        int myAgeDoubled = myAge * 2;
33        float volume = 0.1f;
34        float pitch = 1.0f;
35        double dayOnIo = 152853.5047;
36        String myName = "Andy Hunt";          And call it here
37
38        me.chat("My age " + myAge);
39        me.chat("My age doubled " + myAgeDoubled);
```

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 chat() calls do.

4. 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.

If I compile and install it with ./build.sh, stop the server and restart it, 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.

Use a for Loop to Repeat Code

Computers are much better at repetitive tasks than humans are; you can tell the computer to do something ten times in a row and it will do *exactly* that, ten times, a hundred times, a million times, whatever you want. One way to do the same thing a bunch of times is to use a for loop. for is a Java keyword that lets you loop over a section of code a fixed number of times.

The for loop is a basic control structure in Java—a way to control the order of execution of your lines of code. If you need to make a bunch of blocks or spawn a lot of creepers in a Minecraft world, you'll use a for loop. If you need to loop through all the players that are currently online, you'll probably use a for loop (although there are nicer ways of doing that, which we'll see a little bit later).

For example, this snippet of code from the guts of a plugin will spawn ten pigs at your location. Saddle up!

```
//... somewhere inside a plugin:
for (int i=0; i < 10; i++) {
    spawnEntityLiving(location, EntityType.PIG);
}
```

In this case, the for statement will run the instructions in its braces ten times, so spawn will be called ten times, creating ten pigs.

The for statement has three parts inside the (), separated by semicolons. Here's what they do:

int i=0;

The first part declares and *initializes* the looping variable. Here we'll use i as our loop counter, and it always starts off at 0—you'll see why later, but Java always counts starting at 0.

i < 10;

The loop *test*. This tells us when to keep going with the loop (and more importantly, when to stop). This loop will keep running the code in the following braces as long as i is less than 10. Right now, that would mean forever, so we need the third part:

i++;

The loop *increment*. This is the part that keeps the loop moving along. Here we are incrementing the variable i by 1 each time through the loop. Remember, i++ is shorthand for i=i+1. Either way, you are taking the value of i, adding 1 to it, and saving that back as the new value of i (you can use that kind of shortcut anywhere, by the way, not just in loops).

Use an if Statement to Make Decisions

An if statement lets you make decisions in code and optionally run a piece of code depending on whether a condition is true. This is how you make a computer "think."

It's just like the real world. We run on "if" statements all the time. If your age is >= 16, you are allowed to drive. If the door is unlocked, you can open it, or else you cannot. If the command given to Minecraft is equal to "hello", then send a message. If your health drops to zero, you're dead.

This is what it looks like in Java:

```
if (something) {
    // run this code...
}
```

The part in parentheses (something) can be anything that turns out true or false; in other words, a Boolean condition (more on that in just a second).

If you need to, you can also put code in to run if something is true *and* specify what to run if it's false:

```
if (something) {
    // run this code if something is true
} else {
    // run this code if something is false
}
```

For example, here's a fragment of code that will say something different depending on whether the String in the variable myName contains Notch.

```
if (myname.equalsIgnoreCase("Notch")) {
    say("Greetings from Notch!");
} else {
    say("Notch isn't here anymore.");
}
```

if statements (with or without the else) are critical to programming: that's how you can get the computer to make decisions and pretend to "think." Pretty powerful stuff, but really simple to use.

Compare Stuff with Boolean Conditions

if statements decide what to run based on whether something is true. But just what is true?

Besides numbers and strings that we've seen, you can also make a variable that keeps track of whether something is turned on, like a toggle switch. In Minecraft, we'll use this to determine all sorts of things: if a player is on the ground or not, if a string matches another string, whether to use a game event or ignore it, and much more.

Java calls this kind of variable a boolean,[5] and it can be assigned true or false, or you can give it math expressions using any of these operators, all of which return either true or false:

==	Equal to *(two equals signs)*
!=	Not equal to
!	Not *(so "not true" is false, and "not false" is true)*
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
&&	And *(true if both things are true)*
||	Or *(true if either thing is true)*

For example, given these variables

```
int a = 10;
int b = 5;
String h = new String("Hello");
boolean result = true;
boolean badone = false;
```

Java will figure out these comparisons:

- a == 10 is true
- b == 6 is false
- a < 20 is true
- b >= 5 is true
- a > 100 is false
- result is true
- !result is false *(pronounced "not result"—"not" returns the opposite of a value)*
- result && badone is false *(pronounced "and"—true only if both are true)*
- result || badone is true *(pronounced "or"—true if either is true)*

But this next one won't do what you think it should; it will not be true:

```
h == "Hello"; // Gotcha!
```

That one is tricky. For strings and objects (more on that in the next chapter), use the equals function instead of the double equals sign (==), like this:

```
h.equals("Hello");          // is true
```

5. Named for George Boole, the British mathematician, who came up with these ideas in the 1800's.

```
h.equalsIgnoreCase("hELLO"); // is true
```

Use a while Loop to Repeat Based on a Condition

You use a for loop when you need to run a piece of code a fixed number of times. But what if you aren't certain just how many times you need to loop? What if you wanted to loop as long as needed, as long as some other condition is still true? In that case, you'd use a while loop. A while loop will keep executing a piece of code as long as the Boolean condition is true:

```
while (stillHungry) {
    // ...
    // Something better set stillHungry to false!
}
```

In a way, it's kind of a mix between an if statement and a for loop: it loops over code the same way a for loop does, but it keeps looping as long as the condition is true, testing the condition like an if does.

And yes, if you forget to change the value to false, while will continue forever, and your entire Minecraft server will be stuck until you kill it or reboot or lose power, whichever comes first.

Try This Yourself

Now it's time for you to create a loop yourself. Let's go back to MyHouse.java, in ~/Desktop/code/BuildAHouse/src/buildahouse, and instead of creating just one house with this call:

```
BuildAHouse.buildMyHouse(width, height);
```

write a for loop that will run ten times, with the buildMyHouse call in the body of the for loop. That will make ten houses. Your own mini city!

Edit MyHouse.java and add your for loop, and then build and install the plugin as usual:

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

Stop and restart your server, connect with your client, and type /buildahouse again. You now have a set of ten houses!

Next Up

In this chapter you've learned something about Java syntax, from parentheses to squiggly brackets and semicolons. You know how to declare Java variables and use them to store important information. You can write Java functions that will act on your data, and you can control functions with `if`, `for`, and `while` statements.

Next we'll look at what happens when you package variables and functions together to make objects—the heart of a large system like Minecraft. Minecraft objects let you create plugins to manipulate everything in the Minecraft environment, from creepers to cows. Let's see how.

Your Growing Toolbox

You now know how to:

- Use the command-line shell
- Build with Java, `javac`
- Run a Minecraft server
- Deploy a plugin
- Connect to a local server
- Use Java variables for numbers and strings
- Use Java functions
- Use `if`, `for`, and `while` statements

In this chapter we'll cover a bunch of new Java language tools:

- Use *objects*: bundles of variables and functions
- Use the right *import* for a package
- Use new to create objects

CHAPTER 5

Plugins Have Objects

You may have heard of object-oriented programming or that Java is an "object-oriented language." That's what we're talking about in this chapter: using objects to represent elements of the Minecraft game, from players to cows.

You can do anything in code that you could do in the game, and then some. This comes down to working with mostly three kinds of things: blocks, items, and entities.

Everything in Minecraft Is an Object

The Minecraft world is filled with *blocks*. Every location in the game has a block, which might be made of air or another material. Blocks can exist in the world and in a player's inventory. Anything in a player's inventory is represented by an *item*.

In this world of blocks you have *entities*, which includes players, creepers, and cows. Items in motion are also entities—an arrow in flight or a snowball or a potion that's being flung. And all of these are objects.

Everything in our plugins is an object: locations, blocks, entities, cows, creepers, players, and even the plugin itself. All objects, all the time.

So now the real fun starts! You have variables holding data, and you have instructions expressed as functions, including some control statements to repeat bits of code or make decisions, and now we'll see how to put them all together into objects.

Try This Yourself

Let's try a little demonstration of objects in Minecraft. In the downloaded code there's a plugin called `NameCow`. Go there now and install that plugin, as shown in the following command line session.

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

Stop and restart your server, connect your client, and you'll be able to run the new `namecow` command. This command will spawn a new cow and give it a name. In the Minecraft client, type the command followed by that cow's name:

```
/namecow Bessie
/namecow Elise
/namecow Babe
```

You'll see these cows appear in the game, and when you look at each one, you'll see its individual name appear.

This plugin spawns a new `Cow` object each time you run it. Each cow you spawn has internal variables that keep track of its unique state: the cow's own name, its position in the Minecraft world, its health, and so on.

While it's important that each cow is represented by its own separate object, that's not a good enough reason to use objects. So why do we use objects at all? There is a deeper reason.

Why Bother Using Objects?

Imagine you are a god of your own universe. You have spent a few eons arranging every quark, every atom, every molecule, all the way up to planets and galaxies, just the way you want it. Now the whole thing spins up, and you're responsible for every single subatomic particle in the whole universe, all at once. Even as a godlike being, trying to keep the universe going by dealing with every electron or every quark or even every molecule is just too much work (not to mention incredibly boring).

So instead, you deal with problems on the scale in which they occur. If it's a problem with a planet in the wrong spot, you move the planet. If you need to fiddle with a galaxy, you fiddle with the galaxy—not with every planet, and certainly not with every life form on every planet in every system.

Although it might sound grandiose, creating a program is very much like that. You're a very powerful creator of your own little universe. Maybe not exactly godlike, but you do have to face that same issue of dealing with things at a very low level, like atoms or molecules, and at a very high level, like creatures, mountains, planets, and galaxies. All at once. When programming, you often have to zoom in to "atoms" and zoom out to "galaxies." They're all connected and have to make sense.

That's why we write code using objects. It's a way to organize data (in variables) and behavior (in functions) so that when we want to deal with a cow, we can treat it like a cow and not have to deal with each of the millions of atoms that make up said cow—or a biome, or a world, or a torch.

Even better, we can write code so that only a `Cow` has functions and data a cow needs. There's nothing worse than having assorted functions spilling out all over the place—you might end up with a torch that moos or a cow that lights up. Worse still, you'll end up with a huge pile of functions where you're not really sure *which* function can work with which piles of data.

For example, a `Cow` object in Minecraft has a lot of functions you can call. Here are a few of them:

- teleportTo(Location *location*)
- setFireTicks(int *ticks*)
- setAge()
- getAge()
- swingArm()

It also maintains some internal state: some variables to keep track of that individual cow's location, whether it's on fire, its age, and so on.

These are all fine things you can do with a `Cow` object. But they wouldn't all make sense for a player, or an arrow, or a tree.

You want to keep cow things in the `Cow` object, arrow things in the `Arrow` object, and so on. If you don't, then things can get awfully confusing awfully fast, and you might end up writing a pile of code that you can no longer understand or work with. So this is really just a matter of good hygiene, like keeping milk in the milk carton in the fridge and not storing it in the pretzel bag in the closet. You don't even want to mix corn chips with potato chips, to continue with this metaphor.

Keeping separate things separate in different objects is the easy part.

The hardest aspect of programming is this problem of having to deal with very low-level details and very high-level details at the same time. We call these *levels of abstraction.* Let's face it: when you're writing code to deal with a player in Minecraft, it's not the actual person playing the game. Somewhere there is an actual person, sitting there, sweating, eating chips and listening to loud music. Your piece of code is an abstract representation of that real-life player; an abstraction that includes the data and behavior you need for the game.

And just as in the real world, each abstraction can contain parts. So you can choose to focus on molecules or planets as you need to—or in software, on the molecule objects or planet objects. You can zoom in and out to the level you need to be at, and work with the parts you want.

Let's take a closer look at what that means in Minecraft and Java.

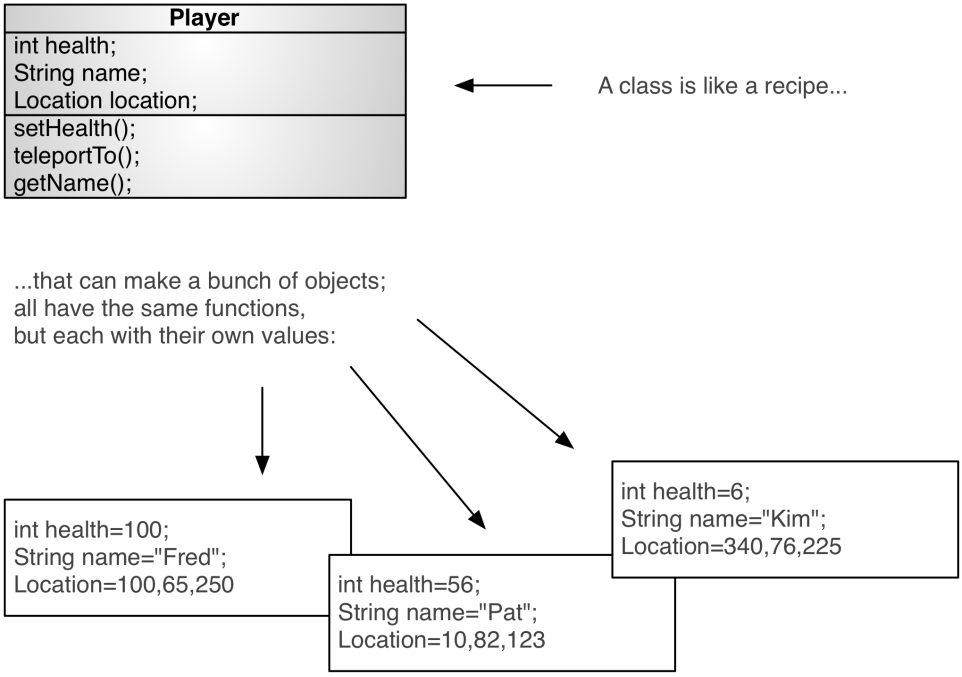
Combine Data and Instructions into Objects

Suppose you were to write a game like Minecraft from scratch. You'd want to have a bunch of players in the game. Each one would have its own name,

inventory, health points, and so on, but the structure of each player would be the same.

That is, every player object would have the same collection of variables (name, health, location, and so on) and the same functions that you'd want to run (set this player's health, teleport the player to a new location, that sort of thing). That's where objects come in. You'd create an object in Java to represent each player in the system. You'd write code to do things to a player, and that code would work no matter which specific player you were using at the time. That's exactly what the Minecraft folks did. Here's how the magic works.

In Java, you can define a pile of variables and a bunch of code that uses those variables, sort of like defining a recipe. You can then create and use an object that's built from that recipe. Java calls this kind of recipe a *class*. From that class recipe, Java will make running objects. Look at the following figure; here you have a few variables and some functions for a sample `Player` class, and some objects it can make.



It's just like building objects from Lego blocks: the blocks may all be the same, but when you follow the printed instructions that come with the kit, you can build a spaceship.

For instance, Minecraft keeps a pile of interesting data and functions related to each player who's online. Its "recipe" is defined by the class net.canarymod.api.entity.living.humanoid.Player, which builds objects for an individual Player. Note that before you can use Player in your code, you must add an import statement at the very top of the file—in this case you'd do an import net.canarymod.api.entity.living.humanoid.Player.

So how can we get at one of these player objects?

The Minecraft server knows who's online at any point, so we can ask the server for a list of Player objects. Or we can ask for a specific user by name, and it will give us a single Player object. Here's how.

Within a plugin, you can get access to the server using the function named Canary.getServer(). That will give you an object representing the server. Once you have the server object, you can ask it for the player you want. You use getPlayer and call it with the name of the player you want.

That sequence of instructions looks like the following. (We're going to look at this in bits and pieces first, and end up with a complete running example.)

You start off with the imports that define the class recipes you'll be using:

```
// Up top
import net.canarymod.Canary;
import net.canarymod.api.Server;
import net.canarymod.api.entity.living.humanoid.Player;
```

Then later, in your plugin code, you can use variables of those types (Player and Server):

```
Server myserver = Canary.getServer();
Player fred = myserver.getPlayer("fred1024");
```

Assuming that "fred1024" is online at the moment, we now have a variable named fred that represents the Player object. If Fred wasn't online, then the fred variable would be equal to the special value null, which is Java-speak for "there ain't one"—in other words, fred is not set to any object at all.[1]

Objects have parts. They contain stuff. An int, like 5, is just the number five. It can't do anything else; it can't store anything else alongside it. It just is what it is. But objects have functions and variables that you access with a period (.).

1. If fred was null and you tried to execute a function from fred, like fred.isSneaking(), you'd get an error and your plugin would crash.

Imports

Each object recipe in Java lives in a package, like java.util or net.canarymod.Canary, or something like that. You have to declare this package in an import statement at the very top of the Java source code file.

If you forget, you'll get an error from javac that says something like "cannot find symbol".

You need to look in the Canary documentation or the Java documentation to find the full package name. We'll use a lot of the common ones in our examples, for Player, Server, Location, Entity, and various Java libraries. For your convenience, these are all listed in Appendix 7, *Common Imports*, on page 253.

So with fred in hand, you can get—and set—data for that player, using the period (.) to indicate which function or variable you want inside that object. Here are a few snippets of code that show what that looks like:

```
// Is fred sneaking?
boolean sneaky = fred.isSneaking();

// Hungry yet?
int fredsHunger = fred.getHunger();

// Make him hungry!
fred.setHunger(0);

// Where's fred?
Location where = fred.getLocation();
```

The Player object has a function named isSneaking(), which returns a true or false, depending on whether that player is in sneaking mode.

Remember, that's the kind of thing you can use in an if statement:

```
if (fred.isSneaking()) {
    fred.setFireTicks(600); // Set him on fire!
}
```

There's also a getHunger() function that returns an int telling you how hungry that player is. You can set the player's hunger level as well. The last snippet here shows how to get the player's current location in the world.

As you might guess, objects contain internal variables that their functions work with. For instance, in these examples the Player object for Fred has a location stored internally. We can get the value of Fred's location, we can set a new value, but at all times Fred has his own internal copy of his current location.

You can also run some interesting commands. For instance, you can execute a command as if you were Fred:

```
fred.executeCommand("tell mary179 I love you")
```

Now Mary will think that Fred sent her a love note. Let's play with Fred's Location and see what other mischief we can create.

```
Location where = fred.getLocation();
```

Now the variable we named where will point to a Location object that represents Fred's location in the world.

Making Objects

You can also make a new location from scratch:

```
double x, y, z;
x = 10;
y = 0;
z = 10;
Location whereNow = new Location(x, y, z);
```

Or do it in one step:

```
Location whereNow = new Location(10, 0, 10);
```

And that's how to make a new object in Java: by using the new keyword.

When you use new to create an object, Java will create the object for you and run its *constructor*: a function that's named the same as the class (for instance, public Location()). The constructor gives you the chance to set up anything in the object that needs setting. It doesn't return anything and isn't declared with a return type; Java automatically returns the new object after you've done your setup.

We'll discuss how to make our own object definitions a bit later in the book, but for now we'll use what Minecraft has given us plus our plugin skeleton.

And now armed with a location, you can whisk Fred away:

```
fred.teleportTo(whereNow);
```

Suddenly Fred will find himself…suffocating in bedrock (because y in this location is zero). Ouch.

Plugin: PlayerStuff

Let's play around with a Player a bit more.

We're going to install the PlayerStuff plugin and change some of the object properties for players in the Minecraft world.

In the Player object, there are all kinds of interesting functions to get information about a player, and to set those values as well. Here are a few we'll look at:

chat()
getWorld()
getDisplayName() (can set it too)
getExperience() (can set it too)
getHunger() (can set it too)
getHealth() (can set it too)
isSleeping()
getLocation()

We can send a message to a player, get some values, and more. Here's code for a full plugin that demonstrates some of these features. It provides the command "/whoami". For more advanced plugins, this approach could be a great way to debug in-game objects by displaying information about Minecraft objects. When you're writing code and it's not working, printing out a couple of values of different variables is a great way to find out what's going on.

Here's the code, which is in the plugin directory Desktop/code/PlayerStuff. The full plugin has the necessary Canary.inf file and such:

PlayerStuff/src/playerstuff/PlayerStuff.java

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

public class PlayerStuff extends EZPlugin {

  @Command(aliases = { "whoami" },
           description = "Displays information about the player.",
           permissions = { "" },
           toolTip = "/whoami")
```

```
  public void playerStuffCommand(MessageReceiver caller, String[] parameters) {
      if (caller instanceof Player) {
        Player me = (Player)caller;
        String msg = "Your display name is " + me.getDisplayName();
        me.chat(msg);
        me.getWorld().setRaining(true);
        me.getWorld().setRainTime(100); // 5 secs
        float exp = me.getExperience();
        int food = me.getHunger();
        float health = me.getHealth();
        Location loc = me.getLocation();

        me.chat("Your experience points are " + exp);
        me.chat("food is " + food);
        me.chat("health is " + health);
        me.chat("you are at " + printLoc(loc));
        me.chat("water falls from the sky ");
    }
  }
}
```

Install that now:

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

Restart your server and reconnect your Minecraft client.

What happens when you type the command "/whoami" from the Minecraft client?

Here's what I get:

Let's walk through the code and see what's going on. We start off getting the player object me, and then the fun begins. Using the me object, we get the player's name and then send that as a message back to that player.

Then just for fun we'll make it rain (or snow) on the player by setting raining to true, and set the rainy time to 5 seconds with setRainTime(100).[2] There are 20 server ticks per second, so 100 server ticks will be about 5 seconds.

Next we'll get the experience points for the next level and the food level, and send those as a message back to the player. You can play around in this world for a while, and run "/whoami" to see if your food and experience have changed any.

It's just that simple: me is an object of type Player, and we can get values for various player values and send commands to me to do playerlike things.

That's what objects are for.

Try This Yourself

In the screenshot, you'll notice I also printed out a line that says whether or not you are sleeping. Add a local boolean variable in PlayerStuff and set it to a true or false depending on whether the player is sleeping or not, and then create an appropriate message to display either way.

Build the plugin with build.sh and try it out.

You can see how I did it in code/MyPlayerStuff/src/myplayerstuff/MyPlayerStuff.java.

Next Up

In this chapter you've seen how to use Java objects: how to import a Java package and class, how to use new to create objects, and how to change properties of objects that will affect the game. We'll need all of that for the following chapters.

In the next chapter we'll take a closer look at how plugins are wired into Minecraft, how to add commands, and how to find things in the Minecraft world.

2. You could make it stop by setting raining to false instead.

Your Growing Toolbox

You now know how to:

- Use the command-line shell
- Build with Java, `javac`
- Run a Minecraft server
- Deploy a plugin
- Connect to a local server
- Use Java variables for numbers and strings
- Use Java functions
- Use `if`, `for`, and `while` statements
- Use Java objects
- Use imports for Java packages
- Use `new` to create objects

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

- Add a command to a plugin
- Add new command annotations and code
- Work with Minecraft coordinates (Location)
- Find blocks or entities (BlockIterator)

CHAPTER 6

Add a Chat Command, Locations, and Targets

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. Canary has kindly defined a basic "recipe," a basic Plugin class that it knows about. I've added an additional EZPlugin class to make things a little easier. 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 EZPlugin:

```
import net.canarymod.plugin.Plugin;
import com.pragprog.ahmine.ez.EZPlugin;

public class MyFavoritePlugin extends EZPlugin {
```

That makes Plugin and EZPlugin parents of your class MyFavoritePlugin, just like the examples in the last chapter.

The Minecraft server already knows how to work with a Plugin, 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 Canary 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, Canary needs a configuration file for the plugin, named Canary.inf. You saw a description of this back on page 35, while we were building plugins the first time. It tells the server some basic information about your plugin, so that the server can load it.

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 creatures (not players) 50 blocks up into the air. Very handy at night with skeletons and creepers about.

Here's the whole source file to the plugin:

SkyCmd/src/skycmd/SkyCmd.java

```
package skycmd;
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.entity.living.EntityLiving;
import java.util.List;
import com.pragprog.ahmine.ez.EZPlugin;

public class SkyCmd extends EZPlugin {
  @Command(aliases = { "sky" },
           description = "Fling all creatures into the air",
           permissions = { "" },
           toolTip = "/sky")
  public void skyCommand(MessageReceiver caller, String[] parameters) {
    if (caller instanceof Player) {
      Player me = (Player)caller;
      List<EntityLiving> list = me.getWorld().getEntityLivingList();
      for (EntityLiving target : list) {
        if (!(target instanceof Player)) {
          Location loc = target.getLocation();
          double y = loc.getY();
          loc.setY(y+50);
          target.teleportTo(loc);
        }
      }
    }
  }
}
```

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

The bit with the @Command at ❶ tells the system that this function, skyCommand, is responsible for handling the /sky command. That is, when the player types the /sky command, your skyCommand function will be called.

The first thing we need to check is a little awkward; it turns out that the MessageReceiver that gets passed to us here may not be a Player. It could be a Player object, 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. (If it's not a Player, then it's probably a console command, if you want to allow those.)

The skyCommand function 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 67. Now that we've confirmed the variable caller is really of type Player (not just a MessageReceiver or any other parent or child), we can convert it to the type Player, using a *cast* operator.

So the expression (Player) caller returns the variable caller, 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.

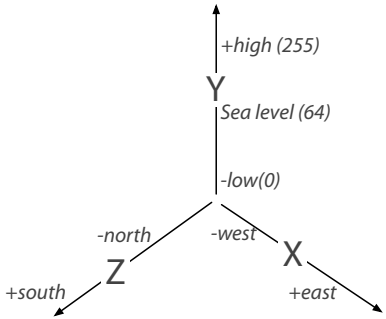
Now that we have a real Player object referenced by me, we can get the list of all living entities with me.getWorld().getEntityLivingList(), which will get us all the living entities in this world and return them in a List that we'll go through with a for loop.

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 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 with a teleportTo().

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.



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.teleportTo(loc) we tell the target to teleport itself to this new location.

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.

Try This Yourself

Well, that was fun! Let's try something a little different: adding a new command to this plugin, all on your own. Add a new command to the SkyCmd plugin that creates ten squid—a squid bomb.

You'll add annotations for a command named "squidbomb", and you'll use a for loop and one of our helper functions.

The spawnEntityLiving is documented to take a location and the thing you want to spawn:

```
spawnEntityLiving(newloc, EntityType.SQUID);
```

spawnEntityLiving returns an Entity, but we won't be using that right now.

For instance, to spawn a squid, you need to import net.canarymod.api.entity.EntityType up at the top of the file. Then later in your function, pass EntityType.SQUID to the spawnEntityLiving function. With that we can make a simple "squid bomb":

```
import net.canarymod.api.entity.EntityType;

//... other parts not shown

    // Spawning some squid. Derp.
    for (int i = 0; i < 10; i++) {
      spawnEntityLiving(location, EntityType.SQUID);
    }
```

Use that in your new command in our SkyCmd plugin. Don't forget to do the following:

- Add your new command (squidbomb) to SkyCmd using the @Command annotation as we've seen previously, and your new function.
- Recompile and install using build.sh.
- Stop and restart the server to pick up the change.

You'll need to add a new command annotation, which looks like this:

```
@Command(aliases = { "squidbomb" },
          description = "Drop a fixed number of squid on your head.",
          permissions = { "" },
          toolTip = "/squidbomb")
```

Now, that's a little bit boring—all the squid kind of pile on top of each other. It might be better to randomize the location for each squid. Java provides a function, Math.random(), that will give us a random number that ranges from 0 up to (but not including) 1.

To get a random number from 0 up to 5, just multiply Math.random() by 5. So for instance, to get a new x-coordinate you might use an expression like loc.getX() + (Math.random() * 5). When multiplying and adding, parentheses are usually a good idea—in this case we want to multiply the random 0..1 by 5, then add that to the original x.

Now it's your turn again: improve the squid bomb by making a new location based on the player's location that you already have, and add a bit of randomness to the x- and z-coordinates. To get the squid to drop on you from above, add 10 to the y-coordinate.

Try going through this exercise all by yourself first. In case you get stuck and need some help, I made a whole new plugin for the squid bomb. You can see my code and config file in code/SquidBomb.

Find Nearby Blocks or Entities

Canary provides a very handy feature in net.canarymod.BlockIterator. A BlockIterator lets you find all the blocks along a line in the game. Most useful is probably the version where you pass in a LineTracer (made from a Player) and a boolean, which is declared in the Canary API like this:[2]

```
public BlockIterator(LineTracer tracer, boolean include_air)
```

That gives you a BlockIterator object, which you can use to retrieve blocks along the line of sight from that entity. The boolean flag says to include Air blocks, and it works like this:

```
BlockIterator sightItr = new BlockIterator(new LineTracer(me), true);
while (sightItr.hasNext()) {
    Block b = sightItr.next();
    // do something with this block, b
}
```

You might check each block along this player's line of sight and find the first block that isn't Air. That would be the player's "target." Or you could set fire to each block along the way and then turn that target into Lava.

Here's a plugin that does exactly that.

Plugin: LavaVision

This plugin runs a BlockIterator for the player, and checks each block along the way, setting a flame effect. The first block that isn't Air is the target, so we'll set that to Lava.

2. This is a recent addition to CanaryMod, and doesn't yet appear in their docs.

For extra punch, I've added an effect on each block we traverse:

```
spawnParticle(b.getLocation(), Particle.Type.LAVASPARK);
```

The LAVASPARK gives us a nice set of sparks along our line of sight.

I've also added a sound effect, using the playSound helper function, which takes a Location, and a Sound (and optionally floats for volume and pitch if you want).

```
playSound(b.getLocation(), SoundEffect.Type.EXPLODE);
```

There's a list of possible effects in the Canary documentation under net.canarymod.api.world.effects.SoundEffect.Type. I picked EXPLODE for drama. (Optionally you can add the volume and pitch as two numbers after that; they are specified to be float, not double, so remember to add the f modifier.)

Here's the full code, with the iterator and sound effect:

LavaVision/src/lavavision/LavaVision.java

```
package lavavision;

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.effects.Particle;
import net.canarymod.api.world.effects.Particle.Type;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.effects.SoundEffect;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import net.canarymod.BlockIterator;
import net.canarymod.LineTracer;
import com.pragprog.ahmine.ez.EZPlugin;

public class LavaVision extends EZPlugin {

  @Command(aliases = { "lavavision" },
           description = "Explode your target into a ball of flaming lava",
           permissions = { "" },
           toolTip = "/lavavision")
  public void lavavisionCommand(MessageReceiver caller, String[] args) {
    if (caller instanceof Player) {
      Player me = (Player)caller;

      BlockIterator sightItr = new BlockIterator(new LineTracer(me), true);
      while (sightItr.hasNext()) {
        Block b = sightItr.next();
        spawnParticle(b.getLocation(), Particle.Type.LAVASPARK);
```

```
        if (b.getType() != BlockType.Air) {
          b.getWorld().setBlockAt(b.getLocation(), BlockType.Lava);
          playSound(b.getLocation(), SoundEffect.Type.EXPLODE);
          break;
        }
      }
    }
  }
}
```

Install this plugin in the usual way:

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

As we loop, we'll spawn a LAVASPARK effect for each block along the line of sight. If we hit something that isn't Air, then we'll set that block's type to Lava, play a sound effect, and break out of the loop. All done.

Stop and restart the server, and in Minecraft look around and pick a target. Type the /lavavision command and watch the lava bubble.

Now it's your turn!

Change the effects that this plugin uses. Change the particle effect type from LAVASPARK to something else interesting. All the available effect types are listed in the docs.[3]

3. https://ci.visualillusionsent.net/job/CanaryLib/javadoc/net/canarymod/api/world/effects/Particle.Type.html

Next, change the sound effect from an `EXPLODE` to something more subtle, like perhaps a `BURP`. The sound effects are listed in the docs too.[4]

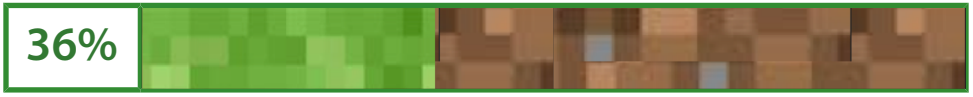
Next Up

In this chapter you've seen how to add a new command to a plugin. Now you can start adding your own new ideas to existing plugins, and you can work with locations and blocks in the game. But as soon as you start dealing with a bunch of locations or a bunch of blocks, you have a problem: how does Java store lists of things like that, and how do you work with "piles" of data that you might need to find by name or in order?

In the next chapter we'll add a command for remembering information: stuff you'll need to keep track of. We'll talk more about variables in Java: who can see them and who can't, and—most importantly—how to keep and work with piles of data.

In short, we're going to look at how to keep track of stuff.

Your Growing Toolbox



You now know how to:

- Use the command-line shell
- Build with Java, `javac`
- Run a Minecraft server
- Deploy a plugin
- Connect to a local server
- Use Java variables for numbers and strings
- Use Java functions
- Use `if`, `for`, and `while` statements
- Use Java objects
- Use imports for Java packages
- Use `new` to create objects
- Add a new command to a plugin
- Work with `Location` objects
- Find blocks/entities

4. https://ci.visualillusionsent.net/job/CanaryLib/javadoc/net/canarymod/api/world/effects/SoundEffect.Type.html

Is your head full yet? I hope not, because you have a few fun things left to learn. In this chapter we'll add these abilities to your toolbox:

- Use local variables that exist only within a block
- Use class-level global variables that can be accessed from anywhere in the class
- Keep piles of data in arrays

CHAPTER 7

Use Piles of Variables: Arrays

Now that we can create a plugin command to do something, we need to take a look at how to remember stuff—in other words, how to better use Java variables to keep track of values. We'll see when to use which kind of variable and how to work with piles of data in different ways (things like lists of Player objects), and we'll build a couple of cool plugins along the way.

Variables and Objects Live in Blocks

Variables live inside of *blocks*. We've seen and been using blocks all along: blocks are the bits of code written between braces—{ and }.

Java is a "block structured" programming language. It's descended from a family of languages going back to the ancient Algol of the 1960s, a line that extends from the mother of all programming languages, C, its children C++ and Objective-C, and down to its half-cousin-stepchild Java. After Java was around, Microsoft came out with C#, which is (totally coincidentally) nearly identical to Java. All of these languages work more or less the same way.

In these languages, you work with blocks of code. Things like if statements work with blocks of code. Objects we define (like our plugins) are blocks of code. The whole structure of the language is based on blocks of code within braces.

And that's where variables live. For instance, in the HelloWorld plugin, look at this section where we create and send a message:

```
public void helloCommand(MessageReceiver caller, String[] parameters) {
  String msg = "That'sss a very niccce EVERYTHING you have there...";
  Canary.instance().getServer().broadcastMessage(msg);
}
```

In the body of this function we declare and assign a variable named msg. It's a *local* variable—it lives only while this particular code block is running,

between its declaration and the closing brace (the } symbol). You cannot use this msg variable before that, or anywhere else in your program. You might have another variable named msg somewhere else, but it will be a totally different variable, with different values. This msg is visible and usable only locally, in this one block of code. We say that its *scope* is local.

The parameters that you declare in a function are considered local as well. Here in the declaration for the helloCommand function:

```
public void helloCommand(MessageReceiver caller, String[] parameters) {
```

the variables caller and parameters are all available within this function, but aren't visible anywhere else. They are local.

Most of the time local variables are all you need. In fact, local variables are pretty safe to use. No other part of the program can use or change them, and it's very clear what line of code set a local variable's value and where it is used.

Global Variables

But you can make variables that have a wider scope and aren't just local (and we're going to need to do that for our plugins in this chapter).

Maybe you've declared a variable that many different functions can use. Maybe your entire class, or even the entire program and all your libraries, can see it and change it. We call that a *global* variable.

We've used this before, but you haven't seen it yet. Let's take a sneak peek inside EZPlugin, which we used in the very first HelloWorld plugin. It has a class-level static variable declared right at the top, named logger.

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

```
public class EZPlugin extends Plugin implements CommandListener {
  /* Boilerplate methods for all of our plugins */
  public static Logman logger;

  public EZPlugin() {
    logger = getLogman();
  }

  @Override
  public boolean enable() {
    logger.info ("Starting up");
```

We make the call to getLogman and ask it for the logger object. It returns that to us, and we assign it to our variable named logger. logger is of type Logman,

which is what the documentation tells us it needs to be, but note that we've added the word static to this declaration.

Static here means two things:

- You access the variable using the name of the plugin without a plugin object. In this case, HelloWorld.logger will work from anywhere, inside or outside an object.
- The static variable (logger) is common to all HelloWorld objects, and lives on, outside the lifetime of any local variables in any object.

That means that any of the functions within a HelloWorld object can come and go, and their local variables can come and go, but the static variables will still be there and still remember their values.

Since logger is the very first thing we declare inside our plugin, and it's not inside a function itself, all the functions in our plugin—everything inside this top pair of { and } characters—can use it. It's not local to any one function.

That means you can use logger anywhere in HelloWorld.java:

```
...
public void myFavoriteFunction() {
    logger.info("Made it this far");
}
...
```

Notice that we don't need any kind of extra declaration; you can just use logger anywhere in this plugin that you'd like. That's a very handy technique to trace what's happening in a plugin: add a logger statement and you can print out the values of variables at that point in the code.

Global variables like these, however, can also be mighty dangerous.

Why are they dangerous? Precisely because *anyone*, anywhere, can change the value of that variable on you. Maybe you know they did it, but maybe you don't—even if "they" is "you," weeks from now. If something goes wrong, you have to go and examine every single piece of code in the system to try to find out which piece of code set the variable badly, and why. That's a lot of work and creates a lot of opportunity to mess things up.

But sometimes you really do need a global variable like that. You may not want anyone else to change it, but maybe a lot of different pieces of code need to refer to it. The logger is a good example: it's a shared service that all plugins and the server itself use. We all need access to the logging object, and we need it in a variable that won't go away. Unlike with a local variable, we want

it to be visible for all the functions in our plugin and to stick around as long as the plugin is around.

Here's a recap of our story so far:

- A block of code is written inside { and }.
- Variables you declare inside the body of a block of code are local to that block of code. They go away when the function finishes and returns.
- A function's parameters are local to that function.
- Variables declared as static will outlive any local variables declared within functions.

Here's a quick plugin that gives us a /caketower command. The idea is that it will build a tower of cake blocks. But there's a subtle problem in the code involving local and global variables, and the tower may not turn out the way you expect. Let's take a look.

Plugin: CakeTower

CakeTower/src/caketower/CakeTower.java

```
package caketower;

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.BlockType;
import com.pragprog.ahmine.ez.EZPlugin;

public class CakeTower extends EZPlugin {

  public static int cakeHeight = 100;                                    (1)

  @Command(aliases = { "caketower" },
           description = "Build a tall tower of cakes",
           permissions = { "" },
           toolTip = "/caketower")
  public void cakeTowerCommand(MessageReceiver caller, String[] parameters) {
    if (caller instanceof Player) {
      Player me = (Player)caller;

      me.chat("1) cake height is " + cakeHeight); // Print it
```

```
        cakeHeight = 50;

❷       int cakeHeight;
        cakeHeight = 5;
        me.chat("2) cake height is " + cakeHeight); // Print it

        makeCakes(me);

      }
    }

    public void makeCakes(Player me) {
      me.chat("3) cake height is " + cakeHeight);  // Print it
      Location loc = me.getLocation();
      loc.setY(loc.getY() + 2);
      setBlockAt(loc, BlockType.Stone);
      for(int i = 0;i < cakeHeight;i++) {
        loc.setY(loc.getY() + 1);
        setBlockAt(loc, BlockType.Cake);
      }
    }
}
```

When run, this code will print out the value of cakeHeight three times. Notice that there are two declarations of the variable cakeHeight, one at ❶ and another at ❷.

What will this code print out, and how many cakes will end up in the tower?

Try to figure it out first. Then compile and install using build.sh as usual.

What Happened?

Welcome to the wonderfully confusing world of *shadowing*.

In this piece of code, a variable named cakeHeight is declared at the top of the plugin. This is the variable you would expect to access anywhere within the plugin—starting on that line and ending at the matching closing brace, }.

But then on the line at ❷ we declare another variable with the very same name. From this point until the next closing brace, }, any time we mention cakeHeight we'll be working with this local one, not the class-level one. This local version *shadows* the class version. So when we set it to 5 and then print it, we're modifying this local version.

Calling makeCakes then uses the class version to build the tower. The makeCakes function has no knowledge of the shadowed variable inside the cakeTowerCommand function. So you end up with fifty blocks; not a hundred, and not five.

Moral of this story: don't do this. As you can see, shadowed variable names can be very confusing. Give your variables unique, memorable names.

Try This Yourself

Since the cake tower doesn't quite work as expected, let's fix it!

Change code/CakeTower/src/caketower/CakeTower.java to use just the one local variable cakeHeight, not the class-level variable, and pass it in to makeCakes.

Now that you know where variables live and can be used, let's look at a couple of different ways you can use piles of data, using Java data collections.

Use a Java Array

While variables with individual values are useful and common, sometimes you need more than that. You need to keep track of all the players in the system, or one player's inventory items, or a to-do list, or a grocery list, or a homework list.

Java has you covered. There are several different ways to keep and access piles of data. We're going to focus on a few: the simple Array, the classier ArrayList, and the remarkably handy if somewhat alien HashMap (covered in the next chapter). First up, the Array.

There are actually a few different Array-like collections in Java, including Array, Vector, LinkedList, and ArrayList types. They each work differently on the inside, are stored slightly differently, and perform differently with large or small data sets, but the idea is the same for all of them.

Arrays are probably the simplest of these piles of data. Arrays are mostly used when you have a small list of values that you want to create directly in code and then use. Arrays are *fixed length*—you can't grow or shrink them. They aren't as useful if you need to add, delete, and move things around a lot in the list (for that, we'll use an ArrayList, which is up next).

You'll usually employ an Array when you want to access its values by an index, or run through all the values and do something to them with a for loop.

You can declare an Array using square brackets, and load it up with values using braces. Here's an example of a list of Strings:

```
String[] grades = {"A", "B", "C", "D", "F", "Inc"};
```

You can access individual elements from the list using brackets:

```
String yourGrade = grades[2];
```

In this case, yourGrade will be a C. Hey, wait a minute—why is that a C, and not a B? That's because Java, like the C language and its predecessors, starts counting at 0. The first element in any list is 0. The second is 1. The third is 2, and so on. You'll get used to it. Think of accessing the first element as adding 0 to the start of the list, and the second element as adding 1 to the start of the list.

That's exactly how an Array is stored in memory in the computer: just a bunch of values all in row. Since the first entry in the Array is right at the start of the memory, it has no offset. The second value is one over from the start, the third is two over from the start, and up you go.

You can tell the length of an Array by looking at its length field (note this is not a function call; there are no parentheses):

```
int numGrades = grades.length;
```

numGrades will be set to 6. That means that each of the six values will be numbered 0 to 5. The index of the last element is always length-1 (in this case, 6-1, or 5).

Instead of sticking in all the values hard-coded as we did, you could make an Array that's a fixed size, then stuff some values into it. Here's what that looks like with a list of int values:

```
int[] quizScores = new int[5];
quizScores[0] = 85;
quizScores[1] = 92;
quizScores[2] = 63;
```

Although there's space for 5 values, we're only using the first 3 here, and that's okay.

Getting values out looks just like putting values in, only the other way around. Using the code we just looked at, you retrieve the values like this:

```
int myBestQuiz = quizScores[1];
int aBadDay = quizScores[2];
```

To get all the values, you can use an old-fashioned for loop:

```
for (int i=0; i < 5; i++) {
    me.chat("Quiz score #" + i + ": " + quizScores[i]);
}
```

Remember that Array is a fixed size; if you try to retrieve a value that's past the end of the array (like quizScores[15]), your plugin will throw an error and crash. In this case, since we define the quizScores array to have a size of 5, you can safely store and retrieve values at index 0, 1, 2, 3, and 4. That's why we use i < 5 in the middle, instead of <=.

It's a lot safer to use the Array's .length field instead of hard-coding a number like "5." So it would be better to write that loop like this:

```
for (int i=0; i < quizScores.length; i++) {
    me.chat("Quiz score #" + i + ": " + quizScores[i]);
}
```

In a little bit we'll discuss an even better way to loop through all the values in an array.

Let's do the same thing now, but with Minecraft blocks.

Here's an example of code for a quick plugin that builds a tower of different block types. I'm using our helper function setBlockAt to change air into one of several different materials.[1] Let's walk through this and see what's happening.

Plugin: ArrayOfBlocks

ArrayOfBlocks/src/arrayofblocks/ArrayOfBlocks.java

```
package arrayofblocks;

import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
```

1. The codes for different materials are in the Canary API docs at https://ci.visualillusionsent.net/job/CanaryLib/javadoc/net/canarymod/api/world/blocks/BlockType.html.

```
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 ArrayOfBlocks extends EZPlugin {

  public void buildTower(Player me) {
    Location loc = me.getLocation();
    loc.setX(loc.getX() + 1); // Not right on top of player

    BlockType[] towerMaterials = new BlockType[5];

    towerMaterials[0] = BlockType.Stone;
    towerMaterials[1] = BlockType.Cake;
    towerMaterials[2] = BlockType.OakWood;
    towerMaterials[3] = BlockType.Glass;
    towerMaterials[4] = BlockType.Anvil;

    for (int i=0; i < towerMaterials.length; i++) {
      loc.setY(loc.getY() + 1); // go up one each time
      setBlockAt(loc, towerMaterials[i]);
    }
  }

  @Command(aliases = { "arrayofblocks" },
           description = "Create an array of blocks",
           permissions = { "" },
           toolTip = "/arrayofblocks")
  public void arrayofblocksCommand(MessageReceiver caller, String[] args) {
    if (caller instanceof Player) {
      Player me = (Player)caller;
      buildTower(me);
    }
  }
}
```

Install the ArrayOfBlocks with build.sh, stop and restart the server, and try the /arrayofblocks command. You should see something like Figure 1, *Array of Blocks*, on page 98 (you might need to turn around to see it).

Notice that I put the guts of the command in its own function, buildTower, instead of in the arrayofblocksCommand function itself.

This is just a simple Array of length 5 that we are loading up with values one at a time. The for loop goes from index 0 to 4 and changes the block to the new material in our list.

Figure 1—Array of Blocks

Try This Yourself

Now it's your turn. Make a small change to reverse the order of the tower's elements, so that the anvil is on the bottom and the stone is on the top.

Change the `for` loop around to do this. Instead of going from 0 to < 5, change the loop to go from 4 down to >= 0. Hint: Subtraction might work better than addition in this case.

Rebuild, stop, start, and try the `/arrayofblocks` command again.

Now you know how to work with `for` loops and indexes, but to be honest, this is an old corner of Java, and it's a tad musty. `Array` objects are handy, but there's perhaps a better choice.

Use a Java ArrayList

An `ArrayList` in Java also keeps track of a list of values, just like a simple `Array` does. `ArrayLists` are a little messy to declare but simple enough to use, and much more flexible and a bit safer than plain old `Array` objects. You can add and delete from the `ArrayList` as many times as you want; it's not a fixed length and will grow or shrink as needed.

Here's an example of an `ArrayList` that will hold `Player` objects:

```
List<Player> myPlayerList = new ArrayList<Player>();
```

There's a lot more gunk in there than we've seen up to now. First of all, I've called our new list myPlayerList. Note that funny syntax with the angle brackets, < and >. You have to specify the kind of list you're making (twice, in fact). Java 7 improves on this a little; you don't have to repeat it on the right-hand side and you can say List<Player> whatever = new ArrayList<>(). Also, in a small bit of weirdness, notice that although it's List on the left, it's ArrayList on the right. The reason for that is...because Java. (Okay, the reason is that List is the parent and ArrayList is one particular child.) Moving on now.[2]

With a list you've created, you can do a lot of fun things, like adding and removing values from the list, retrieving values, and checking to see if a value exists. Here's a bit of sample code:

ListPlay/src/listplay/ListPlay.java

```
public void listDemo(Player me) {
1  List<String> listOfStrings = new ArrayList<String>();
2  listOfStrings.add("This");
   listOfStrings.add("is");
   listOfStrings.add("a");
   listOfStrings.add("list.");

3  String third = listOfStrings.get(2);
   me.chat("The third element is " + third);

4  me.chat("List contains " + listOfStrings.size() + " elements.");

5  listOfStrings.add(3, "fancy");

6  boolean hasIt = listOfStrings.contains("is");
   me.chat("Does list contain the word 'is'? " + hasIt);

   hasIt = listOfStrings.contains("kerfluffle");
   me.chat("Does the list contain the word 'kerfluffle'? " + hasIt);
    // Print out each value in the list
   for(String value : listOfStrings) {
     me.chat(value);
   }

7  listOfStrings.clear();
   me.chat("Now it's cleared out, size is " + listOfStrings.size());

   hasIt = listOfStrings.contains("is");
   me.chat("List contains the word 'is' now is " + hasIt);
}
```

2. "Because Java" is a bit of an Internet joke; see http://www.theatlantic.com/technology/archive/2013/11/english-has-a-new-preposition-because-internet/281601 for details.

Try This Yourself

Read that over and see if you can figure out what it will do when run.

Does it all make sense? Let's go through the code and see what's going on.

First, at ❶ there's the new to create a list that will hold Strings. So far so good. Next, starting at ❷ we're adding a couple of strings to this list, one at a time, using the List function add(). With these added to the list, we can now try to get some data out.

On the line at ❸ we're getting the third element of the list—by asking for the list index of 2. Remember, it's zero-based counting, just like Array, which we talked about earlier. The third element is the string "a".

Next we check to see how many elements are in the list, using the function size() at ❹, and it tells us there are four.

One of the advantages of an ArrayList over an Array is that you can easily add and remove values—even in the middle of the list, as seen here on the line at ❺, where we're using add() and passing in an index of 3. That will add this value at index 3 in the list and move all the other values down one.

You can also remove values, read values, and so on, as much as you like. No matter how we add or shuffle values around in the list, we can check to see if a particular value is in the list without having to look through the whole list, as with the call to contains that looks for the word "is" at ❻. Cool, it's in there. After that we'll try again with a value that *isn't* in there. And indeed, the contains function returns false for "kerfluffle".

You've seen how you could get a single value by index using the get function, but what if you want to go through the list one by one?

You could use a for loop as we did with Array, but that's old-fashioned, ugly, and error prone.

Instead, you can use a *for-each* construct. The statement for(String value : listOfStrings) acts like a for loop that iterates over the collection listOfStrings, and in the body of the loop it will set the variable value to each entry as it goes through. We first saw this back in SkyCmd.

Whew! That's a lot of explanation for a few short lines of code. But it's a powerful idea, and we're going to use this in a plugin in just a bit.

Finally, what happens when we clear out the list entirely (at ❼)? Not much interesting, as it's empty now.

Plugin: ArrayAddMoreBlocks

Let's play with this a bit. We'll start with the ArrayOfBlocks plugin but change it to use an ArrayList instead of an Array. We'll call the new plugin ArrayAddMoreBlocks.

Since we can add to the array list easily, let's make it static:

```
public static List<BlockType> towerMaterials = new ArrayList<BlockType>();
```

Then, thanks to the wonder of ArrayList, we can add a couple of blocks to the new tower each time we call /arrayaddmoreblocks:

ArrayAddMoreBlocks/src/arrayaddmoreblocks/ArrayAddMoreBlocks.java

```
public void buildTower(Player me) {
  if (towerLoc == null) {
    towerLoc = new Location(me.getLocation());
    towerLoc.setX(towerLoc.getX() + 2); // Not right on top of player
    towerBase = new Location(towerLoc);
  }

  towerMaterials.add(BlockType.Glass);
  towerMaterials.add(BlockType.Stone);
  towerMaterials.add(BlockType.OakWood);

  for (BlockType material : towerMaterials) {
    logger.info("Building block at " + printLoc(towerLoc));
    setBlockAt(towerLoc, material);
    towerLoc.setY(towerLoc.getY() + 1); // go up one each time
  }
}
```

To reset the list, you use the clear() function, and I've set up a separate command to do just that:

ArrayAddMoreBlocks/src/arrayaddmoreblocks/ArrayAddMoreBlocks.java

```
public void clearTower() {
  if (towerLoc == null) {
    return;
  }
  while (towerBase.getY() < towerLoc.getY()) {
    setBlockAt(towerBase, BlockType.Air);
    logger.info("Clearing block at " + printLoc(towerBase));
    towerBase.setY(towerBase.getY() + 1); // go up one each time
  }
  towerLoc = null; // Reset for next tower
  towerBase = null;
  towerMaterials.clear();
}
```

One last interesting note about variables: notice that at ❶ and ❷, I used a `new` to make a new variable, as in `towerBase = new Location(towerLoc)`, instead of just assigning it, as in `towerBase = towerLoc`. Why do you think I did that?

Remember that when you create a variable with `new`, the actual variable is sitting out in memory someplace, and your name is just a name. So if I had typed `towerBase = towerLoc`, there would only be one `Location`, with two names. They wouldn't be separate variables. By typing `towerBase = new Location(towerLoc)` I made a whole new variable named `towerBase` that has copied the values from `towerLoc`.

If you're planning on changing the values of a variable, and you don't want to change the original, always make a copy.

The full plugin is in `code/ArrayAddMoreBlocks`. Build, stop, start, and try that now, building and clearing some towers. Remember, you might need to turn around and face a different direction to see the tower.

Try This Yourself

Now change the code and add a couple of different building materials. Then try changing it to use just one material, like `Dirt` perhaps. Or make a nice mix of `Dirt` and `Grass`. You're in control.

Next Up

In this chapter you've learned the difference between local and global variables. You can use a simple `Array` or the more flexible `ArrayList` to store a pile of data, and traverse it using a for-each iterator.

Arrays are great if you don't really care about finding one of the objects in the array by itself. However, if you care about objects by name—like a `Player`—then you'll need something a little fancier.

In the next chapter we'll cover how to use a `HashMap` to store data by name (or by `Location`, or by anything else you might need).

Your Growing Toolbox

You now know how to:

- Use the command-line shell
- Build with Java, `javac`
- Run a Minecraft server
- Deploy a plugin
- Connect to a local server
- Use Java variables for numbers and strings
- Use Java functions
- Use `if`, `for`, and `while` statements
- Use Java objects
- Use imports for Java packages
- Use `new` to create objects
- Add a new command to a plugin
- Work with `Location` objects
- Find blocks/entities
- Use local variables
- Use class-level global variables
- Use `ArrayLists`

In this chapter we'll get a little fancier with piles of data and plugin structure, and add these abilities to your toolbox:

- Keep piles of data in hashes
- Use private to hide your private data, and use public for things other people should see and use

CHAPTER 8

Use Piles of Variables: HashMap

Use a Java HashMap

HashMap is a funny name. The "hash" part refers to how it works internally; it really has nothing to do with how you use it.

But the "map" part is about what you'd think: it maps a key, which can be anything, to a value, which can also be anything.

Other languages might call this a *dictionary*, an *associative array* or some kind of *associative memory*. They all mean the same thing. With an array, you use an integer as the index. With a HashMap, you can use any object as the key, especially Strings (see Figure 2, *A HashMap*, on page 106).

We'll use this a lot in the plugins. It's a great way to keep track of players, cows that you've spawned, or anything else you want. Although it kind of works like an array, you don't use it the same way. Back when we looked at an array you learned that you can get and set a value with an index like this:

```
myList[9] = "Andy";
String who = myList[9];
```

But you can't use that kind of assignment and bracket notation with a HashMap. Instead you use its put and get functions. On the plus side, though, now you can use anything as a key. Most of the time you'll probably use a String. So suppose you have a HashMap cleverly named myHash.

```
myHash.put("Andy", "www.PragProg.com");
// Works like myHash["Andy"] = "www.PragProg.com";

String myUrl = myHash.get("Andy");
// Works like myUrl = myHash["Andy"];
```

Now myUrl will have the value of "www.PragProg.com".

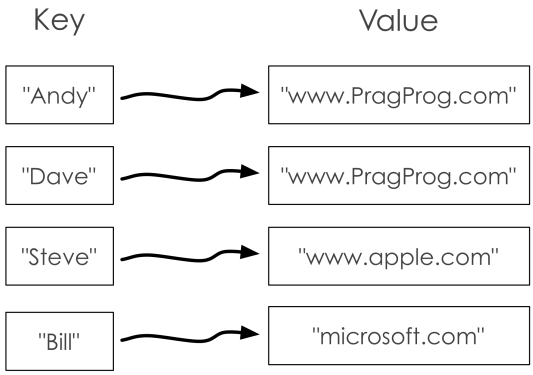


Figure 2—A HashMap

HashMaps are a great way to store a lot of data that's indexed by a string, just like a dictionary. You look up a word and get a bunch of data. You can also use a HashMap to store a bunch of named properties for an object. The nice part about this is that the property names are just Strings. You could add new properties or delete them while the plugin is running—something that's tough to do with hard-coded function or variable names.

Let's play with these a bit. Here's a small standalone program where we can experiment with a HashMap. Maybe you'll make a plugin that implements some kind of *Hunger Games*–style contest, where you'd want to have a score for each player. You'd keep track of each player's score using a HashMap, like this:

HashPlay/src/hashplay/HashPlay.java

```
1 HashMap<String, Integer> currentScores = new HashMap<String, Integer>();

  public static void addToScore(HashMap<String, Integer> allScores,
                    String playerName,
2                   int amount) {
3   int score = allScores.get(playerName);
4   score += amount;
5   allScores.put(playerName, score);
  }
```

Here's how you'd use it:

HashPlay/src/hashplay/HashPlay.java

```
currentScores.put("Andy", 1001);
currentScores.put("Bob", 20);
currentScores.put("Carol", 50);
currentScores.put("Alice", 896);
addToScore(currentScores, "Bob", 500);
me.chat("Bob's score is " + currentScores.get("Bob"));
```

Compiling and running that in a plugin gives us the expected answer of 520 for Bob. Let's go over the code:

On the line at ❶ you see the call to `new()` that creates a new `HashMap` for us to use. Since the `HashMap` has to know what we're going to use for a key and a value, we have to pass in the type names using the angle brackets (< and >), just as we did for an `ArrayList`—except here we need to pass in two types: one for the key (which is a `String`) and one for the value (an `Integer`).[1]

With a new `HashMap` named `currentScores` in hand, we can go through and create some test entries in our hash with player names and their scores. This time, though, we'll do something a little different.

We're going to make a new helper function that will increment a player's score. You can see this starting at ❷. It's a simple function that does three things, but it's a good habit to get into. Any time you have to do a series of steps a couple of times, don't write them out and copy and paste. Instead make a function to do the work for you, then just call the function when you need it.

Our function `addToScore()` is declared to take three arguments: the `HashMap` of all scores, the string with the player name, and the integer value to add to the score. With this data, we're doing three steps:

1. Get the current score for the player whose name is in `playerName` (on the line at ❸).
2. Increase that score by the amount you passed in as `amount` (at ❹).
3. Save that newly incremented score back into the hash (at ❺).

Note that we're not returning any particular value from this function, as it's declared to be `void`, just like `main` is. Instead it's modifying the `HashMap` that's passed in; in this case, the global `currentScores`.

Most of the time, though, you'll want functions that return a useful value. For instance, here's a short, trivial function that adds 10 to any number you pass in:

```
public static int addTen(int originalNumber) {
    int newNumber = originalNumber + 10;
    return newNumber;
}
```

1. Why `Integer` and not `int`? Because Java. When using collections like `HashMap` and `ArrayList`, you have to refer to primitive types (`int`, `float`) by their class names, `Integer` and `Float`. The magic of *autoboxing* takes care of converting from `Integer` to `int` and so on. Just use the capitalized full names in the angle brackets, and life goes on.

All we've done differently is specify what kind of value we're going to return (int in this case) instead of using void in the declaration. Then we use the keyword return with the value that we want to return to our caller. Usually you want to call return as the very last thing in your function. That's because it specifies what value to return *and* performs the return right then and there. No more code will be run in your method after it hits the return. You're done.

Try This Yourself

Modify the HashPlay.java source so that no one's score can go below 0 or above 1,000. Use a helper function that returns a value to clamp the score to be between 0 and 1,000.

You can see my solution in HashPlayClamp/src/hashplay/HashPlay.java. In fact, you might see another subtle trick in there: instead of making the helper function public, I declared it to be private. What's that all about?

Keep Things Private or Make Them Public

So far, we've tended to use the Java keyword public when making static variables and defining functions and plugins. That tells Java that the thing we're defining should be publicly accessible—all of our own plugin code can use it, and any other plugin in the system can use it as well (like when we use logger).

There is another option. You can create functions or variables or even helper objects that *no one else* can see. Instead of public, you can make them private.

In programming, there's a simple rule—so simple it's the kind of thing you'd tell a five-year-old: don't expose your privates.

In other words, if you're using a function or something that *only* you should use, then mark it as private to make sure that no one from the outside can use it. Why would you need to do that?

Suppose that somewhere in your plugin you have a function to mark a player as a super, high-level, über-Wizard. You wouldn't want any other plugin to call that function on a player of its choosing. So where normally you'd declare the function in the plugin like this (leaving out all the other bits):

```
public class WizardingWorld extends EZPlugin {
    ...
    public void makeSuperUberHighWizard(Player p) {
        ...
    }
    // Any other plugin can call WizardingWorld.makeSuperUberHighWizard()
}
```

Stack vs. Heap

You may have noticed that sometimes we create and directly assign a simple value to a variable, like `int a = 25`, and sometimes we use the `new` keyword to create an object, like a plugin.

These two kinds of objects are stored differently. *Immediate values* like integers and floats are kept on Java's list of function calls that you're making. That list is called a *stack*. It's like a stack of pancakes. Each new function call throws a new pancake on the stack, and when it's done you remove that pancake and you're back to the previous one. When a function is finished, its "pancake" is thrown away, and any of its local variables disappear.

But objects that you create with `new` are kept off in a big pile of memory we call the *heap*. They can stick around after your function is gone, if you want them to (like a plugin does).

You just need to keep a variable somewhere that points to your object; that variable can be local and passed around, or global. Java keeps track of how many different variables reference the object created with `new`. Once no one is using that object anymore, it gets tossed in the trash. And then when the system feels like it, it empties the trash and your object is gone. (Java even calls that "garbage collection.") And as we mentioned toward the end of the last chapter, you need to be careful: assignment gives you two references to the same object; using `new` creates a new copy of an object.

you can instead make it `private`, like this:

```
public class WizardingWorld extends EZPlugin {
    ...
    private void makeSuperUberHighWizard(Player p) {
        ...
    }
    // Only functions here can call makeSuperUberHighWizard()
}
```

In general, if you're making a function that you're using internally within this plugin, and other plugins shouldn't call directly, then make it `private`. Variables you are using should almost always be private.

If you want other plugins to see and use it, then make it `public`.

We'll start using `private` for our helper functions now.

Plugin: NamedSigns

Let's put a couple of these ideas together and make a plugin that uses helper functions (with and without return values), `private` functions and variables, some low-level array access, and a hash that will store names and locations.

This plugin lets you create signposts in the game and name them. You can then put text on any one of the signposts by name. For example, suppose I make two signs, named `one` and `two`, by typing these commands in the chat window:

```
/signs new one
/signs new two
/signs set one Hello!
/signs set two Goodbye!
```

I'd see something like this:

Now I can go in and change either sign's text at will just by issuing another `signs set` command, like this:

```
/signs set two Adios!
```

Let's start with the plugin in all its glory, then look at the interesting pieces.

NamedSigns/src/namedsigns/NamedSigns.java

```
package namedsigns;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
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.World;
```

```
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import net.canarymod.api.world.blocks.Sign;
import com.pragprog.ahmine.ez.EZPlugin;

public class NamedSigns extends EZPlugin {
(1)  private static Map<String,Location> signs = new HashMap<String,Location>();

(2)  private void usage(Player me) {
       me.chat("Usage: signs new name");
       me.chat("       signs set name message");
     }

     private void parseArgs(Player me, String [] args) {
(3)    if (args.length < 3) {
         usage(me);
         return;
       }
       if (args[1].equalsIgnoreCase("new")) {
         makeNewSign(me, args);
       }
       if (args[1].equalsIgnoreCase("set")) {
(4)      if (args.length < 4) {
           usage(me);
           return;
         }
         setSign(me, args);
       }
     }

     // signs new sign_name
(5)  private void makeNewSign(Player me, String [] args) {
       Location loc = me.getLocation();
       loc.setX(loc.getX() + 1); // Not right on top of player
       int y = loc.getWorld().getHighestBlockAt((int)loc.getX(),(int)loc.getZ());
       loc.setY(y);
       signs.put(args[2], loc);
       setBlockAt(loc, BlockType.SignPost);
     }

     // signs set sign_name line1
(6)  private void setSign(Player me, String [] args) {
       String name = args[2];
       String msg = args[3];
       if (!signs.containsKey(name)) {
         // No such named sign
         me.chat("No sign named " + name);
         return;
       }
```

```
    Location loc = signs.get(name);
    World world = loc.getWorld();
7   Sign sign = (Sign)world.getTileEntity(world.getBlockAt(loc));
    sign.setTextOnLine(msg, 0);
    sign.update();
  }

  @Command(aliases = { "signs" },
           description = "Create and name signposts",
           permissions = { "" },
           toolTip = "/signs new name, or /signs set name message")
  public void signsCommand(MessageReceiver caller, String[] args) {
    if (caller instanceof Player) {
      Player me = (Player)caller;
      parseArgs(me, args);
    }
  }
}
```

There's a lot of code here, but it's really just a handful of simple parts and things we've seen already. These are the major pieces:

- A private static HashMap at ❶.
- A private helper function that returns nothing (void) at ❷. This function prints a usage message to the player. We'll call it if we find out the player didn't type in the command correctly.
- A private helper function at ❺. This function will create new signs.
- A private helper function at ❻. This function will change the name on existing signs.
- A check of the length of the args array on the lines at ❸ and ❹. Since the user may not have typed in enough words for us to use, we have to check that the length is long enough and indicate an error otherwise.
- A wee bit of magic at ❼. This is how the Canary docs say to get a Sign object from a Block.

That's the gist of it. Now let's go through each of those pieces in detail.

The signs HashMap

First off, we set up a HashMap named signs to keep track of Locations by name (a String). When we're creating a new sign, we'll stuff its location in the hash, using the name the user gave us. When we go to set text on a sign, we'll get the sign's location back out of the hash by using the name.

The parseArgs Function

Since there are a couple of things to check for when the user types in a command, I've split that out into its own function instead of doing it right in the signsCommand function.

Notice we're using a parameter from our command function that we haven't used before: String[] args. These are any arguments that the player types in with the command in the chat window. For example, if we type in

```
/signs
```

that will be passed to signsCommand in the args array as args[0]; args will have a length of 1. If we type

```
/signs set one Hello!
```

then args[0] will still be "signs", args[1] will be "set", args[2] will be "one", and args[3] will be "Hello!".

We know that we need at least three values in the args array: it will either be "signs new name" or "signs set name something".

Remember that with arrays, you have to check that the array is long enough; otherwise you'll get a lengthy error message from the server that ends up with something like this:

```
Caused by: java.lang.ArrayIndexOutOfBoundsException: 1
        at namedsigns.NamedSigns.makeSign(NamedSigns.java:26)
        at namedsigns.NamedSigns.signsCommand(NamedSigns.java:47)
        ... 15 more
```

We're checking the length of the arguments ourselves. But we could also set the "min == 3" part in the @Command annotation. That way if we don't have at least three arguments, the system will send a nice message to the player with our toolTip as a help message.

Safe in the knowledge that we have *at least* three values in the args array to work with, let's see what the player actually typed in.

The "/signs new" Command

Here's the part for the "new" command:

NamedSigns/src/namedsigns/NamedSigns.java

```
// signs new sign_name
private void makeNewSign(Player me, String [] args) {
  Location loc = me.getLocation();
  loc.setX(loc.getX() + 1); // Not right on top of player
  int y = loc.getWorld().getHighestBlockAt((int)loc.getX(),(int)loc.getZ());
```

```
  loc.setY(y);
  signs.put(args[2], loc);
  setBlockAt(loc, BlockType.SignPost);
}
```

First we grab a handy block next to the player (getX() +1) and get the highest block at that location with getHighestBlockAt(). That way we won't be putting the sign underwater or in bedrock or anything.

Next we save this block's location to the hash, using the name the player gave us (args[2]).

Finally we set that block's type to BlockType.SignPost. Now it's a sign.

The "/signs set" Command

And here's the part for the "set" command:

NamedSigns/src/namedsigns/NamedSigns.java

```
// signs set sign_name line1
private void setSign(Player me, String [] args) {
  String name = args[2];
  String msg = args[3];
  if (!signs.containsKey(name)) {
    // No such named sign
    me.chat("No sign named " + name);
    return;
  }
  Location loc = signs.get(name);
  World world = loc.getWorld();
  Sign sign = (Sign)world.getTileEntity(world.getBlockAt(loc));
  sign.setTextOnLine(msg, 0);
  sign.update();
}
```

Notice that we're setting two local variables, name and msg, to args[2] and args[3]. Why bother? Aren't they the same thing? Yes, they are, but it's a lot easier to read name instead of reading args[2] and trying to remember that 2 is the name and 3 is the message.

Next we'll check that we really have an entry in the hash for name, and if not we'll complain to the player. Otherwise we can safely get the location for the sign block.

Next is the bit of magic. We can get the block at the right location, but it's still a block—a net.canarymod.api.world.blocks.Block. A Block doesn't know anything about the functions of a Sign. A Sign is one kind of Block, so we'll have to convince Java to make a net.canarymod.api.world.blocks.Sign out of it.

The Canary documentation tells us to call world.getTileEntity() on the block, and then cast that using the cast operator (Sign) to make a proper sign. With the sign variable in hand we can call the two Sign functions we need: setTextOnLine() and update().

setTextOnLine() puts a line of text on the sign at the given index (0 in this case), and update() makes sure that the sign is redrawn in the client so you can see the new text.

It might look like a fair bit of code, but if you look at each piece one at a time, it's not so hard. In fact, now it's your turn to make some changes.

Try This Yourself

As we have it here, the plugin only sets the first line of the sign, but you can have up to four lines per sign. Modify the plugin so that if the user types in extra words, you'll pass each word to sign.setTextOnLine(). Remember: if it can hold four lines, then they are numbered 0, 1, 2, and 3. We've got the setTextOnLine(msg, 0) already, so you'll need to add the others if args is long enough.

Next Up

In this chapter we've seen how to use a HashMap to keep track of important game data by name or other object, and how to make functions private so that other plugins can't access them, or public so that they can.

In the next chapter we'll do more than just respond to user commands. We'll see how to listen to the Minecraft server and respond to game events as they happen, and even create a few of our own.

Your Growing Toolbox

You now know how to:

- Use the command-line shell
- Build with Java, javac
- Run a Minecraft server
- Deploy a plugin
- Connect to a local server
- Use Java variables for numbers and strings
- Use Java functions
- Use if, for, and while statements
- Use Java objects
- Use imports for Java packages
- Use new to create objects
- Add a new command to a plugin
- Work with Location objects
- Find blocks/entities
- Use local variables
- Use class-level global variables
- Use ArrayLists
- Use HashMaps
- Use private and public to control visibility

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 120.

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},
```

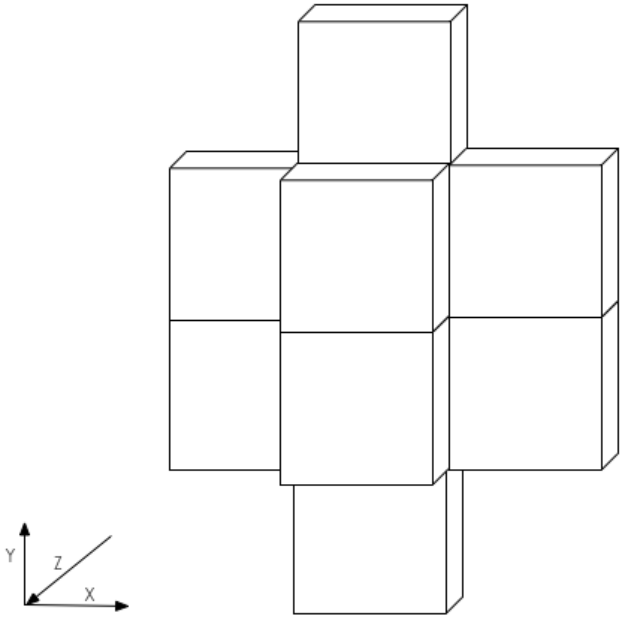


Figure 3—Trapping a player in blocks

```
  {-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

```
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

```
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

```
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.

Listen for Events

Now we get to the best part. You know how to write code for commands the user types in and you know how to do things to affect the world. Now it's time to see how to monitor what's going on in the world so you can respond to gameplay automatically, without typing in a command or anything.

It works like this: you set up your code such that your functions get called when some interesting event happens. In your function, you can let the events happen or you can stop them.

Here's a skeleton of what we need to add to our basic plugin in order to incorporate a listener. There are four parts to it:

1. Import the plugin listener and the event hook(s) classes.
2. Declare that your plugin implements PluginListener.
3. Register to listen for events with registerListener.
4. Add the magical tag @HookHandler, marking your function as an event handler. Give the event you want as an argument.

You'll see all four parts in this short example skeleton of a listener:

```
import net.canarymod.hook.HookHandler;
import net.canarymod.plugin.PluginListener;
// import the particular hook class here

// Add "implements PluginListener" here:
public class HelloWorld extends EZPlugin implements PluginListener {

  @Override
  public boolean enable() {
    Canary.hooks().registerListener(this, this);
    return super.enable(); // Call parent class's version too.
  }

  // Here's one event listener:
  @HookHandler
  public void anyname(SomeHook hookevent) {
        // Some code goes here
  }
}
```

First off, you have to import the class for the event you're interested in (you'd add this somewhere around ❶).

There are a *ton* of events available, all listed in the Canary documentation under net.canarymod.hook and its children. Suppose you're interested in doing something whenever someone in the game teleports. You'd want the TeleportHook class in the package net.canarymod.hook.player, so first thing here you'd import net.canarymod.hook.player.TeleportHook.

Then the declaration for the plugin needs to add the magic words implements PluginListener, as shown at ❷.

Next you need to add your own enable() function, which then calls the parent class's enable as shown starting at ❸. This is a standard piece of boilerplate code that just says "make this plugin listen for events." You need this only once in this file; it will work for all events you'll use. Add it in, and off we go.

Finally we come to the event listener itself, starting at ❹. That @HookHandler thing is an annotation that tells Java that the next function is special, just

like we've seen with @Command. Always start off an event handler with that special annotation.

The function for the listener itself can be named anything you want (shown here as anyname). But the argument list is important: the type of event you list here determines when this function will be called, or if it gets called at all.

An event handler for the TeleportHook would look like this:

```
@HookHandler
public void myTeleportListener(TeleportHook event) {
    // Some code here
}
```

I made up the name myTeleportListener, but that part doesn't matter—it's the TeleportHook that's important. According to the documentation, this event hook object has several interesting functions we can use:

getCurrentLocation()	Return the location the player is teleporting from
getDestination()	Get the location this player is teleporting to
getPlayer()	Get the player
setCanceled()	Lets you cancel this event

For instance, to prevent anything from teleporting anywhere, you could write this:

```
@HookHandler
public void myTeleportListener(TeleportHook event) {
    event.setCanceled();
}
```

Plugin: BackCmd

Let's use some of these features in a complete plugin. Here's the full source for BackCmd that provides a single command named back. Go ahead and build and install it:

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

And restart the server.

Now teleport to a couple of locations, either by using the /tp command in creative mode or by right-clicking on an Ender Pearl in survival mode.

Now type /back, and you'll be teleported back to your last location.

This plugin will listen for events to keep track of where you've been, and let you return to previous locations in order.

BackCmd/src/backcmd/BackCmd.java

```
package backcmd;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Stack;
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.hook.HookHandler;
import net.canarymod.hook.player.TeleportHook;
import net.canarymod.plugin.PluginListener;
import com.pragprog.ahmine.ez.EZPlugin;


(1) public class BackCmd extends EZPlugin implements PluginListener {
(2)   private static List<Player> isTeleporting = new ArrayList<Player>();
      private static HashMap<String, Stack<Location>> playerTeleports =
          new HashMap<String, Stack<Location>>();

      @Override
      public boolean enable() {
(3)     Canary.hooks().registerListener(this, this);
        return super.enable(); // Call parent class's version too.
      }

(4)   public boolean equalsIsh(Location loc1, Location loc2) {
        return ((int) loc1.getX()) == ((int) loc2.getX()) &&
               ((int) loc1.getZ()) == ((int) loc2.getZ());
      }

      @HookHandler
(5)   public void onTeleport(TeleportHook event) {
        Player player = event.getPlayer();
        if (isTeleporting.contains(player)) {
          isTeleporting.remove(player);
        } else {
          Stack<Location> locs = playerTeleports.get(player.getName());
          if (locs == null) {
            locs = new Stack<Location>();
          }
          locs.push(player.getLocation());
```

```
      locs.push(event.getDestination());
      playerTeleports.put(player.getName(), locs);
    }
  }

  @Command(aliases = { "back" },
           description = "Go back to previous places that you teleported to.",
           permissions = { "" },
           toolTip = "/back")
  public void backCommand(MessageReceiver caller, String[] args) {
    if (caller instanceof Player) {
      Player me = (Player)caller;

      Stack<Location> locs = playerTeleports.get(me.getName());

      if (locs != null && !locs.empty()) {
        Location loc = locs.peek();
        while (equalsIsh(loc, me.getLocation()) && locs.size() > 1) {
          locs.pop();
          loc = locs.peek();
        }
        isTeleporting.add(me);
        me.teleportTo(loc);
      } else {
        me.chat("You have not teleported yet.");
      }
    }
  }
}
```

As this plugin is using event listening, here are the special things you need to do:

- At ❶, declare that this plugin implements Listener.
- At ❸, register the plugin for events.
- At ❺ add the @HookHandler annotation and declare that this event listener will listen for TeleportHook events.

Every time a player teleports, the onTeleport function will be called. The first thing we do is get the correct Player object, which is stored in the event that was passed in to us. Now we know who we're dealing with.

Next we're looking in the list named isTeleporting to find out whether this player is someone we are in the middle of teleporting. What's isTeleporting, you ask?

Ah, take a look up at the top of the plugin around ❷, and you'll see two variables declared to be private static. That means no one else can see them, and the values will stick around in between this command being run. So here

we have isTeleporting and playerTeleports to keep track of players and the locations they've teleported to and from.

We're checking to see if our isTeleporting list contains this player already. If it doesn't, we'll add it—this player has just started teleporting. If the isTeleporting list already contains this player, then we should ignore this event (and remove the player from the list). That means we generated this event ourselves, and we don't want our own teleport to trigger the teleport, because it would keep doing that forever. So we throw this extra teleport event out.

If this player has teleported before, we'll get a locs from the playerTeleports hash (a hash of locations) using the player as a key. We'll add this location to the existing list. If this is the player's first time teleporting, we'll make a new list (of Locations) and add this location as its first entry.

But this is no ordinary list. It's a kind of list, all right, but it's a Stack, not a plain list. A Stack works like a stack of pancakes. You add to the top of the stack, and when you remove a pancake it comes off the top of the stack. That's the model we want for our list of places we've teleported. As you teleport, each location is added to the top of the stack—and it's the top one you want to go back to next, then the one under that, and so on.

To add an object to the top of a stack, you push it on. To get the value and remove the object at the top, you pop it off. To just check on the top value without removing it, you can peek at it.

As players teleport around, we'll keep a stack of each of their teleport locations, stored in a Stack in our playerTeleports variable. We can get to the stack using the Player object as a key. You can see what this looks like on page 132.

Finally, we get down to backCommand. When we get a back command, we'll get the player's list of teleport locations (if there is one), and peek at the location on the top of the stack, saving that in loc for now.

Maybe we teleported to this spot and haven't moved yet. In that case, we'd want to go to the previous spot—not stay at this location. So we'll test for that: if we're still at the same location as the last teleport (or close to it), we'll remove that location from the locs list by using the pop, and set loc to the next location in the stack.

Either way, we've got a loc pointing to the spot that we want to teleport to. We'll add the player to the isTeleporting list and teleport the player to the new location.

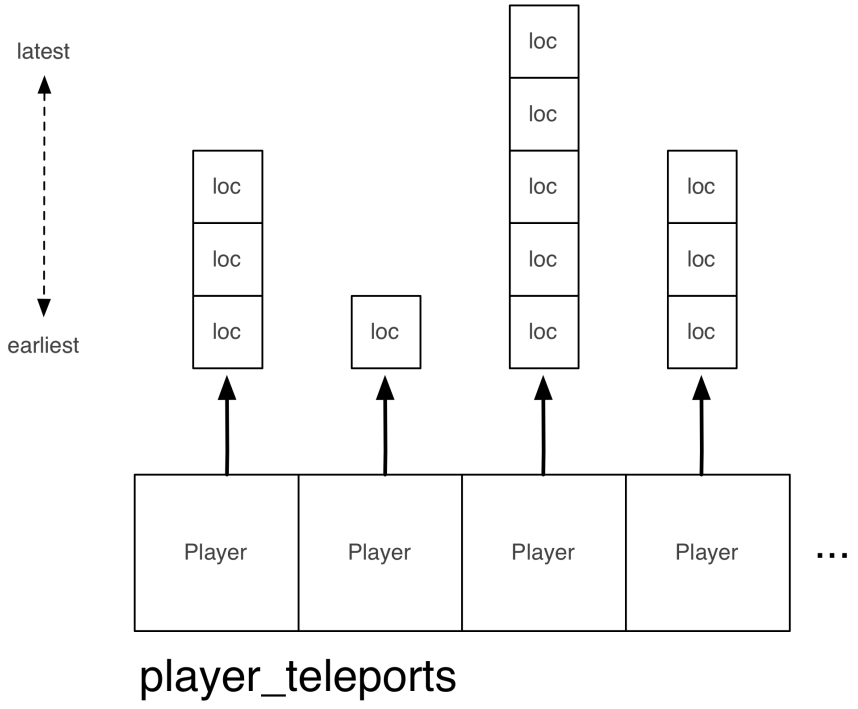


Figure 4—playerTeleports is a hash with Player keys and Stack values.

Now what do we mean by "or close to it"? Notice that we're using a helper function named equalsIsh (declared at ❹). It's checking two locations to see if the x- and z-coordinates are within the same block. By casting the floating-point values to integer (using the (int) keyword), we're throwing out the fractional part of the coordinate. We're interested only if you're in the same block, not at a slightly different position within the same block.

Try This Yourself

For this exercise, you're going to create a brand-new plugin that uses a listener.

Create a plugin named FireBow. If you fire an arrow, we want to make it explode in a huge, crater-making explosion when it hits something. To do this, listen for a ProjectileHitHook and call world.makeExplosion. Use a large value for the power, like 100.0f to make a really large crater. Don't forget to cancel the arrow event; that way, there will be no arrow—only a boom.

Hint: You can get the arrow from the ProjectileHitHook event using event.getProjectile().

My example of code that does all this is in code/FireBow. Try it yourself first, from scratch, and if you get stuck take a look at the example. I added two commands that allow you to enable or disable the firebow behavior.

Check Permissions

Sometimes when writing a plugin, you might want to restrict the commands you've created—maybe only ops should run them, or maybe you have different kinds of players or teams, and each should only be able to run certain commands.

For instance, suppose you want to restrict the firebow commands to a select set of players. We'll invent a new name; let's call it commands.firebow.enable.

You can then use the permissions field in the @Command annotation to check if the player issuing the command is one of the chosen few:

```
@Command(aliases = { "nofirebow" },
          description = "Disable firebow behavior",
          permissions = { "commands.firebow.enable" },
          toolTip = "/nofirebow")
```

Now only players who have the permission for commands.firebow.enable can shoot the massively exploding firebows. Great. So how do you grant players these permissions?

Setting and Managing Permissions

Setting and managing permissions can be a big deal if you're running a large server with lots of users. You might set up groups or classes of users in the server, using the Canary permissions commands.

Canary provides a bunch of in-game commands, including commands to set permissions for players and groups, and to check what permissions someone has. You can run these commands in the server console, or from inside the client. Here are some examples:

- /playermod permission add *playerName permissionName*—grant the *permissionName* to *playerName*.
- /playermod permission check *playerName permissionName*—check if *playerName* has the permission *permissionName*.
- /playermod group set *playerName groupName*—add *playerName* to the group *groupName*.
- /groupmod permission list *groupName*—list permissions for *groupName*.

- /groupmod permission add *groupName permissionName*—add the permission *permissionName* to members of *groupName*.
- /groupmod permission check *groupName permissionName*—check if members of *groupName* have the permission *permissionName*.

For example, in our plugin, to give jack37 permissions to use our /firebow command, you'd type the following commands in the console:

```
playermod permission add jack37 commands.firebow.enable
```

Check out the Canary docs for more details.

Next Up

We got a lot of new abilities in this chapter: you can now modify Minecraft blocks and spawn entities, manage plugin permissions, and, most importantly, listen for interesting events to happen in the game and react to them. That's the real heart of plugins that can change how the Minecraft world works by automatically acting on events as they occur.

In the next chapter we'll look at how to schedule events in the future, and make an exploding cow shooter. See you there.

Your Growing Toolbox

You now know how to:

- Use the command-line shell
- Build with Java, javac
- Run a Minecraft server
- Deploy a plugin
- Connect to a local server
- Use Java variables for numbers and strings
- Use Java functions
- Use if, for, and while statements
- Use Java objects
- Use imports for Java packages
- Use new to create objects
- Add a new command to a plugin
- Work with Location objects
- Find blocks/entities
- Use local variables
- Use class-level global variables
- Use ArrayLists
- Use HashMaps
- Use private and public to control visibility
- Modify Minecraft blocks
- Modify and spawn entities
- Listen for and react to game events
- Manage plugin permissions

In this chapter you'll learn more about objects, classes, and scheduling tasks that will run later. You'll add this knowledge to your toolbox:

- How to create a separate class
- How to schedule a task object to run later
- How to schedule a task object to keep running later

Your toolbox is nearly complete now. You can write some really cool plugins just based on what we've done so far.

CHAPTER 10

Schedule Tasks for Later

Now we'll talk about how to make things happen in Minecraft that aren't in direct response to either an event or a user-issued command: how to schedule tasks that will happen sometime in the future—and even keep running—all on their own. This feature helps you implement things that seem to act independently, like attacking creepers or other enemies.

On the Java front, you'll also see how to make your own classes in separate files, and we'll take a look at some of the problems Java has with running multiple things at once.

What Happens When?

When we talk about running something "later," it exposes the ugly truth that the world doesn't stop changing just because your code is running. Computers can do more than one thing at once, and are doing that all the time. But most code isn't written that way.

Think about some of the plugins we've looked at so far. What would happen if, right in the middle of running that piece of code, another player typed the same command and that same code started running from the top?

Take a look at the figure on page 136. Each arrow and code snippet represents a set of instructions that the computer is running at the same time. We call each separate set a *thread*—kind of like the thread of a subplot in a novel, or the thread of one conversation in the middle of many others at lunch.

Normally you think of the computer running through your code once, as in the left part of the diagram. You'd think the computer executes instructions one after another, from top to bottom, and nothing else is going on at the same time. That would be one thread of execution (like one conversation).

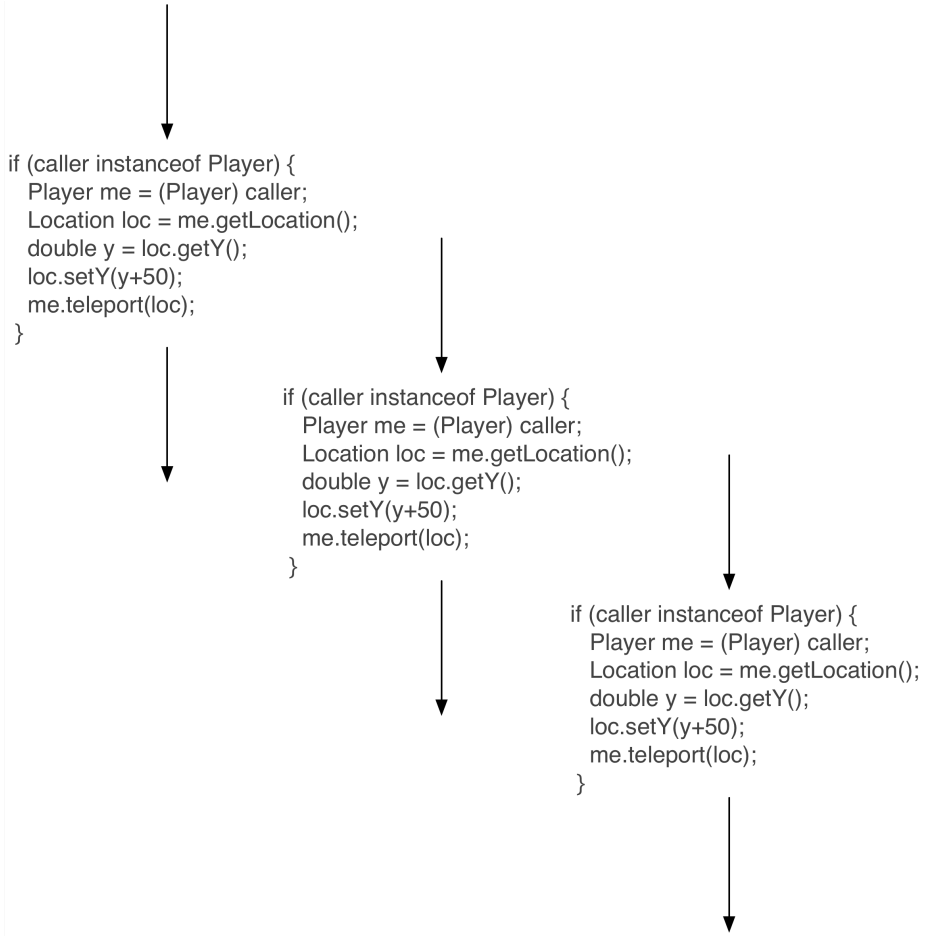


Figure 5—Various threads executing code

But in fact that's not the case; the computer can be running a bunch of different threads of execution, apparently all at the same time. Can our code work that way? What would happen to our code if it were run by multiple threads?

If you're using only local variables and aren't trying to change anything in common in the Minecraft world, that would probably be fine—each version of the code as it's running would have its own copy of variables, and no harm done. But what if you're using a static variable, something like a big `HashMap` to keep track of players and values? Then you have two bits of code reading and writing from it at the same time, and that generally leads to disaster.

Imagine that you read the player's health as 50 in one thread, but then a second thread sets it to 0—killing the player. You don't know that, and you subtract 10 from the player's health and reset it to 40. Now the other thread thinks it's killed the player, but instead of 0 the health has "magically" been set to 40! That's a simple and innocent example, but far worse things can happen that could cause strange bugs or make your code—and the server—crash.

There are ways to write code that can be run by multiple threads safely; however, many libraries, and most of the Canary API, are not designed that way.

That's important enough to repeat: *the Canary API, and our plugin code that uses it, are not built to be run by multiple threads.* The code will break.

So when we want to run a piece of code "later," we can't actually let it run at some random time. In particular, we can't let a piece of our code run at the same time as another piece of our code, or at the same time as the server code.

Instead, we have to let the server determine when to run the code. Then it can run it pretty much as if it were a player typing in a command or responding to an in-game event; it's the only thing running at the time. We calls that a *synchronous* task, and that's what you'll learn how to set up in this chapter.

But to set up a task, we'll first see how to make our own classes.

Put Code in a Class by Itself

First, let's be a little more precise in what we name things. In Java source code, you declare a plugin to be a `public class`, which is Java's way of saying, "This is a recipe. I can make objects of this class at runtime." While I've been loosely talking about recipes and parent recipes, I'm going to start calling them by their proper Java name: *classes*. "Recipes" are classes.

So far we've sort of cheated—we've been putting all new code in our one plugin class, in one source-code file. When the Minecraft server runs, it makes one object from that class file and uses that.

But if you look at other plugins out on the Internet, you'll notice that often a plugin (and larger programs) are made up of several different classes, all working together. The main class file has the plugin itself, while other class files might contain related objects, or tasks to be scheduled, or any other kind of helper code that the plugin might need. We've been adding code right

in our main class, but we aren't going to do that with schedule-able tasks; we'll define that code in its own class.

Mechanically, making another source-code file is easy. If you can hit your text editor's File -> Save (or something similar), you can make a new file. As long as you've saved it in the correct directory, the build system will see the new file and know what to do. But that's not enough. You need to know how to tell the plugin about the new file, and the new file about the plugin.

What Should I Put in a Class?

Beginning programmers might look at a class as just a box to toss functions and data into. Overall, that's a bad idea—just like a messy attic or garage, you end up with a lot of boxes full of junk spilling out and tangled up in each other.

Instead, any class you create should be responsible for just one thing. In other words, any class should have only one reason to change. If it's responsible for a lot of different things, then suddenly it's got a lot of reasons it may need to change: it's fragile, and fragile code leads to suffering.

In this case, the Canary API guides us in the right direction. We will be making our plugin class, but in order to make a scheduled task, we need to make a second class for it as well, so we'll end up with two classes: the plugin and the task. We'll need to get them to work with each other as well.

So let's see what it takes to make a task that we can schedule to run (when the server thinks it's okay to run it) sometime in the future. We'll look at the bits and pieces first, and then put it all together in a cool plugin.

Make a Runnable Task

Here's the simplest code for a task that just sends a broadcast message. It may not do much, but you can use this piece of code as a starting template when making your own tasks.

Just add the code you want down in the body of the run function, change the package name, and add any imports you may need. (Remember, the common ones are listed in Appendix 7, *Common Imports*, on page 253, and you can find others in the Canary or Java docs.)

```
1 package examplepackage;
  import net.canarymod.Canary;
  import net.canarymod.tasks.ServerTask;
  import com.pragprog.ahmine.ez.EZPlugin;
```

```
❷ public class ExampleTask extends ServerTask {

    private final EZPlugin plugin;

❸   public ExampleTask(EZPlugin plugin) {
      super(Canary.getServer(), 0, true); // delay, isContinuous
      this.plugin = plugin;
      // you can keep a reference to your plugin as I've done here,
      // or to any other variable you pass in
    }

    public void run() {
      // Do something interesting...
      Canary.instance().getServer().broadcastMessage("Surprise!");
    }
}
```

In this case, you'd put this code in its own file, which must be named ExampleTask.java (to match the name on the line at ❷), and be in a directory named after the package (to match the name at ❶).

So far so good, but what's up with this interesting-looking function that has no return type and is named ExampleTask—the exact same name as our class? Remember, that's called a *constructor*. That's the function Java calls when it's creating a new ExampleTask.

So to create our ExampleTask, you have to pass in a EZPlugin to the new (which gets passed to your constructor). In other words, you have to give it your main plugin object. That way this task can get at all the server things it might need to, like the world, players, and so on. That's not mandatory, but it can be useful.

Now that you have a task that will do something interesting (well, sort of), how do you schedule it?

Schedule to Run Later

Now back to your main plugin: once you have the task defined over in its own class, you can create an object of that class from your main plugin using new, and schedule it to be run in the future using addSynchronousTask().

```
ExampleTask task = new ExampleTask(plugin);
Canary.getServer().addSynchronousTask(task);
```

plugin should be a variable that's set to your main plugin. If you're putting this code in a non-static function in your plugin, you can use the Java keyword this, which means "this plugin object." If you're putting this code in a static function, you need a static variable (like plugin) that has previously been set

to the plugin. For example, in an object function (a *method*) like your constructor or enable, you'd set plugin = this.

You might want to hang on to the task variable, but it's not necessary. With or without it, you've created an ExampleTask which will schedule the run function to be executed sometime in the future.

Schedule to Run Once, or Keep Running

You can specify how your task is run by setting a delay until it starts, and setting whether it should be run just once or keep running. Have a look at the first thing we do in our constructor: the call to super at ❸. The call to super will in turn call the constructor in the parent class, ServerTask, with the needed information.

The ServerTask constructor takes three arguments: the server itself, a delay to wait before running, and a boolean flag that should be true if this task should keep running continuously, or false if it should be run only once (a one-shot). If you set it to run continuously, the task will run forever until you cancel it.

We could change the previous example to delay for 60 seconds before running once:

```
super(Canary.getServer(), 1200, false);
```

At 20 ticks per second, 1200 ticks will be about 60 seconds (one minute).

Plugin: CowShooter

Now we have enough parts to make a really fun plugin: the CowShooter. If you have a piece of leather in your hand in the game world, you can click to shoot a flaming cow out into the world. When the cow hits the ground, it will explode in a ball of flame, fire, and hamburger.

This plugin is a little different from what we've seen so far; there's no @Command section at all. It's driven entirely by events.

One problem with a flaming cow is that it won't stay flaming forever, or even for very long. Normally a flaming cow will catch fire, then the cow dies and the fire goes out. That's not quite suitable for our purposes: we need the cow to stay on fire as long as it's flying through the air.

To make that happen, we'll schedule a task. The task will keep repeating as long as the cow is in the air, and in that task we can keep the cow alive and on fire until it hits the ground.

Here's the code for the main event listener:

CowShooter/src/cowshooter/CowShooter.java

```
@HookHandler
public void onInteract(ItemUseHook event) {

  Player player = event.getPlayer();

  if (player.getItemHeld().getType() == ItemType.Leather) {
    Location loc = player.getLocation();
    loc.setY(loc.getY() + 2);

    Cow victim = (Cow)spawnEntityLiving(loc, EntityType.COW);

    Canary.getServer().addSynchronousTask(new CowTask(victim));

    fling(player, victim, 3);
    victim.setFireTicks(600);
  }
}
```

The ItemUseHook event can mean the player used one of many different items, so first we'll need to check and see if the player is holding leather. If the answer is no, we just ignore the event and life goes on.

But if the answer is yes, we'll use our fling helper method to increase the cow's velocity—we want to fling it in the direction the player is facing. To do that, we'll bump up the velocity by multiplying it by 3, as that seems to look pretty cool.[1] So beginning at ❶ we spawn a cow, add the new scheduled task, set its velocity using fling, and light it on fire.

Now as mentioned, there's a problem with a flaming cow (or anything on fire, really). It won't stay on fire for long; it will burn up and die. So that's what our scheduled task will take care of: it will keep the cow alive and watch for it to land.

Here's the separate class that contains the runnable task:

CowShooter/src/cowshooter/CowTask.java

```
package cowshooter;

import net.canarymod.Canary;
import net.canarymod.api.entity.EntityType;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.entity.living.animal.Cow;
import net.canarymod.api.world.position.Vector3D;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import net.canarymod.tasks.ServerTask;
```

1. Sometimes you just have to experiment.

How Do I Get a Piece of Leather?

For testing or just playing you might not have a piece of leather handy. Here's how to get one:

- Go into creative mode by typing /gamemode c.
- Press E.
- The icon in the upper-right corner is a search bar; click it.
- Start typing leather, and you should see the Leather material appear.
- Click it once, then click again on your hotbar (the bottom row of slots in the inventory) to put it there.
- Close your inventory by pressing the escape key, Esc.
- Press the corresponding number (1 through 9) to select the box where you put the leather.

When you try to go into creative mode you might see an error saying that you don't have permission. In that case, you'll need to "op" yourself—that is, give yourself operator privileges.

```
import com.pragprog.ahmine.ez.EZPlugin;

public class CowTask extends ServerTask {

    private Cow cow;

    public CowTask(Cow myCow) {
        super(Canary.getServer(), 0, true); // delay, isContinuous
        cow = myCow;
    }

    public void run() {
      if (cow.isOnGround()) {
        Location loc = cow.getLocation();
        cow.setHealth(0);
        cow.kill();
        cow.getWorld().makeExplosion(cow,
          loc.getX(), loc.getY(), loc.getZ(),
          2.0f, true);
        Canary.getServer().removeSynchronousTask(this);
      } else {
        cow.setFireTicks(600);
        cow.setHealth((float)cow.getMaxHealth());
      }
    }
}
```

If the cow has hit the ground (tested with cow.isOnGround()), then we make the cow explode and remove this task so it won't run again. We're done. But if the cow is still flying through the air, then we'll make sure it's still on fire with cow.setFireTicks(600) and bump up its health to the maximum to keep it alive a little longer. That actually looks a lot crueler in writing....

The result? You wave your leather as a magic wand, and shoot flaming cows that explode on impact.

The full code for both CowShooter.java and CowTask.java is in code/CowShooter/src/cowshooter.

Now isn't that a lot more fun than printing Hello, World?

Next Up

In this chapter you saw some Java mechanics, including how to create a new class and more on creating objects from classes. But the cool part is being able to run tasks in the background that keep running, which lets you implement cool things like flaming, exploding cows.

Next we'll cover how to save data and remember it despite the game being shut down, rebooted, or turned off.

Your Growing Toolbox

You now know how to:

- Use the command-line shell
- Build with Java, `javac`
- Run a Minecraft server
- Deploy a plugin
- Connect to a local server
- Use Java variables for numbers and strings
- Use Java functions
- Use `if`, `for`, and `while` statements
- Use Java objects
- Use imports for Java packages
- Use `new` to create objects
- Add a new command to a plugin
- Work with `Location` objects
- Find blocks/entities
- Use local variables
- Use class-level global variables
- Use `ArrayLists`
- Use `HashMaps`
- Use `private` and `public` to control visibility
- Modify Minecraft blocks
- Modify and spawn entities
- Listen for and react to game events
- Manage plugin permissions
- Create a separate class
- Schedule a task to run later
- Schedule a task to run periodically

Now we'll add some storage capability to your toolbox, and discover some more advanced Java language features. You'll add these skills:

- Save and load user-editable configuration data from a local file
- Save and load your private game data from a database
- Build up more complex code from smaller functions
- Use Canary's DataAccess to use their built-in database functions
- Understand Java exceptions and annotations

CHAPTER 11

Use Configuration Files and Store Game Data

Now we're getting someplace. From what you've learned so far, you can listen for events, schedule tasks to run later, keep data around in lists or hashes, and let your whole plugin use it as long as the server is up and running. That puts you in a good position to build some fun plugins. However, you're missing an important piece: servers don't run forever.

Any Minecraft server (including yours) can just stop running. You've stopped yours a bunch of times by now, just to install plugins! Even if you weren't installing new plugins, hardware can crash, or maybe you just closed your laptop. And once the server stops running, your plugin will forget everything it used to know. All its data is gone.

So we need to figure out how to save important data on disk somewhere, where we can save it often and load it back in when we need to.

There are two different kinds of data that you might care about:

- Configuration data, which contains things you need to know about how the plugin should work. This is probably set only once and updated rarely, but it's read every time the plugin runs. You've probably used configuration files in other plugins or in the server setup for Minecraft itself.
- Game data, which contains things like player scores, inventory, and stuff that changes frequently as the game plays. That gets a little tricky, as we'll see shortly.

Let's look at each of these types of data and how to save and load them.

Use a Configuration File

Many plugins, and the server itself, use *configuration files* to fine-tune and customize different values and select different behaviors. A configuration file is just a simple text file, designed to be read and edited directly by a person. The idea is that you can customize aspects of the plugin without having to fool around with the source code or recompile anything. End users, server operators, developers—everyone can use them. And now we will too.

We're going to modify a plugin to use a configuration file. Previously, on page 83, we built a plugin that dropped a bunch of squid on your current location—a "squid bomb." We're going to add configuration options to the SquidBomb plugin:

- The number of squid to drop
- The height from which the squid fall
- The ability to get rid of the squid cleanly, or set the squid on fire (a new feature!)

Lucky for us, the Canary plugins come with a configuration-file mechanism built in and ready to use. Here's how it works.

The configuration file itself will be named *PluginName*.cfg, and placed in a subdirectory named for the plugin under Server/config. So for our SquidBombConfig plugin, the config file would be here:

```
~/Desktop/server/config/SquidBombConfig/SquidBombConfig.cfg
```

The config file itself looks something like this:

```
numSquids=6
squidDropHeight=5.0
setFire=false
```

The format used in this file is a very simple format. In fact, you've been using it all along—it's the format of the Canary.inf file also.

Each line represents one setting and has just the name of the value you want to store (no spaces in the name, please), followed by an equals sign (=) and then the value. You can store strings, integers, booleans (true or false), doubles, and so on.

If the cfg file doesn't exist when the plugin starts up, it will create a default one based on values you set in the plugin. That's an easy way to get started; once the file is created, you (or a user or server operator) can go in and make changes to the file on disk.

Let's look at how to modify the plugin code to use a config file. The important part is in the enable() function. As it first starts up, it'll go and grab values from the configuration file and set them into a few static variables for our plugin to use. Here's what I added:

SquidBombConfig/src/squidbombconfig/SquidBombConfig.java

```
private static int numSquids;
private static double squidDropHeight;
private static boolean setFire;

// Server/config/SquidBombConfig/SquidBombConfig.cfg:
//    numSquids=6
//    squidDropHeight=5
//    setFire=false

public boolean enable() {
  super.enable();//Compiler will call this if you don't
  logger.info("Getting config data");
  PropertiesFile config = getConfig();
  numSquids = config.getInt("numSquids", 6);
  squidDropHeight = config.getDouble("squidDropHeight", 5.0);
  setFire = config.getBoolean("setFire", false);
  config.save(); // Create a new one if needed
  return true;
}
```

First, we call getConfig() to get the PropertiesFile object for our plugin. (If there is no actual file yet, we'll get an empty object instead.) With this we can now make calls to get values out of the config file.

Each call to read data looks like config.get*Type*(), where *Type* is the type of the variable you're trying to get, so you have getInt, getDouble, getBoolean, getString, and such.

Note that you have to use the camel-case version of the data type, so it's getInt, not getint; getBoolean, not getboolean; getDouble, not getdouble; and so on.

Using functions named get*XXX* and set*XXX* to read and save variables is a pretty standard part of Java. We refer to these kinds of functions as *getters* and *setters*. Okay, perhaps not the most original names.

For each of the config file getters, you can specify a default value to use in case that setting isn't in the file. So for instance, calling:

```
numSquids = config.getInt("numSquids", 6);
```

will set numSquids to 6, even if the setting for numSquids doesn't exist in the config file, or if the file doesn't even exist at all. If there *is* a setting in the file, then numSquids will be set to that value.

At the end of our function we'll call save(). In case the actual configuration file didn't exist yet, this will create it with default values. Otherwise, if we haven't changed any values, it won't do anything.

Later in the plugin, we'll use these static, class-level variables instead of the hard-coded numbers we used in the previous version:

SquidBombConfig/src/squidbombconfig/SquidBombConfig.java

```
double y = loc.getY();
loc.setY(y + squidDropHeight);
me.chat("Spawning " + numSquids + " squid.");
// Spawning some squid. Derp.
for (int i = 0; i < numSquids; i++) {
    spawnEntityLiving(loc, EntityType.SQUID);
}
```

Easy peasy.

Try This Yourself

Run build.sh on your new SquidBombConfig plugin and try the squidbombc command—it should work exactly as it did in the previous version (but this version has a "c" at the end, so it's squidbombc).

Now go to your server's config directory, and you'll see a new SquidBombConfig directory. In that directory you'll find your new SquidBombConfig.cfg file:

```
~/Desktop/server$ cd config
~/Desktop/server/config$ ls
SquidBombConfig       motd.txt              plugin_priorities.cfg worlds
db.cfg                ops.cfg               server.cfg
~/Desktop/server/config$ cd SquidBombConfig/
~/Desktop/server/config/SquidBombConfig$ ls
SquidBombConfig.cfg
~/Desktop/server/config/SquidBombConfig$ cat SquidBombConfig.cfg
numSquids=6
squidDropHeight=5.0
setFire=false
```

Here I used the command cat to dump out the contents of the file.

Edit that file, and change the number of squid to something larger, say 12 or so. Save the file and restart your server.

Now when you run the SquidBomb command, you are inundated with squid.

Congratulations! Your users now have the ability to tweak your plugin without needing access to the source code at all.

All these squid piling up are probably messing up your world a bit, so we need a way to quickly purge the world of all squid. To launch the Great Squid Purge, I added a command squidpurge to this plugin (we'll look at the source in just a second). Give it a try.

You can also change the behavior of plugins, not just change limits and quantities. Go back to the SquidBombConfig.cfg file and change setFire to true. Restart the server, and now when you purge squid they'll be set on fire instead of just dying.

A dozen flaming squid.

Plugin: SquidBombConfig

Let's look at the full code for our final SquidBombConfig plugin.

SquidBombConfig/src/squidbombconfig/SquidBombConfig.java

```
package squidbombconfig;

import java.util.Collection;
import java.util.Iterator;
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 net.canarymod.api.entity.EntityType;
import net.canarymod.api.entity.living.EntityLiving;
import net.canarymod.api.entity.living.animal.Squid;
import com.pragprog.ahmine.ez.EZPlugin;
import net.visualillusionsent.utils.PropertiesFile;

public class SquidBombConfig extends EZPlugin {
  private static int numSquids;
  private static double squidDropHeight;
  private static boolean setFire;

  // Server/config/SquidBombConfig/SquidBombConfig.cfg:
  //    numSquids=6
  //    squidDropHeight=5
  //    setFire=false

  public boolean enable() {
    super.enable();//Compiler will call this if you don't
    logger.info("Getting config data");
```

```
    PropertiesFile config = getConfig();
    numSquids = config.getInt("numSquids", 6);
    squidDropHeight = config.getDouble("squidDropHeight", 5.0);
    setFire = config.getBoolean("setFire", false);
    config.save(); // Create a new one if needed
    return true;
  }

  @Command(aliases = { "squidbombc" },
           description = "Drop a configurable number of squid on your head.",
           permissions = { "" },
           toolTip = "/squidbombc")
  public void squidbombCommand(MessageReceiver caller, String[] args) {
    if (caller instanceof Player) {
      Player me = (Player)caller;
      Location loc = me.getLocation();
      double y = loc.getY();
      loc.setY(y + squidDropHeight);
      me.chat("Spawning " + numSquids + " squid.");
      // Spawning some squid. Derp.
      for (int i = 0; i < numSquids; i++) {
          spawnEntityLiving(loc, EntityType.SQUID);
      }
    }
  }

  @Command(aliases = { "squidpurge" },
           description = "Get rid of squid.",
           permissions = { "" },
           toolTip = "/squidpurge")
  public void squidpurgeCommand(MessageReceiver caller, String[] args) {
    if (caller instanceof Player) {
      Player me = (Player)caller;

      Collection<EntityLiving> squidlist = me.getWorld().getEntityLivingList();
      for (EntityLiving entity : squidlist) {
        if (entity instanceof Squid) {
          Squid victim = (Squid)entity;
          if (setFire) {
            victim.setFireTicks(600);
          } else {
            victim.setHealth(0.0f);
          }
        }
      }
    }
  }
}
```

The arrows show where I've added the squidpurge command. I get a list of all squid in your world with me.getWorld().getEntityLivingList() and then traverse the list with an iterator. For each squid, depending on the setting in the config file, I can then decide whether to set it on fire, like this:

```
victim.setFireTicks(600);
```

or just kill it by setting its health to zero suddenly:

```
victim.setHealth(0.0f);
```

So with just a humble if statement and a config file, you can have your plugin users tune your plugin's behavior without modifying the source code. Sweet.

Store Game Data in a Database

That's great for configuration data, but not so useful for game data—data that changes as the game plays on, like player scores and status, inventory, health, and things like that. We need to access data a little differently, and be able to write game data from the plugin as well as read it.

There is one important limitation to storage approaches, however. You can only save and load simple types: strings, integers, floating point, and boolean values. You can't save Minecraft-specific objects—things like Location or Player. You can save a player's name as a String, and save a Location as a set of double x-, y-, and z-coordinates, but you can't save the Minecraft objects directly.

That turns out to be a reasonable limitation, because you really *should not* store Player objects, worlds, locations, or other "live" game elements.

Keep in mind that the game is still playing in real time. Players log off, they change locations, blocks change types, and things catch on fire, fall, and go in and out of inventory, all while your code is running. If you store a Player object on disk, or even in a list in memory, there's no guarantee that object will still be valid when you go to use it the next day or next week. In fact, odds are it probably *won't* be valid. All a player needs to do is disconnect and reconnect, and *bam*—the old Player object is no longer valid.

So instead of storing a Player object, you store the player's name. When you load the data back in and are ready to use it, you'd check to make sure the player is actually online.

Now there are two ways to go about saving and loading game data using Canary. One is to use a full-fledged SQL database like SQLite or MySQL, using the SQL language to search for and store data. But that's a lot more complicated than we have space to cover here. If you're dealing with thousands

of users on a large server, and complicated sets of nested data, you may need to look into the SQL techniques. There's an official tutorial online.[1] But it's a much more advanced and potentially confusing topic, even for professionals.

On the other hand, if your data needs are relatively flat and simple, and you're only dealing with hundreds of users, then Canary has a nice option for you: DataAccess objects and the Database front end.

When dealing with databases, the words we use to describe things are a little different from the words we use in code. In code, you might have a class or a record, made up of variables, or fields. Databases have these same things, but we call fields *columns.* A bunch of columns are grouped together in a *table.*

And since searching for records is a big part of databases, we have a new word for the thing we might search for the most: a table's *primary key* field. A primary key means there's only one record in that table with that value. For instance, player_name might be a good primary key, because every user has a unique name, so it's an easy way to find an individual player.

DataAccess objects

In the Canary API, you can create a DataAccess object that defines what you want to store in a database. For example, suppose you want to store the location of a player. You'd want to be able to save and look up that record by the player's name, and maybe save the x-, y-, and z-coordinates.

To do that, you'd start with a separate class, derived from DataAccess, that just has the data you want to load and save:

```
public class AllPlayerLocations extends DataAccess {
  public String player_name;
  public double x;
  public double y;
  public double z;
}
```

That's the data itself, but Canary needs a few more things before you can use this with a database.

First off, it needs to know what to call this field in the database, and what type of data it represents. For each of x, y, and z, we'll name the database field the same thing as the variable. Each of these is a Java double. You'd specify these details using the @Column annotation, like this:

1. http://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html

```
@Column(columnName = "x",
        dataType = DataType.DOUBLE)
public double x;
```

Now the system will know how to store each coordinate. We can do almost the same thing for the player's name:

```
@Column(columnName = "player_name",
        dataType = DataType.STRING)
public String player_name;
```

this time, specifying that it's a String type. But we need to add a little something extra for the player_name. We want this field in the database to be unique: the *primary key*. There should only be one record in the database with an x, y, and z for any one player name. You specify that as the columnType, like this:

```
@Column(columnName = "player_name",
        columnType = Column.ColumnType.PRIMARY,
        dataType = DataType.STRING)
public String player_name;
```

Now that the columns are specified, you need to name the database table that will hold these records. You do that right in the constructor:

```
public AllPlayerLocations() {
  super("all_player_locations");
}
```

That tells the database that you want to save these fields in a database table named all_player_locations.

Finally, you need to make this class extend the DataAccess parent class, and add a method named getInstance that returns an object of this class. So all together, the final DataAccess class looks like this:

LocationSnapshot/src/locationsnapshot/AllPlayerLocations.java

```
package locationsnapshot;

import net.canarymod.database.Column;
import net.canarymod.database.Column.DataType;
import net.canarymod.database.DataAccess;

public class AllPlayerLocations extends DataAccess {
  @Column(columnName = "player_name",
          columnType = Column.ColumnType.PRIMARY,
          dataType   = DataType.STRING)
  public String player_name;

  @Column(columnName = "x", dataType = DataType.DOUBLE)
  public double x;
```

```
  @Column(columnName = "y", dataType = DataType.DOUBLE)
  public double y;

  @Column(columnName = "z", dataType = DataType.DOUBLE)
  public double z;

  public AllPlayerLocations() {
    super("all_player_locations");
  }

  public DataAccess getInstance() {
    return new AllPlayerLocations();
  }
}
```

Let's see how to use this code within a plugin.

Plugin: LocationSnapshot

Here's an example of using a DataAccess (in this case, AllPlayerLocations) in a new plugin, LocationSnapshot. For this plugin, we'll provide two new commands:

- /savelocations
- /loadlocations

You might want to use this kind of feature as part of a competition, where you can return all players back to their starting points at the end.

The savelocations command, as you'd expect, saves the current locations of all online players to disk. loadlocations reads them back in and teleports everyone back to those saved locations. We'll use Canary's Database functions with our AllPlayerLocations DataAccess object to save and load a hash of Players and Locations.

There are two main pieces to this plugin: saveLocations and loadLocations. To keep things clean and start forming better habits, we'll put the code for each in its own function.

savelocations

The logic for savelocations will look like this:

LocationSnapshot/src/locationsnapshot/LocationSnapshot.java

```
private void saveLocations() {
  List<Player> playerList = Canary.getServer().getPlayerList();
  // For all players...
  for (Player player : playerList) {
    // Save the raw coordinates, not the Location
    AllPlayerLocations apl = new AllPlayerLocations();
    apl.player_name = player.getDisplayName();
    Location loc = player.getLocation();
```

```
      apl.x = loc.getX();
      apl.y = loc.getY();
      apl.z = loc.getZ();

(1)   HashMap<String, Object> search = new HashMap<String, Object>();
      search.put("player_name", player.getDisplayName());

      try {
(2)     Database.get().update(apl, search);
      } catch (DatabaseWriteException e) {
        logger.error(e);
        logger.info("error");
      }
    }
}
```

Here we run through the list of all online players. For each, we stick their name and x-, y-, and z-coordinates into a new AllPlayerLocations object. That will get saved into the database with the call to update() down at ❷.

How does the database know what to update? You have to pass it a HashMap that specifies the record you want to update (or create, if this is the first time). So at ❶, we'll build a quick hash that tells the database "update the record where the field named player_name is equal to this Player's getDisplayName()."

Now we can call the database's update function, which looks like this:

```
try {
  Database.get().update(apl, search);
} catch (DatabaseWriteException e) {
  logger.error(e);
  logger.info("error");
}
```

Now there's something we haven't seen before; what does this extra, mysterious code do?

Catch Exceptions in Java

See the code block bracketed by the try/catch keywords? The database functions are declared to "throw exceptions" when something goes wrong (maybe there was an error writing the file because the disk is full). What's an exception?

An *exception* interrupts the code that's currently being run, throws away the rest of the function, and jumps straight to the catch block. It's sort of like if the fire alarm rang right now, you'd jump up and run out, and not finish reading the page.

If there's no error, then the catch block will never be run.

To simplify things, our code catches the `DatabaseWriteException` and just logs the error—basically just ignoring it. That can be a very "beginner's" way to handle exceptions, though, and you shouldn't always just ignore them. Instead, the question to ask yourself is "whose problem is it, and who can fix it?"

Normally, code that you're calling in a library has no idea what you're using it for. When something bad happens, it doesn't know what you need to do to handle the problem. Should the whole server exit? Should you return an error to the user? Do you need to change some variables and clean up whatever you were trying to do? Since the library code doesn't know, it doesn't try to fix it. It just raises an exception—it throws up its hands and gives up.

When dealing with exceptions, you have two choices. You can either accept the responsibility for dealing with any errors and catch and handle the exception yourself, or you can throw up your hands as well, and pass the exception up to the caller. To pass the exception up, you just declare that *your* function also `throws DatabaseWriteException` (or whatever the actual exception type is). You don't need any `try/catch` at all; now it's someone else's problem.

In theory, however, exceptions should be used only for exceptional things. If you went to spawn a cow and instead got a creeper, that would be exceptional: an unexpected disaster. If you just added a player to a list of players and the length of the list was still zero, that would indicate a disaster in progress.

But trying to search for something that isn't in the database, for example, isn't a disaster. It's a common occurrence. So in that case you might want to catch the exception and ignore it.

And what happens if you fail when trying to write data to the database, as we're doing here? That might mean there's a more serious problem—you might be out of disk space on the server, for instance. What should you do?

If it were me, I'd want to shut down the server as gracefully as possible. A dead program does a lot less damage than a broken program that's still running. When writing code for most professional systems, you'll find that's almost always the best action to take: fail quickly and loudly.

But this isn't our server necessarily; we've just made a plugin that someone else is running, and it would be rude of us to shut down their server just because we can't write a file. So in this case, we'll log an error message to the Minecraft console and struggle on.

loadlocations

Sometime later, when you type the loadlocations command, you'll run through the list of players who are online and look up their saved location from the database; convert the x-, y-, and z-coordinates back to a proper Location; and teleport them there:

LocationSnapshot/src/locationsnapshot/LocationSnapshot.java

```
private void loadLocations() {
  //Go through list of players; if they are in the hash, teleport them.
  List<Player> playerList = Canary.getServer().getPlayerList();
  for (Player player : playerList) {
    String name = player.getDisplayName();

    AllPlayerLocations apl = new AllPlayerLocations();
    HashMap<String, Object> search = new HashMap<String, Object>();
    search.put("player_name", name);

    try {
      Database.get().load(apl, search);
    } catch (DatabaseReadException e) {
      logger.info(name + " is not online");
      continue;
    }

    // Reconstitute a Location from coordinates
    Location loc = new Location(apl.x, apl.y, apl.z);
    logger.info("Teleporting " + name + " to " + printLoc(loc));
    player.teleportTo(loc);
  }
}
```

The "save" obviously needs to be done by going through the list of all players online, and it's easiest to do the same thing for the load as well. We *could* start with a list of all the players we saved in the database, and then check to see if they are online, but it's probably easier this way. In either case, there may have been players who were online before and aren't now, or vice versa.

So all we have to do is find this player in the database, using a search map again, and the load function. When we are ready to teleport the player, we make a new Location object based on the coordinates we saved and off they go.

Compile and install LocationSnapshot and give it a try. Connect from your client and run the /savelocations command.

Exit your client, stop the server, and restart everything. Go somewhere else in the game and try the /loadlocations command. You're back at the snapshot location (as is every other player on your server).

XML or SQLite

In the server configuration file, server/config/server.cfg, you can choose the back end that the Database will use for loading and saving. By default, it uses an XML format file, which will be saved in the server/db directory.

You can change that to use any of xml, mysql, or sqlite by editing this line in the server config file:

```
data-source=xml
```

For better performance, you can just change that option to sqlite without having to do any additional configuration. To use the even higher performance mysql option, you'd need to set it up using server/config/db.cfg and run a MySQL server.

Plugin: BackCmd with Save

Using the same ideas as with LocationSnapshot, you're going to add data storage to the BackCmd plugin so that you can save the locations across server restarts. I'll provide the outline and function signatures, but you will be writing the bodies of the functions, the actual code to make it work. Buckle up!

Now, the BackCmd plugin is a little more complicated than our simple LocationSnapshot. As you may remember from the plugin on page 128, the BackCmd plugin tracks more than just a location per player; it tracks a Stack of Location objects for each player.

Bear in mind that the Database system can't actually store Location objects. It can store a list, which we can use as a stack, but it can only store a list of a single basic type. It cannot store a list of separate x-, y-, and z-coordinates, or a list of arrays. So we're going to cheat a bit: we'll store the coordinates in a string when we write, and pull them back out from a string into separate doubles when we read.

Here's your to-do list for the code you're going to write:

- Create a SavedLocation, which will be a DataAccess object.
- The main plugin deals with players, and will want to just add a new Location and retrieve the last location for a given player. So it makes sense to make the SavedLocation look like a stack, with a pop and push method.
- Because SavedLocation will just look like a stack to the plugin, it will do the database load and save functions internally (as private functions). You'll need to write the private functions in SavedLocation that call the Database and check for any exceptions.

- Write a conversion function that takes a Location object and returns a string that contains each coordinate, separated by "," characters.
- Write a conversion function that takes a string of coordinates, separated by "," characters, and builds a Location object.

Now that's just a first pass at what you might need to do. As we go along, other things might come up, and that's okay: that's how software development really works.

And as you go along, don't be shy about adding logger.info() messages to help trace what the plugin is doing.

Okay, you have your hands full. I'll get you started.

Create a SavedLocation Class

Since you can't save a stack of Location objects directly to disk, you need to make an object that you *can* save to disk: a DataAccess object. You'll create a SavedLocation that will save a list of strings for each player.

Let's get started writing some code. Go into the BackCmd plugin's src/backcmd directory and create a new file named SavedLocation.java. The first thing you'll need is a package statement and some imports.

The start of your SavedLocation class will look like this:

BackCmdSave/src/backcmdsave/SavedLocation.java

```
package backcmdsave;

import java.util.ArrayList;
import java.util.HashMap;
import net.canarymod.database.Column;
import net.canarymod.database.Column.DataType;
import net.canarymod.database.DataAccess;
import net.canarymod.database.exceptions.*;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.World;
import net.canarymod.database.Database;
import net.canarymod.Canary;

public class SavedLocation extends DataAccess {
```

Copy that into your new file, or type it in as we go.

Next up you'll need to add the @Column annotations to describe the fields you want to save in the database. Remember, you'll need two fields:

- player_name — a string, and a primary key, just like we used in the LocationSnapshot plugin.
- location_strings — a list of strings. To make a list for a field, add the parameter isList = true to the annotation, and make the variable a list of strings: ArrayList<String> instead of just a plain String.

Give that a try now. You should end up with something that looks like this:

BackCmdSave/src/backcmdsave/SavedLocation.java

```
@Column(columnName = "player_name",
  columnType = Column.ColumnType.PRIMARY,
  dataType = DataType.STRING)
public String player_name;

@Column(columnName = "location_strings",
    dataType = DataType.STRING,
    isList = true)
public ArrayList<String> location_strings;
```

You need two more things before this piece of code will even compile, just like we did up in the SavedLocation plugin:

- Add a default constructor that supplies a name for this table in the database (let's call it "saved_player_locations")
- Add a function named getInstance that returns a new one of these as a DataAccess object.
- Add one extra thing: a constructor that takes a string for the player name. In the body of the constructor, you'll need to call super just like in the default constructor, then make the assignment to player_name.

When you're all done, it should like something like this:

BackCmdSave/src/backcmdsave/SavedLocation.java

```
public SavedLocation() {
  super("saved_player_locations");
}

public SavedLocation(String name) {
  super("saved_player_locations");
  player_name = name;
}

public DataAccess getInstance() {
  return new SavedLocation();
}
```

Make sure that much compiles, using build.sh as usual.

Read From and Write To the Database

As far as the database knows, we're dealing with only two fields, player_name and location_strings, which use the player_name as a key.

Looking back at LocationSnapshot, you can copy the database's load and update functions, with the exception handling, and make two new functions here:

```
private void myRead(final String name) {
}

private void myWrite() {
}
```

In each function, set up the search HashMap, then do the Database's load or update in a try-catch block.

Go try that now. When you're done, you should have something like this:

BackCmdSave/src/backcmdsave/SavedLocation.java

```
private void myRead(final String name) {
  player_name = name;

  HashMap<String, Object> search = new HashMap<String, Object>();
  search.put("player_name", name);

  try {
    Database.get().load(this, search);
  } catch (DatabaseReadException e) {
    // Not necessarily an error, could be first one
  }
  if (location_strings == null) {
    player_name = name;
    location_strings = new ArrayList<String>();
  }
}

private void myWrite() {
  HashMap<String, Object> search = new HashMap<String, Object>();
  search.put("player_name", player_name);

  try {
    Database.get().update(this, search);
  } catch (DatabaseWriteException e) {
    //Error, couldn't write!
    System.err.println("Update failed");
  }

}
```

Make sure that much compiles, using build.sh as usual.

Now you've got a DataAccess class that will update and load from the database. But there's no good way to connect it to the plugin yet, so we need to take a look at that.

Convert Location to String and Back

Before we get much further, we may as well jump in and tackle the conversion functions for a Location, as we'll need those before we start working on the stack bits.

Converting a Location to a string is easy: just build a string from each of the getX(), getY(), and getZ() results using a non-numeric character to separate each number. You could use spaces, or commas, but I prefer commas (",") as that seems easier to read.

Write a function named locationToString that takes a Location and returns a String.

But now how to go the other way? Given a string that looks like "110,75,220", how can you split it up into three parts, one for each number?

Java to the rescue! In the Java doc under String, you'll find a function named "split", which takes a string and splits it up into an array. So if str="110,75,220", then str.split(","); would return an array of three strings, "110", "75", and "220". Then it's just a matter of using Double.parseDouble(string) to convert each string into a double. With the doubles, you can create a new Location.

Now you can write a function named stringToLocation that takes a String and returns a Location.

When you're done, it might resemble this:

BackCmdSave/src/backcmdsave/SavedLocation.java

```
private String locationToString(Location loc) {
  return loc.getX() + "," +
          loc.getY() + "," +
          loc.getZ();
}

private Location stringToLocation(String str) {
  String[] arr = str.split(",");
  double x = Double.parseDouble(arr[0]);
  double y = Double.parseDouble(arr[1]);
  double z = Double.parseDouble(arr[2]);
  return new Location(x, y, z);
}
```

Make sure that much compiles, using build.sh as usual.

Act Like a Stack

Even though this is a DataAccess object, we are free to add other functions and bits of data to it to help make building our plugin easier. So from the perspective of the main plugin, we'd like to be able to treat a SavedLocation for a given player as a stack of Locations. That means we'll need to expose at least a public push and pop function.

Here's the first hitch: location_strings is declared as an ArrayList<String>, not a Stack. So how would you implement push and pop using just a plain old list?

Well, the push is easy: an array add method will add a new item to the end of the list. So you just use that for the push.

In order to pop the stack, you have to remove the last item in the list using its index, the size() of the list minus one.

Okay, that sounds reasonable. In the public push functions, you'll need to read this player's stack from the database; call an internal, private function to actually push that location onto the stack; then save it back to the database. pop will be similar: read from the database, pop the stack, and write the new stack back to the database. If you remember back in the BackCmd plugin, it wasn't just enough to pop the latest value—we had to use an equalsIsh function to make sure we'd moved far enough away from the teleport location.

Putting it all together, the public functions in SavedLocation would look like this:

BackCmdSave/src/backcmdsave/SavedLocation.java

```
public void push(Location loc) {
  myRead(player_name);
  //Make sure previous location is different if it exists
  if (peek_stack() == null ||
    !equalsIsh(peek_stack(), loc)) {
    push_stack(loc);
    myWrite();
  }
}

public Location pop(Location here) {
  myRead(player_name);
  if (location_strings.size() == 0) {
    return null;
  }
  Location loc = pop_stack();

  myWrite();
  return loc;
}
```

There are a few things you still need to write: the internal push_stack, pop_stack, and peek_stack functions, and the database functions, myRead and myWrite. You can copy the equalsIsh function straight from the BackCmd plugin.

Let's get started on the internal stack functions first. We want to be able to pass in a Location to the push, and get a Location back from the peek and pop, so use the conversion functions locationToString and stringToLocation.

Go give that a try on your own first. When you're done, come check back here and see if it looks close to this:

BackCmdSave/src/backcmdsave/SavedLocation.java

```
private void push_stack(Location loc) {
  String s = locationToString(loc);
  location_strings.add(s);
}

private Location peek_stack() {
  if (location_strings.isEmpty()) {
    return null;
  }
  String s = location_strings.get(location_strings.size()-1);
  return stringToLocation(s);
}

private Location pop_stack() {
  Location loc = peek_stack();
  location_strings.remove(location_strings.size()-1);
  return loc;
}
```

Make sure that much compiles, using build.sh as usual.

Add Save and Load to BackCmd

Now that you have a SavedLocation that knows how to push and pop individual locations for a Player, it's time for you to add the code into BackCmd.java, which will call them.

Open up BackCmd.java and start making the following changes:

- In onTeleport, delete all the code in the else portion of the block, and replace it with code that creates a new SavedLocation and calls its push with the player's location.
- In backCommand, get the location to teleport to by creating a new SavedLocation and calling its pop to get the latest value.

The new plugin main class should end up looking like this:

BackCmdSave/src/backcmdsave/BackCmdSave.java

```
package backcmdsave;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Stack;
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.hook.HookHandler;
import net.canarymod.hook.player.TeleportHook;
import net.canarymod.plugin.PluginListener;
import com.pragprog.ahmine.ez.EZPlugin;


public class BackCmdSave extends EZPlugin implements PluginListener {
  private static List<Player> isTeleporting = new ArrayList<Player>();

  @Override
  public boolean enable() {
    Canary.hooks().registerListener(this, this);
    return super.enable(); // Call parent class's version too.
  }

  @HookHandler
  public void onTeleport(TeleportHook event) {
    Player player = event.getPlayer();
    if (isTeleporting.contains(player)) {
      isTeleporting.remove(player);
    } else {
      SavedLocation sp = new SavedLocation(player.getName());
      sp.push(player.getLocation());
    }
  }

  @Command(aliases = { "dback" },
           description = "Go back to previous places that you teleported to.",
           permissions = { "" },
           toolTip = "/dback")
  public void backCommand(MessageReceiver caller, String[] args) {
    if (caller instanceof Player) {
      Player me = (Player)caller;

      SavedLocation sp = new SavedLocation(me.getName());
      Location loc = sp.pop(me.getLocation());
```

```
      if (loc != null) {
        isTeleporting.add(me);
        me.teleportTo(loc);
      } else {
        me.chat("You have not teleported yet.");
      }

    }
  }
}
```

(I renamed the command to be "dback" so it would be different from the existing `BackCmd`. You're welcome to rename it also, or just use it instead of the old "back.")

Now compile and install the whole mess with `build.sh`, and you're good to go.

Test It

Assuming everything worked, you should be able to test it:

1. Start the server.
2. Connect from your client.
3. Teleport to a new place, either by using the `/tp` command or by throwing an Ender Pearl. Do this a few times.
4. Disconnect and shut down the server.
5. Restart and reconnect.
6. Now type the `/back` command in the client. Your history was saved and reloaded from disk, and you should be able to go back to each teleport point.

w00t.

Try This Yourself

You know what we need? A "clearback" command so that we can clear out our teleport history (Canary already provides a "clear" command, so you need to call it something other than that).

Go ahead and implement that. It's a new command, so you'll need a new `@Command` section and function. In the code, all you need to do is to clear out the stack for the player and save it back to the database.

Next Up

That was fun! And you're almost done. This is the last batch of Java techniques you needed. Now that you're able to save and load data from the database, use Java file functions, and handle Java exceptions, you're in good shape.

In the next chapter we'll switch things up a bit, and instead of writing code, we'll look at a technique to help manage the code you write. You're going to set up a giant "undo button" for your plugin project code.

After that, you'll be ready to wrap up with a walk-through of how to design your very own plugin from scratch.

Your Growing Toolbox

You now know how to:

- Use the command-line shell
- Build with Java, `javac`
- Run a Minecraft server
- Deploy a plugin
- Connect to a local server
- Use Java variables for numbers and strings
- Use Java functions
- Use `if`, `for`, and `while` statements
- Use Java objects
- Use imports for Java packages
- Use `new` to create objects
- Add a new command to a plugin
- Work with `Location` objects
- Find blocks/entities
- Use local variables
- Use class-level global variables
- Use `ArrayLists`
- Use `HashMaps`
- Use `private` and `public` to control visibility
- Modify Minecraft blocks
- Modify and spawn entities
- Listen for and react to game events
- Manage plugin permissions
- Create a separate class
- Schedule a task to run later
- Schedule a task to run periodically
- Save and load configuration data
- Build up complex code from simple functions
- Save and load plugin game data
- Use `DataAccess` to use the database
- Catch and throw Java exceptions

In this chapter we'll add Git to your arsenal. With Git you can change your code without fear that you'll lose anything or mess anything up. You'll add these skills to your toolbox:

- How to install Git
- Use Git to keep track of changes to code
- Go back to earlier versions of code (an "undo button")
- Maintain multiple versions of code at the same time
- Back up your code to the cloud

Git gives you one of the most powerful tools of all: an "undo button" for your project.

CHAPTER 12

Keep Your Code Safe

By now you might recognize a certain frustrating moment when coding: everything was working just fine, but you added a couple of lines of code in a couple of different files, and now *nothing works anymore*. What changes did you make? Which one messed everything up? Was it in this file or that one?

Wouldn't it be great to be able to press a sort of giant "undo button," to toss out this set of changes that didn't work out so well, and go back to the code from a few minutes ago that was working all right so you can try again?

Have I got a treat for you.

You can use a tool named Git to do exactly that and more. Git acts like a huge memory of all the changes you make to your code. You can go back in time to any previous point, even if you've accidentally deleted a file or renamed it, moved code from one file to another, and so on—Git will track it all for you. In addition, you can set it up so that a copy of Git's memory can be backed up in the cloud, so even if your whole computer gets damaged or stolen, all of your source code and the full memory of it are safe. You can restore it to your new computer or a friend's computer and continue on.

You don't *have* to use this kind of tool, but I really, really recommend it. It's not hard to set up, and the first time it saves you from stupidly clobbering a file it will be worth it.

There are tons of tutorials and how-tos for Git on the Web, and a lot of books as well, but we'll take a quick pass at the basics here to get you started.

Install Git

First off, you need to download and install the Git software for Windows, Mac, or Linux from http://git-scm.com.

Once it's installed, you should configure it globally with your name and email address:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "you@example.com"
```

Git on Windows

The Git distribution for Windows includes a well-intentioned but problematic version of the Bash shell. Readers tell me it doesn't work very well. In the Git installer for Windows, you'll want to choose "Run Git from Windows command prompt" instead of selecting the default "Git bash" option.

Also, you may run into complaints about line-ending differences. Windows uses the CR LF characters to mark the end of lines, and other systems just use LF. Java is fine with either convention, and so are most editors. But git may warn you that it's trying to convert line endings; don't worry about it.

You set up Git once per project. Let's do that right now, in the CowShooter plugin. cd to the top-level CowShooter directory and type git init:

```
$ cd CowShooter/
$ ls
Canary.inf  Manifest.txt  bin  build.sh  dist  src
$ git init
Initialized empty Git repository in /Users/andy/Desktop/code/CowShooter/.git/
```

That creates a magical, hidden directory named .git where Git will store its memories—what Git calls your *repository*. You need to do this only once for each project, which in our case will be once per plugin.

Remember Changes

Now that you have Git, you need to tell it which of your files it needs to remember. You don't actually want all of them.

Typically you don't want to store .class or .jar files, as those can always be created again. You definitely want Git to remember all your .java source code, and your plugin-configuration file and build script.

So first thing, let's clean up the plugin directory and get rid of the extra junk:

```
$ rm -r bin
$ rm -r dist
```

That removed the generated .class and .jar files. Now you're left with a clean directory tree, with the files you want Git to remember and track changes for.

You use git add *files...*, where *files* can be individual file or directory names. Since for now you want everything in the current directory from here on down, you can just type git add .:

```
$ git add .
```

(Remember that "." means the current directory.) Now Git knows to watch all these files.

But it hasn't yet taken a snapshot of the current state of your files. To have it do that, you have to *commit* your changes:

```
$ git commit -a -m 'My first commit'
```

That command will record the current state of your source code in Git's repository. The -a means to commit changes in *all* the tracked files, and the -m lets you specify a *message*.

The message is very important: it's how you can tell what you were doing when you changed these files. If you use a wonderfully descriptive message like "I did stuff," then this won't help you any.

Let's play with that a moment. Create a new file in the CowShooter directory and name it README.txt. Put anything you want in the file: comments about the plugin, your own personal manifesto, nonsense text, whatever.

Now add the file and commit the add:

```
$ git add README.txt
$ git commit -a -m "Add my personal screed"
[master 50847c8] Add my personal screed
 1 file changed, 1 insertion(+)
 create mode 100644 README.txt
```

To see the history of your commit messages, you use git log:

```
$ git log
commit 50847c8a3e60dbfc8f441894202765eb23fbd9a5
Author: Andy Hunt <andy@toolshed.com>
Date:   Fri Jan 7 15:51:22

    Add my personal screed

commit aa58dfa87fa79b01e2c7ce172bbcdd41b7e962d6
Author: Andy Hunt <andy@toolshed.com>
Date:   Tue Jan 7 15:50:31

    First Commit
```

That shows the full log entries, with the long version of the commit identifier, the author, the date, and the log message. Those long, random-looking strings (which are cryptographically interesting SHA1 hashes, in fact) identify each commit. But they are really long. Usually you can just use the first four to eight characters of the hash as long as that produces a unique string.

You can get a more concise report of the same information by using this:

```
$ git log --oneline
50847c8 Add my personal screed
aa58dfa First Commit
```

The shorter version of the commit hash is usually all you need.

So remember: when you create a new file and need Git to track it, don't forget to do the add:

```
$ git add myNewFile.java
```

And then do a commit to save a snapshot of all your files in the repository. You can always check to see what files Git knows about, which ones it doesn't, what's been committed, and what hasn't, by typing this:

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#       bin/
#       dist/
nothing added to commit but untracked files present (use "git add" to track)
```

Oops! I must have done a compile in there somewhere, because the `bin` and `dist` directories are back. That's fine, as we don't want Git to track them, but we also don't want Git to complain about them every time.

If these were files you cared about, you would just do the `git add` so that Git can track them. But in this case, we don't want Git to ever track these directories. We want it to ignore them. Here's how to make that happen:

1. Create a file named `.gitignore` in this directory (if there isn't one already; `mkplugin.sh` will make an empty one for you).
2. In the file, put the names of the files or directories you want to ignore.
3. Add the file and commit it.

In this case the `.gitignore` file will look like this:

```
bin
dist
```

Then we'll add it and commit just that one file by specifying the file name instead of the -a we usually use:

```
$ git add .gitignore
$ git commit .gitignore -m 'ignoring bin and dist'
[master cc0e424] ignoring bin and dist
 1 file changed, 2 insertions(+)
 create mode 100644 .gitignore
```

Let's see what git status tells us now:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

Great! We got the reassuring message "nothing to commit, working directory clean." That message is what you want to see when you're finished coding for the time being.

Now add some text to the bottom of README.txt, save the file, and try git status again:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   README.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Ah, that's clever: Git recognized that you'd made some changes to the README.txt file and that those changes haven't been saved yet. A commit will fix that, and once again you can get the relaxing "nothing to commit, working directory clean" message:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

So there's one important rule to remember:

Don't leave the scene until the working directory is clean.

An Easy Undo

Say you're going along and accidentally remove a file from your plugin directory. Or maybe you made a series of edits that turns out to be a really bad idea, and you want to go back. Oops. Have no fear: we've all done it.

In fact, let's do it now, deliberately—and see how to fix it, using the CowShooter plugin where you've set up a Git repository.

Bring up CowShooter/src/cowshooter/CowTask.java, and somewhere randomly in the middle of the file, type some nonsense like "Zombie cows are coming! Run for your life! Braaaainsss...." That will do for our example of making a mistake in a file. Java, of course, is not hip to our warning of the impending zombie-cow apocalypse, and has no idea what this means. Try to compile with build.sh, and you'll see a plethora of angry error messages like this:

```
src/cowshooter/CowTask.java:20: illegal start of type
    Zombie cows are coming! Run for your life! Braaaainsss....
                                                          ^
src/cowshooter/CowTask.java:20: <identifier> expected
    Zombie cows are coming! Run for your life! Braaaainsss....
                                                           ^
src/cowshooter/CowTask.java:22: ';' expected
    public void run() {
          ^
src/cowshooter/CowTask.java:22: invalid method declaration;
return type required
    public void run() {
```

Argh! Something we typed blew up the compile. Now, if this were a real emergency you wouldn't know it was the bogus zombie warning text you just typed in.

So there's an interesting question: what has changed locally in your files that is different from the last time you did a commit (that is, took a snapshot)?

You can see what's changed by using git diff:

```
$ cd src
$ cd cowshooter
$ git diff CowTask.java
diff --git a/src/cowshooter/CowTask.java b/src/cowshooter/CowTask.java
index 4894201..556d0d3 100644
--- a/src/cowshooter/CowTask.java
+++ b/src/cowshooter/CowTask.java
@@ -16,6 +16,8 @@ public class CowTask extends ServerTask {
         cow = myCow;
     }
+
+Zombie cows are coming! Run for your life! Braaaainsss....

     public void run() {
       if (cow.isOnGround()) {
```

The output has some gunk at the top and then a few lines from the middle of CowTask.java, including two marked with plus signs (+). Those are the two lines that were added. Ah, that's the problem! Adding those lines was a bad idea; in fact, it would be nice just to scrap everything since the last commit, and restore this file to how it was before.

If you haven't committed a file yet, you can always get back to the last commit (like a save point in a game) by typing this:

```
$ git checkout MyMessedUpFile.java
```

Silently but surely, MyMessedUpFile.java goes back to the way it was. Anything you typed in since the last commit is gone. Vanished.

So in our case, we can do this:

```
$ cd src/cowshooter
$ ls
CowShooter.java CowTask.java
$ git checkout CowTask.java
```

Now CowTask.java is back to a known, running state. We can even recompile:

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

So, it's easy to throw out bad changes and get back to the last save point. But what if you committed bad changes a while back, and only noticed the problem now? You can use the same checkout command, but this time specify which commit to fetch.

Let's try that now. Go back into CowTask.java and add the bad zombies line again. But now commit it with this:

```
$ git commit -a -m 'Added zombie warning'
[master e4ee198] Added zombie warning
 1 file changed, 2 insertions(+)
```

Now let's break it even worse. Find the line that says

```
private Cow cow;
```

and delete it. Commit that as well:

```
$ git commit -a -m 'Deleted variable declaration'
[master 44b39f3] Deleted variable declaration
 1 file changed, 3 deletions(-)
```

Now let's see what the log says for CowTask.java (in the src/cowshooter/ directory):

```
$ cd ~/Desktop/code/CowShooter/src/cowshooter
$ git log --oneline CowTask.java

44b39f3 Deleted variable declaration
e4ee198 Added zombie warning
aa58dfa First Commit
```

Your commit IDs, those magic numbers that are listed on each line, will be different from mine. But let's use mine for this example.

So in this case, we want to go back to the last known good version of this file. That means we want to skip back past commits 44b39f3 and e4ee198, and check out this file at my commit ID aa58dfa, when life was happy:

```
$ git checkout aa58dfa CowTask.java
```

(Remember, your commit ID will be different.)

And silently but surely, CowTask.java reverts to its previous state, back before you added the zombie text and before you deleted those variables.

Now the file has actually been changed, just as if you'd edited it by hand, so be sure to commit this latest change with an appropriate message (perhaps something like "That was a bad idea; back to the drawing board."):

```
$ git commit -a -m 'That was a bad idea'
[master 0b6b051] That was a bad idea
 1 file changed, 3 insertions(+), 2 deletions(-)
```

Our bad code hasn't been forgotten; just like a bad day at school or a rotten quiz grade, it's part of history, as git log reveals:

```
$ git log --oneline CowTask.java
0b6b051 That was a bad idea
44b39f3 Deleted variable declaration
e4ee198 Added zombie warning
aa58dfa First Commit
```

And if you really wanted to, you could even go back to the bad version of the code at commit 44b39f3 or commit e4ee198. That's part of Git's beauty: it remembers everything, good and bad, and you can always go back in time.

You can use this same idea for multiple files that were involved in the same commit—by specifying all the files you need instead of just the one. You can even do it across the entire project—just don't specify any file to git checkout, and it will operate on the whole project repository.

Visit Multiple Realities

Being able to revert to a previous commit is great; it's like time travel. You can always revisit your past. But why stop at just one past?

Git has a really neat feature called "branches." These aren't like branches on a tree, but more like branches in the space-time continuum: they are alternate realities, or alternate timelines like in science-fiction stories.

Here's how you might use branches: Suppose you have everything working, and maybe you've even released your plugin to the world. You want to experiment with a new feature or two, but you need the current version of your plugin around as well, in case there are any fixes you need to make for your users. You need two different timelines: one where the plugin stays as it is, maybe with a few fixes, and one where it's growing and gaining new features. Maybe you want another timeline as well, where you try implementing your features in a totally different way.

Not a problem!

You can create a new timeline easily using Git's `branch` command. `branch` by itself will list all the current branches in your project:

```
$ git branch
* master
```

There's only one branch by default. It's named `master`, and you're in that timeline. Now let's split the universe into two and make an alternate reality! It's simple (don't type this in yet, we'll get to that in a second):

```
$ git branch cow-plane
```

And now you've created a new timeline. But you're not in it yet; you're still in `master`. (Type `git branch` and you'll see the star is still next to `master`.) To switch into the other timeline, type this:

```
$ git checkout cow-plane
```

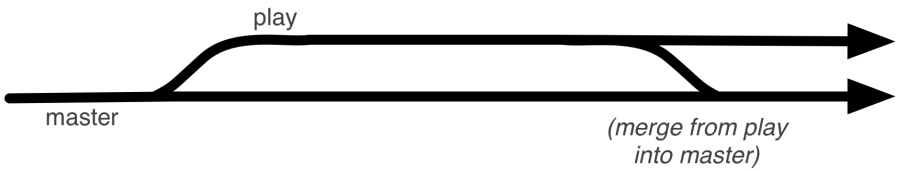
Yes, it's our good friend `git checkout` again. Powerful magic. It's now transported you to a different reality. Nothing you do here will affect the `master` timeline. You can edit files, delete files, add new files, commit changes—everything. And it's all in the cow-plane universe. You can go back to `master` at any time:

```
$ git checkout master
```

Now you'll see the world as it was in master's time, with none of your changes from cow-plane. You fix problems and release this version of the plugin, without any of the work-in-progress bits from your cow-plane timeline.

Try This Yourself

Let's try that for real in CowShooter. We're going to make a new branch, called play, make some changes in both the master and play branches, and merge changes from play back into master. Picture it like this:



The master time stream is going along, minding its own business. We'll create a new branch to play on, and call it play:

```
$ cd ~/Desktop/code/CowShooter
$ git branch play
```

Then switch to that branch using git checkout:

```
$ git checkout play
Switched to branch 'play'
```

When in doubt as to where you are, just type git branch with no arguments, and it will tell you what branches exist and mark the one you're on with an asterisk (*):

```
$ git branch
  master
* play
```

So there are now two branches, and we're on play. Right now the content of the two branches is identical. Let's fix that.

Edit CowShooter.java. Near the bottom of the if statement (that checks to see if the player is holding leather), there's a call to our helper function, fling:

```
...
Canary.getServer().addSynchronousTask(new CowTask(victim));

fling(player, victim, 3);
victim.setFireTicks(600);
...
```

Let's make our own version. Create a new function in this plugin and name it myFling. Copy the version from EZPlugin into the body of the function, and it will look like this:

```
public static void myFling(LivingBase player,
                           LivingBase entity, double factor) {
  double pitch = (player.getPitch() + 90.0F) * Math.PI / 180.0D;
  double rot = (player.getRotation() + 90.0F) * Math.PI / 180.0D;
  double x = Math.sin(pitch) * Math.cos(rot);
  double z = Math.sin(pitch) * Math.sin(rot);
  double y = Math.cos(pitch);

  entity.moveEntity(x * factor, y + 0.5, z * factor);
}
```

And change the 0.5 on the last line to some other number, say 1.5.

Then change the call to myFling instead of fling:

```
...
Canary.getServer().addSynchronousTask(new CowTask(victim));

myFling(player, victim, 3);
victim.setFireTicks(600);
...
```

Run build.sh to make sure it still works, and commit your changes:

```
$ git commit -a -m 'Moved vector calculation'
[play 411208c] Moved vector calculation
 1 file changed, 9 insertions(+), 1 deletion(-)
```

Now with that safely in a snapshot in the play branch, let's go back and take a look at the original:

```
$ git checkout master
Switched to branch 'master'
```

And now the version of CowShooter.java on disk is the version in the master reality. Take a look at it and see. One note of caution: Git changed the text file on disk. The version in your editor's buffer might be the old one. Most editors are savvy enough to realize when a file has changed out from under them, but how that's handled is up to the editor.

Now back here in master, let's change the cow shooter to shoot creepers instead.

In both CowShooter.java and CowTask.java, add the import for Creeper at the bottom of the list of imports:

```
import net.canarymod.api.entity.living.monster.Creeper; // Add this line
```

In CowShooter.java, change the spawn line to make a Creeper instead of a Cow:

```
Creeper victim = (Creeper)spawnEntityLiving(loc, EntityType.CREEPER);
```

Great! Then over in CowTask.java, change our variable at the top:

```
private Cow cow; // Delete this line
private Creeper creeper; // Add this line
```

And again change all the references from cow to creeper to match (there are a lot).

Once it builds successfully, commit it for safekeeping:

```
$ git commit -a -m 'Changed to shoot creepers'
[master ea618af] Changed to shoot creepers
 2 files changed, 13 insertions(+), 13 deletions(-)
```

While we're working in master, it would be nice to bring the "fling" changes over from the play branch. For that, you can use a git merge:

```
$ git merge play -m 'Bringing over vector calc'
Auto-merging src/cowshooter/CowShooter.java
Merge made by the 'recursive' strategy.
 src/cowshooter/CowShooter.java | 8 ++++++++
 1 file changed, 8 insertions(+)
```

Now take a look at the CowShooter.java file, and you'll see the changes from the play branch have been incorporated. master now contains the move to the fling function and the changes from Cow to Creeper.

Having merged in changes from the play branch doesn't mean it's gone away. There's still a play branch out there and you can still play with it as long as you want. If you are really, truly done with it and never want to see that branch again, you can delete it:

```
$ git branch -d play
Deleted branch play (was f3e6538).
```

And it is unceremoniously deleted from reality, as git branch shows:

```
$ git branch
* master
```

Back Up to the Cloud

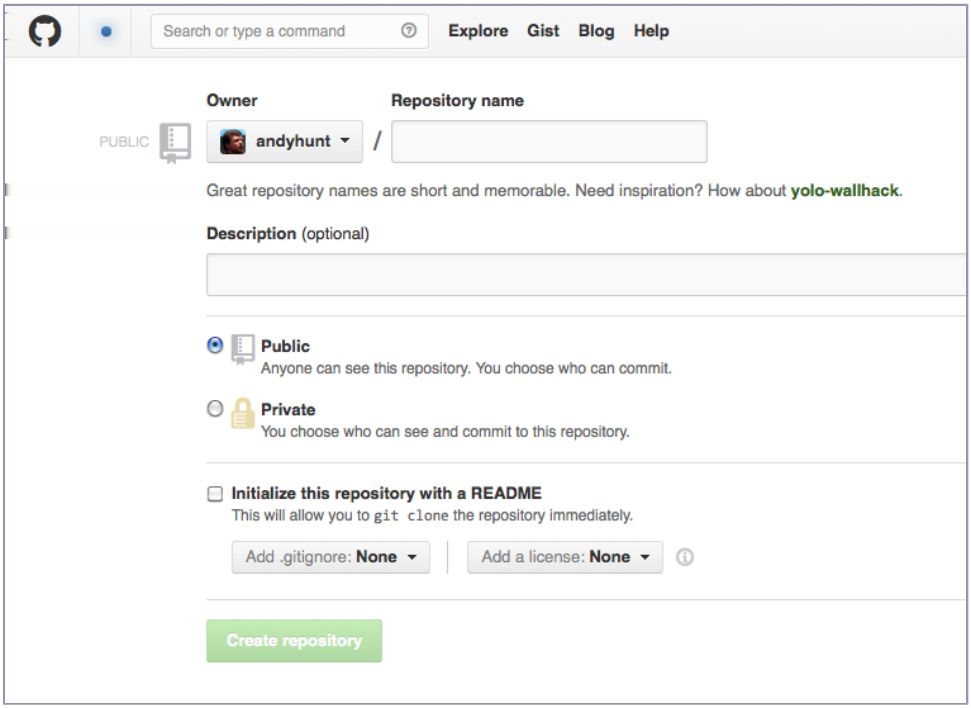
This is a fairly advanced topic. Feel free to skip it on your first reading. But do read it at some point—after all, hard disks fail, laptops break, and thumb drives get lost.

Git keeps a local copy of all of your files and all of your changes in its local repository, and you can set it up to keep a copy of your repository in the cloud as well. You choose when to send changes to the cloud by running the command `git push`. After a push, the remote repository in the cloud will have all the same content and changes you have locally.

To set that up, you'll need an account on a Git host.

The most popular one is GitHub, although many people like Bitbucket as well.[1] [2] GitHub is the standard for code that you intend to share with others or make available as open source, while Bitbucket is better suited to private projects that you don't intend to share with the world.

Both offer really simple web interfaces to get you set up and running. For instance, once you create an account on GitHub, you can click the button to create a new repository, and you'll get a screen similar to the following:

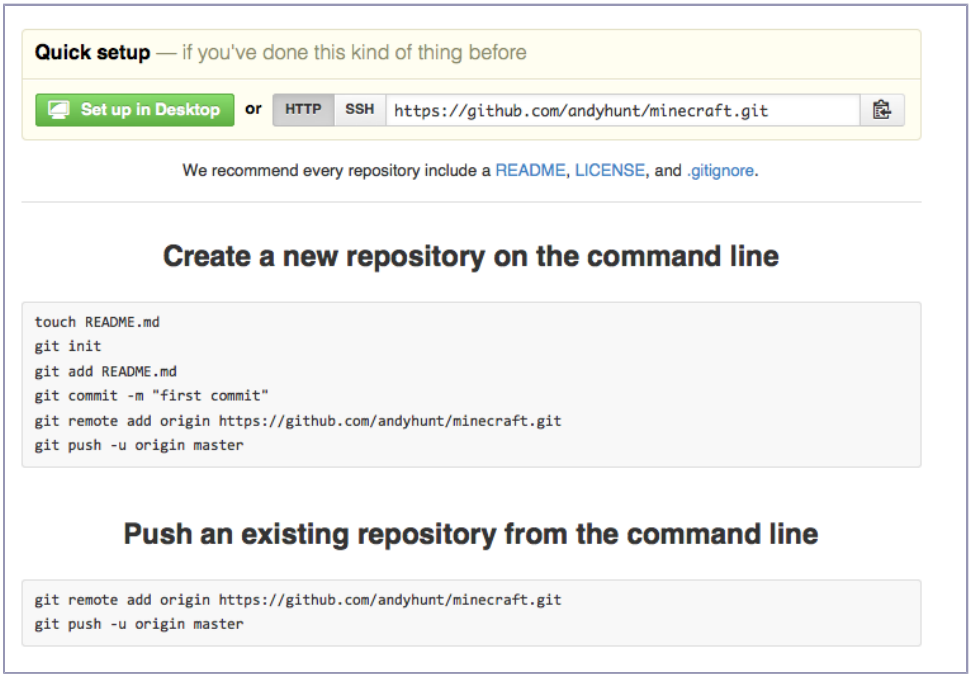


Pick a name for your project, and select Public or Private. Public repositories on GitHub are available for free, but private repositories cost money. Bitbucket, on the other hand, offers unlimited private repositories for free. Don't initialize

1. http://github.com
2. http://bitbucket.com

the repository, as you already have one locally. Just click the Create Repository button, and *bang*—it's done.

Now you'll see instructions on what to do next.



First, copy and save the URL to your repository at the top. In this example, that would be https://github.com/andyhunt/minecraft.git.

Since you already have a local repository, follow the instructions at the bottom for "Push an existing repository from the command line." Mine looks like this:

```
$ git remote add origin https://github.com/andyhunt/minecraft.git
$ git push -u origin master
```

The first command sets up a remote named origin and associates it with the given URL. Next, the push with the -u option sets the master branch to be tracked to origin. Now whenever you reach a good stopping point, you can run git push and all your code and changes will be safely saved in the cloud.

You (or anyone else if it's a public repository) can get a copy of all the code and the history of the changes by using git clone to clone a copy of the repository into an empty directory:

```
$ mkdir newcopy
$ cd newcopy
$ git clone https://github.com/andyhunt/minecraft.git .
```

Friendly Code

When you're writing code to share with others—which means sharing with the whole world—you need to make a little extra effort to write "friendly" code:

- Neatness counts. Put braces where they belong; don't try and jam everything on one line or use inconsistent indentations. The format of the code can be as important as the code itself for human readers.
- Keep it tight. Don't make lines or functions hugely long.
- Comment your functions. Explain *why* a function is there, and *what* it's used for. Don't comment on *how* it does it—that's what the code is for.
- Honor naming conventions. Java prefers camel-case identifiers (like in the word *camelCase*), with embedded capital letters. Other languages may use names with underscores or other conventions. Name variables so that readers can easily understand what they are used for. Although single-letter variables (like `i`) are often used for loop counters, that's not very descriptive. But it's even worse to use `i` (which is normally an index) as a string or a `Location`! Stick to the conventions.

Share Code

Distributing code via GitHub is a very popular way of sharing your code with the world. All you need to do is give out your GitHub project URL (in this example, https://github.com/andyhunt/minecraft.git). Now anyone in the world can use `git clone` to get a copy of your code, and they can make their own changes, compile it, install it, and so on.

One of the advantages of releasing your source code to the world is that other programmers can help add features and fix bugs.

Both GitHub and Bitbucket have nice web interfaces to help with that.

Suppose one of your fans has code for a new feature she'd like you to include in your plugin. She'd fork (make a copy of) your repository and make the proposed changes in a named branch in her repository. Then through the magic of GitHub (or Bitbucket), she'd submit the changes in that branch to you as a *pull request*. That is, she's sending you a request to pull her branch into your repository and use it.

In the web interface, you can reply to pull requests with questions and comments, and have a whole discussion about the changes. If and when you want to incorporate them, just click the Merge Pull Request button. If the new code can merge in without any conflicts, you're done. Otherwise, click the Command Line button, and you'll get further instructions.

We've only scratched the very surface of Git: it's a really powerful and sometimes quite complicated tool. Whole books are devoted just to Git (including *Pragmatic Version Control Using Git [Swi08]* and *Pragmatic Guide to Git [Swi10]*). But hopefully this is enough to get you started.

Next Up

Git is a powerful tool, not just because it gives you a project-wide "undo button," but because it allows you to experiment *with confidence*. You don't have to be afraid that you'll screw everything up. When you *do* screw everything up, Git lets you "unscrew" it back to the point where it all worked. With branches, you can even try a couple of different approaches in code. Not sure if you should use an array or a hash? Not sure if the listener should be in a separate class? You can use branches to try things out.

With that freedom in hand, you're ready for the final step.

git config --global user.email *email*	Set your email address
git config --global user.name "*Name*"	Set your user name
git init	Initialize a Git repo in the current directory
git add *files*	Add files (or directories) to be tracked by Git
git commit -a -m '*Commit message*'	Commit changes (-a all)
git log --oneline *files*	Show commit messages for one or more files
git status	Show what files you've changed, files you've created but Git doesn't know about (untracked), and so on
git diff *files*	Show differences between versions
git checkout *files*	Discard local changes for files (use with caution)
git checkout *name*	Switch to named branch
git branch	List all branches
git branch *name*	Switch to named branch
git remote add origin *url*	Add a remote repo as "origin"
git push -u origin master	Copy changes on the master branch to the remote named "origin"

Next, and finally, we'll look at how you can go about designing and building your own plugin from scratch.

Your Growing Toolbox

You now know how to:

- Use the command-line shell
- Build with Java, `javac`
- Run a Minecraft server
- Deploy a plugin
- Connect to a local server
- Use Java variables for numbers and strings
- Use Java functions
- Use `if`, `for`, and `while` statements
- Use Java objects
- Use imports for Java packages
- Use `new` to create objects
- Add a new command to a plugin
- Work with `Location` objects
- Find blocks/entities
- Use local variables
- Use class-level global variables
- Use `ArrayLists`
- Use `HashMaps`
- Use `private` and `public` to control visibility
- Modify Minecraft blocks
- Modify and spawn entities
- Listen for and react to game events
- Manage plugin permissions
- Create a separate class
- Schedule a task to run later
- Schedule a task to run periodically
- Save and load configuration data
- Build up complex code from simple functions
- Save and load plugin game data
- Use `DataAccess` to use the database
- Catch and throw Java exceptions
- Use Git to keep track of changes to code
- Go back to earlier versions of code (an "undo button")
- Maintain multiple versions of code at the same time
- Back up your code to the cloud

In this chapter you're going to design your own plugin. You'll add the last necessary bits to your toolbox. You'll learn the following:

- How to assign responsibilities to classes
- How to translate responsibilities into functions
- How to do a little design, a little coding, and a little testing, all as you go

You'll be able to start working on your very own plugins.

CHAPTER 13

Design Your Own Plugin

For all the plugins we've looked at so far, I've led the way and shown you what to do. In this chapter I'll give you some suggestions and hints to help you develop your own plugins from scratch. Together we'll go through the steps for a brand-new plugin and see how it works.

We're going to (more or less) follow these steps:

1. Have an idea: I want a plugin to do _____.
2. Gather your materials.
3. Lay them out.
4. Try each part.
5. Knit it all together.
6. Profit!

IMPORTANT NOTE: Even though I've laid out these steps in order, and we'll go through them in order, the real world doesn't usually work that way.

Creativity and invention rarely take a direct, linear path from idea to execution. Instead, you'll discover that something doesn't work the way you thought it did. The code you wrote is all wrong. Or the code is perfectly right but not what you need. That's all totally normal, and really it's what writing code is all about.

You may need to throw out anything you've done and do it over, up to and including the entire project. Hey, even the government does that with $100 billion projects, so we can do it too.

Professional programmers tackle this problem by taking very small steps: do one small, bite-size thing at a time, make sure *that* works, then go on to the next. Any time you realize that you've made a mistake or misunderstood something, go back and fix it right then and there. Don't think you'll remember to fix it later; you won't. Trust me on that one.

Let's dig in and design a new plugin. I'm going to go through these steps on my new plugin, and you'll do the same thing for *your* new plugin alongside of me. Yours should do something different. And it's okay if this takes a while; this isn't the sort of chapter you'll whip through in one evening.

Have an Idea

Before you start, you need some idea of what you want to write. It may not be a great idea or a perfect idea yet, and that's okay. This is *soft*ware, after all, and you're allowed to change your mind. But we need to start somewhere.

So I think I'd like to figure out how to randomly generate "creeper cows"—that is, cows that jump around and try to attack you, and that explode when they jump on you. That might be fun.

At this point, I have no idea how to even start such a thing.

Try This Yourself

What is your idea? Jot it down now. If you're stuck, take a look at plugins other people have written, or just browse the Canary documentation and see if something inspires you.

Gather Your Materials

Armed with an idea, you now "gather your materials." If you were building a craft project, you'd gather all the raw materials (glue, wood, paper, googly eyes, plutonium) and tools (scissors, hammer, arc welder, cooling tower—whatever). We need to do the same. But what materials do we need?

Well, we need a first guess at what sort of data we'll require in the plugin, how to keep track of it, and what makes the plugin run—is it run from a command, or an event, or a timer? Here are some good questions to ask:

- What do you need to keep track of?
- How long do you need to keep it? (during a command, while the server is running, across server reboots on disk…)
- What are the triggers? (user-entered command, in-game event, internal state, timer, combination…)
- What parts of the game do you need to affect (blocks, players, potions, inventory, and so on) and what Canary or Java functions will you need to use for them?
- What can go wrong?

You may not have all the answers yet. But that won't stop us; let's get started (using my "creeper cow" idea).

What do you need to keep track of? We probably need to keep track of the cows we spawn. We might need to keep track of players that each cow is targeting and trying to attack, or maybe we'll figure that out for each cow as we go along. We don't know which yet. Anything else? Oh, we want to randomly create locations for the cows; do you remember what function to use to get random numbers in Java?

How long do you need to keep it? Since these are randomly generated attack cows, I don't think we need to keep track of them when the server shuts down—we'll just make new ones. We will need to keep track of them in memory while the server is running, though, so at a minimum we'll need some kind of a static list or hash of the cows we've spawned. Reread Chapter 7, *Use Piles of Variables: Arrays*, on page 89, or Chapter 11, *Use Configuration Files and Store Game Data*, on page 145, if you need a refresher.

What are the triggers? It would be kind of stupid to have to type in a command to get attacked by creeper cows, so we won't be using a `@Command` function. Instead, we'll need an event to kick off cow generation, and maybe a timer to keep the cow attacking and make it eventually explode. We'll have to find some kind of suitable event to listen for. Have a look back at Chapter 9, *Modify, Spawn, and Listen in Minecraft*, on page 117, and Chapter 10, *Schedule Tasks for Later*, on page 135, for details.

What parts of the game do you need to affect? We'll need to move the killer cows around and blow them up, which we know how to do from the earlier plugins. We'll need to blow up the player that we're attacking. Can we just blow up when the cow is next to a player, or do we need to explicitly kill the player (by setting health to zero or setting the player on fire or something)? We'll need to experiment and see how that works.

What can go wrong? This is a question to ask yourself *constantly* when creating a plugin. For now it seems likely our cows could get confused when attacking and get stuck somewhere, or not have any player nearby to attack. So we'll need to deal with that. Also, it's probably a good idea to limit the number of cows we're going to spawn, so we don't accidentally create the Great Cow Deluge.

It's not much, but it's a start. Here's what we've gathered so far:

- A static list or hash of cows we've spawned
- The Java function to make a random number (what was that again?)

- An event to spawn cows
- The idea that each cow needs a timer so it has a chance to find players and attack them
- The idea that the cow needs to not get stuck in the terrain (what if there's no player nearby?)
- The knowledge that we may need to put a limit on the number of cows to spawn

Try This Yourself

Now go through these questions for the plugin you want to build, and come up with your own list of things you'll need—your "materials."

Include things you aren't sure about—there's no penalty if you end up not using them.

Lay Them Out

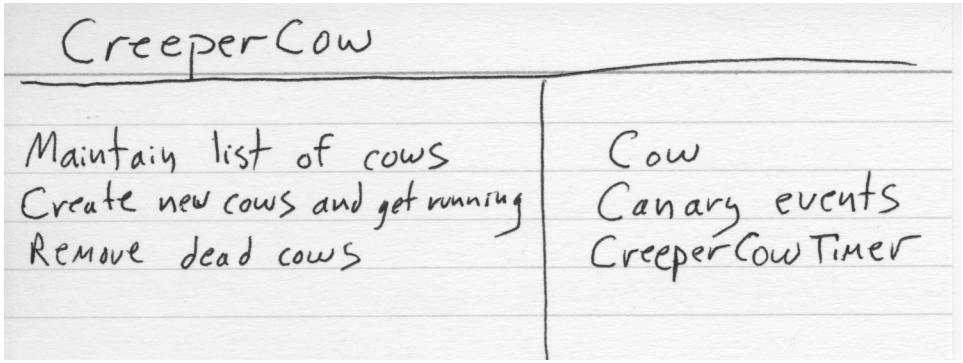
Before we create the plugin and start writing code, let's think about what might go where. In other words, will this fit all in one function or even one class, and if there are several functions or classes, what goes where? If there are additional classes, what do they need to know about us or each other? If you've forgotten some details of functions and objects, check back with Chapter 4, *Plugins Have Variables, Functions, and Keywords*, on page 43, and Chapter 5, *Plugins Have Objects*, on page 67, for a refresher.

So here are our questions:

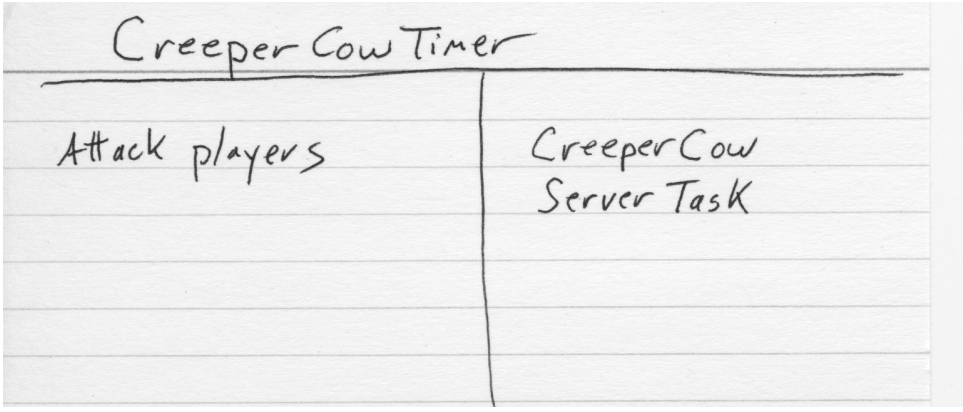
- What goes where? What functions and classes do you need?
- Why do you need this function (or class)? What is it responsible for?
- Who else needs to know about this function or class (if anyone)?

One way of attacking this, especially on complicated plugins, is by using good old-fashioned index cards. (These are called CRC cards, where CRC is short for "class-responsibility-collaborators," invented by our friends Ward Cunningham and Kent Beck.) Divide each card into three parts: a title up top, a list of things it is responsible for on the left, and other classes it needs to work with on the right.

So our main plugin class, so far, would look like this:



Then there's the `CreeperCowTimer`, which will hold a cow and take care of the things an attacking cow needs to do:



Notice we haven't gotten into any detail of particular functions yet: you want to get a firm idea in your head of *why* a particular class needs to exist. What is it responsible for doing?

For each responsibility, take a stab at working out what it needs to do. So as not to get too hung up on Java syntax and issues (or those of any other programming language, for that matter), say what you want to do in plain English. Tell the story of *what* you want to do—don't worry about *how* yet.

Each step will become a function, or *multiple functions* if you decide to split the step into a couple of simpler functions and/or classes.

1. Create our list of cows.
2. Spawn creeper cows and add them to our list.
3. Set up each creeper cow with a timer.
4. Listen for events to create and remove cows.

That part was easy. But what's a creeper cow to do? We said its responsibility is to "attack players." What does that mean? How about this:

1. Find the closest player. (There might not be one.)
2. Jump toward the closest player. (Don't die from the impact!)
3. If you hit the closest player, explode.

So for the CreeperCow plugin, let's take our list of gathered materials plus the list of steps, make up some function names, and see what we need to code:

CreeperCow (the main plugin class):

- cowList: A static list of cows we've created.
- Math.random(): We need a random number, and the Java docs say this will return a double between 0 and 1.[1]
- spawnCows(): Create some random number of cows, no more than some established limit. We'll create a CreeperCowTimer to hold each cow and set its timer going.
- eventListener(): A listener to call spawnCows().

CreeperCowTimer (one per cow):

- findClosestPlayer(): There might not be one.
- jump(Location loc): Jump means fly up into the air and don't die on impact.
- explode(): If we hit a player (or are close enough), explode and tell the plugin to remove this cow.

There's a lot we haven't figured out yet, but this gives us enough to start putting some code together. And *that* will help us figure out the remaining questions. And probably raise some new ones. Onward we go.

Try This Yourself

Get a couple of index cards or some bits of paper, and jot down your main class and its responsibilities, and any parts of Canary it might need. If you need extra classes as we did here (like for a task runner), or anything else you need to keep track of, create those cards as well.

For each responsibility, write a list of steps that reads like a story. Then go through your list of "materials" and take a first stab at making a list of functions and variables, asking the same questions we asked here.

1. http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html#random%28%29

Once you have your list together, it's time to start writing some code—but not all at once, as we'll see in the next section.

Try Each Part

We're going to start coding now, but as exciting as that is, we want to go slowly. Whether you're following along with me here or working on your own plugin, remember to take one small step at a time—don't rush it.

Let's begin by creating a new directory and the usual files that every plugin needs. Once again we'll use the mkplugin.sh script to get started. Don't forget, you'll be making your own plugin under desktop; my version is under Desktop/code.

```
~/Desktop$ ./mkplugin.sh CreeperCow
~/Desktop$ cd CreeperCow
~/Desktop/CreeperCow$ ls -a
.            .gitignore   Manifest.txt build.sh     src
..           Canary.inf   bin          dist
```

And we'll start right away by using Git to track our work (check back with Chapter 12, *Keep Your Code Safe*, on page 169, for a refresher if you need to).

The mkplugin.sh script thoughtfully made us a .gitignore that will ignore the bin/ and dist/ directories. So we'll add our first couple of files and commit them as a baseline:

```
~/Desktop/CreeperCow$ git init
Initialized empty Git repository in /Users/andy/Desktop/CreeperCow/.git/
~/Desktop/CreeperCow$ git add .gitignore build.sh Canary.inf Manifest.txt src
~/Desktop/CreeperCow$ git commit -a -m First
[master (root-commit) 2e483f1] First
 5 files changed, 95 insertions(+)
 create mode 100644 .gitignore
 create mode 100755 build.sh
 create mode 100644 Canary.inf
 create mode 100644 Manifest.txt
 create mode 100644 src/creepercow/CreeperCow.java
```

(If you are hooked up to a remote repository, you can do a git push now as well.)

Now on to the code.

We know that we need to create a static cowList and a spawnCows() method, and we'll need to listen for some events, so let's start with that in CreeperCow.java. Here's the interesting part (omitting other imports and the usual other stuff):

```
import net.canarymod.plugin.PluginListener;
import net.canarymod.api.entity.living.animal.Cow;
import java.util.ArrayList;
import com.pragprog.ahmine.ez.EZPlugin;

public class CreeperCow extends EZPlugin implements PluginListener {

  private static ArrayList<Cow> cowList = new ArrayList<Cow>();

  public void spawnCows() {

  }
```

We made the main plugin a PluginListener and added a cowList variable and the start of the spawnCows() function. Before we go any further, let's test *just that much*—see if it builds without errors, and run the (empty) spawnCows function.

Small steps, remember. We don't have an event listener yet (in fact, we don't even know *what* event we're going to listen for), so we'll use a trick.

We're going to add a command to @Command that's just for us—it's not for users. But we can use it to test what we're doing as we go along.

So let's wire that up first, even before we've added any code to spawnCows:

```
@Command(aliases = { "testspawncows" },
    description = "Test cow spawning",
    permissions = { "" },
    toolTip = "/testspawncows")
public void testSpawnCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
    Location loc = me.getLocation();
    spawnCows();
  }
}
```

(We'll need to import net.canarymod.api.world.position.Location; too.)

Now we have the ability to log in to the game and run the /testSpawnCows command to watch our cows spawn and make sure that works.

We don't actually spawn any cows yet, but let's just try it now and make sure everything compiles, using build.sh, before we add any more code.

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

That's a good opportunity to make sure we've got the right imports and everything, and haven't made any typos. And since it's working, I'll do a `git commit`.

Try This Yourself

Now do this much yourself: create a new plugin directory with `mkplugin.sh`, get a Git repository set up for it, and add any functions and data that you think you need. Refer back to earlier parts of the book if you need a refresher on anything.

Add the testing commands to `@Command`, even if you have real commands that you're going to add as well. Again, small steps. You want to make small functions that run by themselves, and get them right before continuing. Don't try to flesh out the functions yet; just start with empty function bodies like we did here.

Don't forget to do a `git commit` to "save your game" as you go along.

Filling In the Details: the spawnCows() Function

Now that we have a way to test it, let's move on to the guts of `spawnCows()` itself. We noted earlier that we need to do the following:

1. Create our list of cows.
2. Spawn creeper cows and add them to our list.
3. Set up each creeper cow with a timer.
4. Listen for events to create and remove cows.

Let's take this one at a time. We already made the simple static list of cows, so let's look at spawning.

We'll need to create some random cows. This raises a question: where should we put them? All over the game? Near where you are? Right on top of your friend's head? All of these choices may have their appeal.

One of the best things you can do when programming is to delay making decisions. In many cases, you don't know the "right" answer, and you may not know it for a while—or ever. But that's what function parameters are for: we don't have to decide the details right now. Instead of implementing some particular detail, pass that it in as a parameter. Now you don't have to decide, and the function can be used with all sorts of different details.

So we need to change the temporary declaration we had in place for `spawnCows()` and pass in something to indicate where we want these cows. What should that look like?

Let's start with a Location, and then a number of blocks square. The location will be one corner of a large square, and the square will then extend for some number in the x and z directions. So we'll change the function signature to look like this:

```
public void spawnCows(Location start, int size) {
```

Then we can decide later what to pass in. For testing, we can pass in your Player location and a number you pick. Let's wire that up now.

Ah, one small hitch: a command function passes us the arguments to a user command as an array of String objects. We don't want a string to pass to testSpawnCows(), though—we want an int. How do you do that conversion, again?

A Google search for "java convert string to int" reveals the magic incantation:

```
int foo = Integer.parseInt("1234");
```

Now we can finish up the command to call spawnCows from our test command:

```
@Command(aliases = { "testspawncows" },
    description = "Test cow spawning",
    permissions = { "" },
    min = 2, // Number of arguments
    toolTip = "/testspawncows <number to spawn>")
public void testSpawnCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
    Location loc = me.getLocation();
    spawnCows(loc, Integer.parseInt(args[1]));
  }
}
```

Okay, *now* it's safe to add the code to implement spawnCows, because we have a way to check it. Professionals work like this: write some way of checking your code before you write the code itself. We can't quite do it all automatically—we still have to manually log in to the game and visually confirm that we're spawning creeper cows, but it's a similar idea.

So we need to start making some cows! We'll use a for loop, the starting location, and the size to create a new location to spawn each new cow. How many cows to spawn? Oops. We didn't think about that. Better add that to the function signature:

```
public void spawnCows(Location start, int size, int number) {
```

and pass it in from the testSpawnCommand() for testing:

```
spawnCows(loc, Integer.parseInt(args[1]), Integer.parseInt(args[2]));
```

Now we know how to do the for loop to create cows, and add them to our list of Cows. We can finally put the code in spawnCows:

```
public void spawnCows(Location target, int size, int count) {
  World world = target.getWorld();
  double x = target.getX();
  double z = target.getZ();
  for (int i=0; i< count; i++) {
    Location loc = new Location(world,
      x + (Math.random() * size),
      0,
      z + (Math.random() * size),
      0,0
    );
    loc.setY(world.getHighestBlockAt((int)loc.getX(), (int)loc.getZ()) + 2);
    Cow cow = (Cow)spawnEntityLiving(loc, EntityType.COW);
  }
}
```

To recap: we're multiplying the random number (0..1) by the size of the square, so that will give us a number between zero and the size of the square. We'll use that for an x and z of the location on the square, then ask the server for the highest block at that point and use that as the y. That's the location where we'll spawn the new cow.

Let's compile and build it, and test it out.

I'll log in to the game and try the command /testspawncows 10 8, which will spawn eight cows within a 10×10 block from my current location.

Hey, we made some cows! They aren't very frightening, though. They're just sitting there, as cows do. We need to get them to jump around and attack.

I've written some code, tested it, and now I'll do a git commit to save this state of the world before going on.

Try This Yourself

Your turn! Start fleshing out your functions just as I did here. Don't feel they have to do *everything* yet—just enough to get started (for instance, our cows here don't jump yet).

For each function you make, include a test command so you can try it from inside the game.

Make sure all your test commands work with as much as you have done so far. And don't forget to save your progress with Git.

Filling In the Details: CreeperCowTimer

Earlier we decided to put the code for the jumping, attacking, exploding cow into a new CreeperCowTimer class. Eventually we'll crank this up from an event in the main CreeperCow plugin.

But first off we need a function to make a cow jump and attack. We'll build and test that first.

Given a target and our cow's location, we create a new Vector3D by taking the difference in Location coordinates in the x and z directions: that gives us the displacement between the cow and its target. But that's too large a number for a velocity, so we then multiply it by a convenient number (0.075) that I found by experimenting. Finally we set that as the cow's velocity, which will make it jump along that Vector3D at that speed.

```
public class CreeperCowTimer {
  private Cow cow;

  public void jump(Location target) {
    Location cowLoc = cow.getLocation();
    double multFactor = 0.075;
    Vector3D v = new Vector3D(
      (target.getX() - cowLoc.getX()) * multFactor,
      0.8,
      (target.getZ() - cowLoc.getZ()) * multFactor
    );
    cow.moveEntity(v.getX() + (Math.random() * -0.1),
      v.getY(),
      v.getZ() + (Math.random() * -0.1));
  }
}
```

I'll do a git add on the new CreeperCowTimer.java file so my changes can be tracked.

Great. Now we need some way of setting that private cow object. We'll hook this up to the spawnCows function later, but right now we'll just pass in a cow that's been spawned already. For that, we'll use the constructor function:

```
CreeperCowTimer(Cow aCow) {
  cow = aCow;
}
```

Next we need to add a jump command to the plugin for our testing. Now we'll go back into the spawnCows function and add the spawned cows to the ArrayList. Then, in the jump command we can go through the list and get each of the spawned cows to jump toward us.

So we'll start with this:

```
Cow cow = (Cow)spawnEntityLiving(loc, EntityType.COW);
cowList.add(cow);
```

Then we'll add a test command:

```
@Command(aliases = { "testjump" },
    description = "Test cow jumping",
    permissions = { "" },
    min = 1, // Number of arguments
    toolTip = "/testjump")
public void testJumpCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
      for (Cow c : cowList) {
        c.jump(me.getLocation());
      }
  }
}
```

Oops, that's not going to work.

We shouldn't have kept a list of cows. We really need to keep track of CreeperCowTimer objects, as that will get us the jump() function and the other guts that we need to write, as well as the Cow itself.

And if we do that, we should probably change the list to be a HashMap instead of an ArrayList so we can look up the CreeperCowTimer objects by their Cow, as that's what we'll get from events and spawning and such. We'll need to redo a few things now.

Make a list of things we need to change for this, and come on back once you have it.

Changes We Need

Here's what I came up with:

1. Change the `import java.util.ArrayList;` to `import java.util.HashMap;`.
2. Change the `cowList` from `public static ArrayList<Cow> cowList = new ArrayList<Cow>();` to use a hash instead: `public static HashMap<Cow, CreeperCowTimer> allCows = new HashMap<Cow, CreeperCowTimer>();`. Also, I renamed it to "allCows" so that the type of list isn't part of the name.
3. Change the `cowList.add(cow);` to `allCows.put(cow, new CreeperCowTimer(cow));` since you have to `put` in a hash.
4. Change the `for` loop to iterate through the `HashMap` so the new loop looks like this:

```
for (Cow c : allCows.keySet()) {
  CreeperCowTimer superCow = allCows.get(c);
  superCow.jump(me.getLocation());
}
```

Right about now this might feel frustrating.

That's okay: it's not important to get it right the first time. That rarely happens and doesn't gain you much. It *is* important to get it right the last time.

With these changes we can now test jumping. I'll fire up the server, connect from the client, and test by spawning just a single cow first. Then I can use the `testJump` command to see if it starts heading toward me:

```
/testspawncows 5 1
```

Great! Now for a few jumps:

```
/testjump
/testjump
/testjump
```

Ah, right. The cow died. We ran into that same problem back in the CowShooter plugin—we have to set the cow's health to the max using something like this:

```
cow.setHealth(cow.getMaxHealth());
```

But where should we do it? And what else do we need to finish up this plugin? The current version is working but has a problem. Do a git commit to save the current state of the code at this point before we start making more major changes.

Knit It All Together

We need to add a few things from our list of materials and take care of a couple of problems.

First off, we need to fix the dying cows, so let's add an event handler to listen for DamageHook, and cancel the event if its cause is DamageType.FALL. Ah, if we cancel the damage event, we don't need to bump up the cow's health.

When testing the spawn function and the jump, we used ourselves as the target. We need to do a little better than that, so we'll have to add a getClosest-Player() function to find the closest player to any given cow.

And as we saw earlier, we need to pick an event to use to spawn these creeper cows. There are probably a couple of ways to do that. You don't want to spawn cows all over the world; it would be great if there were an event that gets sent when each new chunk of the world gets loaded into the server.

A quick look through the Canary documentation for events under net.canarymod.hook.world reveals just what we need: a ChunkLoadedHook when a part of the world is loaded, and a ChunkUnloadHook when it's deleted. So each time a new 16×16 chunk of the world gets loaded, we'll get an event. One minor nit: the chunk tells us which chunk it is in a grid of chunks; you have to multiply by 16 to get a real-world coordinate.

Our to-do list now includes these items:

- Add an DamageHook event handler so the cow doesn't die.
- Add getClosestPlayer so each cow can find a nearby target.
- Add ChunkLoadedHook to spawn cows in a 16×16 area.
- Add the ChunkUnloadHook to remove cows from that chunk.
- Set up a task timer so each cow can find a target, jump, and explode if needed.
- Manage the allCows in all these functions (add to it on spawn, delete it on death, or unload).

Phew! That's a bit of work. But all of these activities are using functions and skills we've used already, so I won't bore you with a lengthy play-by-play.

Here is the code in its entirety for you to read over and crib from as you need.

CreeperCow/src/creepercow/CreeperCow.java

```
package creepercow;

import java.util.HashMap;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Collection;
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.Entity;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.world.World;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.position.Vector3D;
import net.canarymod.hook.HookHandler;
import net.canarymod.api.inventory.ItemType;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import net.canarymod.api.entity.EntityType;
import net.canarymod.api.entity.living.animal.Cow;
import net.canarymod.api.entity.EntityType;
```

```
import net.canarymod.plugin.PluginListener;
import net.canarymod.hook.world.ChunkLoadedHook;
import net.canarymod.hook.world.ChunkCreatedHook;
import net.canarymod.hook.world.ChunkUnloadHook;
import net.canarymod.hook.entity.DamageHook;
import net.canarymod.api.DamageType;
import net.canarymod.api.world.Chunk;
import com.pragprog.ahmine.ez.EZPlugin;

public class CreeperCow extends EZPlugin implements PluginListener {

  private static HashMap<Cow, CreeperCowTimer> allCows =
    new HashMap<Cow, CreeperCowTimer>();

  private static boolean enabled = false;

  private final static int CHUNK_SIZE = 16;

  @Override
  public boolean enable() {
    Canary.hooks().registerListener(this, this);
    return super.enable(); // Call parent class's version too.
  }

  public void spawnCows(Location target, int size, int count) {
    World world = target.getWorld();
    double x = target.getX();
    double z = target.getZ();
    for (int i=0; i< count; i++) {
      Location loc = new Location(world,
        x + (Math.random() * size),
        0,
        z + (Math.random() * size),
        0,0
      );
      loc.setY(world.getHighestBlockAt((int)loc.getX(), (int)loc.getZ()) + 2);
      logger.info("[CreeperCow] spawned cow at " + printLoc(loc));
      Cow cow = (Cow)spawnEntityLiving(loc, EntityType.COW);
      CreeperCowTimer task = new CreeperCowTimer(this, cow);
      Canary.getServer().addSynchronousTask(task);
      allCows.put(cow, task);
    }
  }

  public void cowDied(Cow cow) {
    logger.info("[CreeperCow] cow died.");
    allCows.remove(cow);
  }

  @Command(aliases = { "creepercows" },
```

```
    description = "Turn Creeper Cows on and off",
    permissions = { "" },
    min = 2, // Number of arguments
    toolTip = "/creepercows on|off")
public void enabledCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
    if (args[1].equalsIgnoreCase("on") ||
      args[1].equalsIgnoreCase("yes") ||
      args[1].equalsIgnoreCase("true")) {
      enabled = true;
      me.chat("Creeper Cows are enabled");
      // Start off with a few right here ;)
      spawnCows(me.getLocation(), 25, 5);
    } else {
      enabled = false;
      me.chat("Creeper Cows are disabled");
    }
  }
}

@Command(aliases = { "testspawncows" },
    description = "Test cow spawning",
    permissions = { "" },
    min = 3, // Number of arguments
    toolTip = "/testspawncows <size of square> <number to spawn>")
public void testSpawnCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
    Location loc = me.getLocation();
    spawnCows(loc,
        Integer.parseInt(args[1]),
        Integer.parseInt(args[2]));
  }
}

@Command(aliases = { "testjump" },
    description = "Test cow jumping",
    permissions = { "" },
    min = 1, // Number of arguments
    toolTip = "/testjump")
public void testJumpCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
    for (Cow c : allCows.keySet()) {
      CreeperCowTimer superCow = allCows.get(c);
      superCow.jump(me.getLocation());
    }
  }
}
```

```
@Command(aliases = { "testexplode" },
    description = "Test cow explode",
    permissions = { "" },
    min = 1, // Number of arguments
    toolTip = "/testexplode")
public void testExplodeCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
    List<CreeperCowTimer> list = new ArrayList<CreeperCowTimer>();
    for (Cow c : allCows.keySet()) {
      CreeperCowTimer superCow = allCows.get(c);
      list.add(superCow);
    }
    for (CreeperCowTimer superCow : list) {
      superCow.explode();
    }
  }
}

@HookHandler
public void onChunkLoad(ChunkLoadedHook event) {
  if (enabled) {
    World world = event.getWorld();
    Chunk chunk = event.getChunk();

    if (Math.random() > 0.10) { // Only make a cow 1 in 10
      return;
    }
    logger.info("[CreeperCow] Spawning");
    // The X and Z from the chunk are indexes;
    // we have to multiply by 16 to get an actual
    // block location.
    Location start = new Location(
        chunk.getX() * CHUNK_SIZE,
        0,
        chunk.getZ() * CHUNK_SIZE);
    spawnCows(start, 16, 1);
  }
}

@HookHandler
public void onChunkUnload(ChunkUnloadHook event) {
  Chunk chunk = event.getChunk();
  List<Entity>[] all = chunk.getEntityLists();
  for(int i = 0; i < all.length; i++) {
    for (Entity ent : all[i]) { // List of 16 block subchunks
      if (ent instanceof Cow) {
        Cow cow = (Cow) ent;
        if (allCows.containsKey(cow)) {
```

```
            allCows.get(cow).removeMe();
            allCows.remove(cow);
          }
        }
      }
    }
  }

  @HookHandler
  public void onEntityDamage(DamageHook event) {
    Entity ent = event.getDefender();
    if (ent instanceof Cow) {
      Cow cow = (Cow) ent;
      if (event.getDamageSource().getDamagetype() == DamageType.FALL) {
        if (allCows.containsKey(cow)) {
          event.setCanceled();
        }
      }
    }
  }

}
```

CreeperCow/src/creepercow/CreeperCowTimer.java

```
package creepercow;

import java.util.List;
import java.util.ArrayList;
import net.canarymod.Canary;
import net.canarymod.api.entity.EntityType;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.entity.living.animal.Cow;
import net.canarymod.api.world.position.Vector3D;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import net.canarymod.tasks.ServerTask;
import com.pragprog.ahmine.ez.EZPlugin;


public class CreeperCowTimer extends ServerTask {
  private Cow cow;
  private CreeperCow plugin;

  CreeperCowTimer(CreeperCow parentPlugin, Cow aCow) {
    super(Canary.getServer(), 0, true); // delay, isContinuous
    cow = aCow;
    plugin = parentPlugin;
  }

  public Player getClosestPlayer(Location loc) { //return -1 on failure
```

```
  List<Player> list = Canary.getServer().getPlayerList();
  Player closestPlayer = null;
  double minDistance = -1;
  for(int i = 0; i < list.size(); i++) {
    Player p = list.get(i);
    Location ploc = p.getLocation();
    if (Math.abs(ploc.getY() - loc.getY()) < 15) {
      double dist = distance(loc, ploc);
      if (dist < minDistance || minDistance == -1) {
        minDistance = dist;
        closestPlayer = p;
      }
    }
  }
  return closestPlayer;
}
//
// Find the distance on the ground (ignores height)
// between two Locations
//
public double distance(Location loc1, Location loc2) {
  return Math.sqrt(
    Math.pow(loc1.getX() - loc2.getX(), 2) +
    Math.pow(loc1.getZ() - loc2.getZ(), 2)
  );
}

// Explode yourself
public void explode() {
  plugin.cowDied(cow); // notify parent
  Location cowLoc = cow.getLocation();
  cow.getWorld().makeExplosion(cow,
        cowLoc.getX(), cowLoc.getY(), cowLoc.getZ(),
        3.0f, true);
  removeMe();
}

// We are all done, either from chunk unload or explosion
public void removeMe() {
  cow.kill();
  Canary.getServer().removeSynchronousTask(this);
}

// Jump this cow toward the target
public void jump(Location target) {
  Location cowLoc = cow.getLocation();
  double multFactor = 0.075;
  Vector3D v = new Vector3D(
    (target.getX() - cowLoc.getX()) * multFactor,
    0.8,
```

```
      (target.getZ() - cowLoc.getZ()) * multFactor
    );
    cow.moveEntity(v.getX() + (Math.random() * -0.1),
      v.getY(),
      v.getZ() + (Math.random() * -0.1));
  }

  // Callback to run and execute body of task
  public void run() {
    if (cow.isOnGround()) { // otherwise it's still jumping
      Location cowLoc = cow.getLocation();
      Player p = cow.getWorld().getClosestPlayer(cow, 10000);
      if (p == null) {
        return;
      }
      Location pLoc = p.getLocation();
      double dist = distance(cowLoc, pLoc);

      if (dist <= 4) {
        explode();
      } else if (dist <= 200) {
        jump(pLoc);
      }
    }
  }

}
```

You might notice that I didn't actually implement everything I mentioned in my lists. For example, there's no check for spawning too many cows, and I don't kill the target player directly. I may end up adding those, or just let it be. Just because I *thought* I needed those elements doesn't mean I have to write them yet. I can always add them later if needed.

And I might run into other problems that I hadn't thought about. For instance, what happens to these cows when the server shuts down? The cows will still be in the world, but they won't be CreeperCows anymore, as the plugin doesn't keep track of them. Maybe I should despawn the cows on shutdown. Or maybe having extra cows isn't really a problem? Ah, software.

Try This Yourself

That was our journey with the CreeperCow plugin. Your journey with your plugin will probably be a little different. But try to follow these same general steps: figure out what parts you need, possibly using index cards, then write out a sequence in English of what should happen. Take all that and create your functions, pass around the data you need to, store the stuff you need to

remember, and always add test commands so you can test each part separately.

Remember to remove the test commands before shipping your plugin to your friends. Or, for a bit of extra flash, set it up so that you need a special "Developer" permission to run the tests, and give yourself that permission.

And that's it! Your toolbox is now complete:

Your Full Toolbox!

You now know how to:

- Use the command-line shell
- Build with Java, `javac`
- Run a Minecraft server
- Deploy a plugin
- Connect to a local server
- Use Java variables for numbers and strings
- Use Java functions
- Use `if`, `for`, and `while` statements
- Use Java objects
- Use imports for Java packages
- Use `new` to create objects
- Add a new command to a plugin
- Work with `Location` objects
- Find blocks/entities
- Use local variables
- Use class-level global variables
- Use `ArrayLists`
- Use `HashMaps`
- Use `private` and `public` to control visibility
- Modify Minecraft blocks
- Modify and spawn entities
- Listen for and react to game events
- Manage plugin permissions
- Create a separate class
- Schedule a task to run later
- Schedule a task to run periodically
- Save and load configuration data
- Build up complex code from simple functions
- Save and load plugin game data
- Use `DataAccess` to use the database
- Catch and throw Java exceptions
- Use Git to keep track of changes to code
- Go back to earlier versions of code (an "undo button")
- Maintain multiple versions of code at the same time
- Back up your code to the cloud
- Use CRC cards to think about classes and responsibilities
- Decompose responsibilities into functions
- Test as you go

Just the Beginning

It's been a fun trip, but we've barely scratched the surface. There is more Java you need to learn, and there's a lot more to Canary than we've covered here. Plus, there's a ton more to programming in general that you'll discover as you go along.

But I hope this has been a fun start for you. Don't stop now! Get a couple more books on Java or another language, on programming, on web design—whatever. Never stop reading and learning.

And when you make something really cool, email me and let me know.

Thanks for buying this book, and all the best,

andy@pragprog.com

APPENDIX 1

How to Read Error Messages

Error messages from the Java compiler, the runtime system, and the Minecraft server try to be self-explanatory, but they don't always succeed. The javac compiler in particular can get more than a little confused and spit out unhelpful messages.

Please read through this whole appendix even if you aren't getting a particular error at the moment, as some of this information might help you decipher other error messages that aren't included here.

I've included some of the common error messages you might run into, along with some observations and commentary. If you run into an error that you just can't figure out, and it isn't listed here, try Googling for the text of the error message. Odds are that someone else has had the same problem at some point, and you can benefit from their experience.

Java-Compiler Error Messages

Java-compiler error messages usually look something like this:

```
src/helloworld/HelloWorld.java:21: cannot find symbol
symbol  : class MessageReceiver
location: class helloworld.HelloWorld
      public void helloCommand(MessageReceiver caller, String[] parameters)
```

Java is trying to tell you exactly where the error occurred and what it thinks the problem is.

The first bit of text is the name of the file where Java thinks the problem is located—in this case, the file src/helloworld/HelloWorld.java. Next is a number in between colons—that's the line number in the file (21 here). Next is the error message itself, "cannot find symbol." After that come the details specific to

this error message, which in this example is the symbol that it can't find, and some more information about the location of the missing symbol.

So on line 21 of HelloWorld.java, Java doesn't know about the thing named MessageReceiver. Let's look at some possible causes.

javac: Cannot Find Symbol

"Cannot find symbol" means that the compiler has come across a word, a piece of text, that it doesn't understand.

This error can be caused by several problems. First, what happens if I just stick in an assignment statement like i = 10 without ever declaring what i is?

```
src/helloworld/HelloWorld.java:23: cannot find symbol
symbol  : variable i
location: class helloworld.HelloWorld
        i = 10;
        ^
```

The compiler has no idea what i is or what it should be, so it complains.

To fix it I can add a declaration like int i; above this code or on the same line as int i = 10;. Now the compiler knows that i is a local variable.

But what if I have declared the variable and I still get an error? For instance, in the call to helloCommand, I'm declaring a parameter MessageReceiver caller. caller is my variable, of type MessageReceiver, but I get the same error:

```
src/helloworld/HelloWorld.java:21: cannot find symbol
symbol  : class MessageReceiver
location: class helloworld.HelloWorld
      public void helloCommand(MessageReceiver caller, String[] parameters)
```

This can indicate a missing or misspelled import statement. The compiler knows that caller is a variable of type MessageReceiver, but it doesn't know what a MessageReceiver is.

In this case, adding

```
import net.canarymod.chat.MessageReceiver;
```

at the top of the file fixes the error. See Appendix 7, *Common Imports*, on page 253, for a list of common imports we've used in the book, or look it up in the Java or Canary doc.

javac: Missing Semicolon

```
src/helloworld/HelloWorld.java:6: ';' expected
import net.canarymod.chat.MessageReceiver
```

We forgot the semicolon at the end of the import line. It should be this:

```
import net.canarymod.chat.MessageReceiver;
```

However, this error message isn't always foolproof. For instance, if I leave off the opening brace, {, of a code block by mistake, like this:

```
public void disable()
  log.info("Stopping.");
}
```

I'll get a slew of errors, starting with "missing semicolon" and continuing on to the next several lines of code, past our initial error:

```
src/helloworld/HelloWorld.java:18: ';' expected
  public void disable()
                      ^
src/helloworld/HelloWorld.java:21: class, interface, or enum expected
  public void helloCommand(MessageReceiver caller, String[] parameters)
         ^
  ...
```

The compiler is confused: it thinks maybe we should have had a semicolon after `onDisable()`, but actually we needed an opening brace, {. Then it has no idea what's supposed to be happening, and it starts complaining that the *entire rest of the file* is not what it expected.

That's why in most cases you only want to read the first one or two errors and ignore the rest for the moment, as many of the following errors will disappear once you fix the first one.

javac: Illegal Start of Expression

This is a generic error message if we didn't tidy up an expression as we should have and the compiler isn't ready to start a new expression yet—for instance, if I leave off the closing brace at the end of a function:

```
public void disable() {
  log.info("Stopping.");
```

Again I'll get a bunch of errors, starting at the point of the missing brace and continuing way past it into the rest of the file:

```
src/helloworld/HelloWorld.java:21: illegal start of expression
  public void helloCommand(MessageReceiver caller, String[] parameters)
  ^
src/helloworld/HelloWorld.java:21: ';' expected
  public void helloCommand(MessageReceiver caller, String[] parameters)
                           ^
src/helloworld/HelloWorld.java:21: ';' expected
  public void helloCommand(MessageReceiver caller, String[] parameters)
                                                             ^
src/helloworld/HelloWorld.java:21: not a statement
```

Just remember that "illegal start" really means "didn't finish properly."

javac: Class Is Public, Should Be Declared in a File Named…

In my HelloWorld.java, I get creative and start to add another class declaration at the end of the file:

```
public class TooMany {
  // ...
}
```

This generates a very descriptive error message:

```
src/helloworld/HelloWorld.java:32: class TooMany is public,
should be declared in a file named TooMany.java
public class TooMany {
```

Remember that every public class must be in a separate file that is named for the class, in a directory named for the package.

However, you *can* declare a private class inside your public class within the same file. This can sometimes be helpful for small helper classes.

javac: Incompatible Types

In the BackCmd plugin version where we're saving player locations to disk, I made a typo: playerTeleports returns a Stack of Location objects. But I accidentally tried to assign it to a stack of Player objects, like this:

```
Stack<Player> locs = playerTeleports.get(player.getName());
```

That results in the fairly straightforward error message "incompatible types":

```
src/backcmdsave/BackCmdSave.java:98: incompatible types
found   : java.util.Stack<net.canarymod.api.world.position.Location>
required: java.util.Stack<net.canarymod.api.entity.living.humanoid.Player>
        Stack<Player> locs = playerTeleports.get(player.getName());
```

Java says it found a Location where it was expecting me to use a Player. Of course that's backwards—that's not what I meant at all.

At times like this it really helps to think like the computer does (in this case, to think like the compiler does). So imagine you're the Java compiler. You just completed reading a line of code through to the semicolon, and you're starting the next line. You see this first part:

```
Stack<Player> locs =
```

Ah! The programmer is declaring a variable named locs, and it's a generic Stack of Player objects. And we're about to assign an initial value to it.

Then the compiler sees the next part of the statement:

```
playerTeleports.get(player.getName());
```

You (the compiler) look up playerTeleports.get() and see that it returns a Stack of Location objects. Well, that won't do at all. Here the programmer says we've got a Stack of Player objects, and now the code is trying to assign a Stack of Location objects. So from the point of view of the compiler, you expected a Player and instead got a Location.

The compiler, of course, is wrong.

That's because no compiler can really infer your *intent*. The compiler can only judge what you've actually done, not what you intended to do.

Always bear that in mind when trying to decipher compiler error messages: they are from the compiler's point of view, and it cannot read your mind. At least not yet. We're working on it.

At any rate, correcting the type of the Stack to match corrects the problem:

```
Stack<Location> locs = playerTeleports.get(player.getName());
```

Canary Server Error Messages

The Canary server will display error messages either in the server log (at ~/Desktop/server/logs/latest.log) or right in the Minecraft game console as you're playing. Here are the most common errors you might see with a new plugin.

Server Log: Plugin Won't Load

Most errors in the log file or in the server's console (in your screen session) are more straightforward than the compiler error messages. One of the most critical and common ones is this error:

```
[SEVERE] Could not load 'plugins/HelloWorld.jar' in folder 'plugins'
net.canarymod.exceptions.InvalidPluginException:
java.lang.ClassNotFoundException: helloworld.HelloWorld
```

You've written a new plugin, and you're getting an error that it can't be loaded.

This usually means the package name and plugin name specified in your Canary.inf file don't match the package name and class name in the code. Check your spelling. Remember that the package name is customarily all lowercase, and that the class name is mixed uppercase and lowercase.

Minecraft Console: Unknown Command

If the plugin compiles okay but Minecraft gives you the "Unknown command" error, be sure to check in your @Command() annotation to make sure you've spelled the command correctly, and if you've specified a min number of arguments or a permission setting, that those are correct as well.

APPENDIX 2

How to Read the Canary Documentation

Many times you'll need to read the Canary documentation directly to find out how to do something in the Minecraft world involving a `Block`, a `Player`, an `Ocelot`, a `Cow`, a `Creeper`, or whatever. You need to find out what classes to use and what functions they offer. The Canary documentation lists this information for you, and we'll take a quick look here at how to find it.

Canary JavaDoc Documentation

The Canary JavaDoc documentation is centered on the system's classes, listed within their packages. Here you'll find the following:

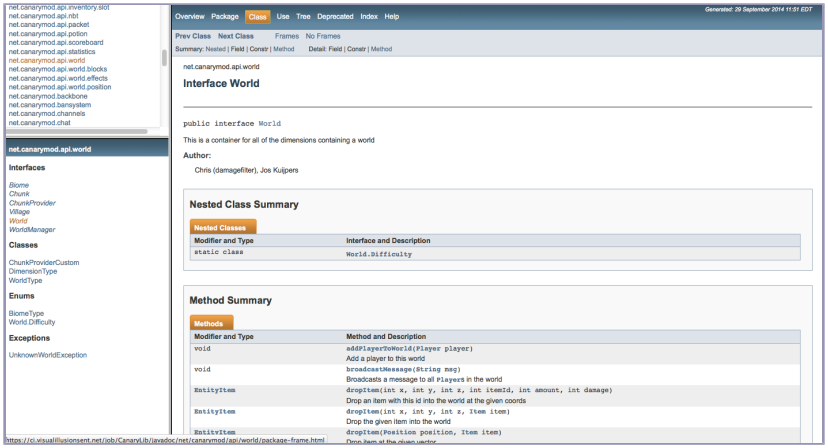
- The package name, which is what you use in the import statement.[1]
- The class name, which is what you use to declare variables and create objects with `new`.[2]
- The functions (methods) in the class that you can run with () and parameters.[3]
- Parent classes or interfaces that it uses, which will include additional functions you can call.[4]

Point your browser to the Canary documentation, and all the hidden treasures of Canary are yours![5] Ah, but what does all this stuff mean? And where is everything?

1. Chapter 2, *Add an Editor and Java*, on page 15, and Appendix 7, *Common Imports*, on page 253.
2. Chapter 4, *Plugins Have Variables, Functions, and Keywords*, on page 43, and Chapter 5, *Plugins Have Objects*, on page 67.
3. Chapter 7, *Use Piles of Variables: Arrays*, on page 89.
4. Chapter 5, *Plugins Have Objects*, on page 67.
5. https://ci.visualillusionsent.net/job/CanaryLib/javadoc

On the left side of your screen, there's a listing of all of the top-level package names in Canary, starting with net.canarymod.

Throughout the doc, you'll see links. Click on a link, and all the children of that element will show up. Click on a package in the upper-left corner, like net.canarymod.api.world, and all the classes and such in that package will show up in the lower-left panel.



Click on World in the lower-left panel, and you'll see all the functions (methods) you can use in a World. Click on any method or class name for more details.

Oracle JavaDoc Documentation

If you're looking for general Java class documentation, not just the Canary-specific classes, point your browser to Oracle's documentation on the Web.[6]

It looks pretty much the same as the documentation on the Canary site (see Figure 6, *JavaDoc for Java*, on page 219).

In the upper-left corner, there's a scrolling list of the packages, including very useful ones like java.util and java.math. Under that is a list of interfaces (like classes, but without their own functions) and the classes in that package. In the right-hand pane is the documentation for the class you've clicked on. Here I've gone to the package java.util, the class ArrayList.

The class documentation is great when you know roughly what you want but need the specifics. But what if you don't know where to start?

6. http://docs.oracle.com/javase/7/docs/api

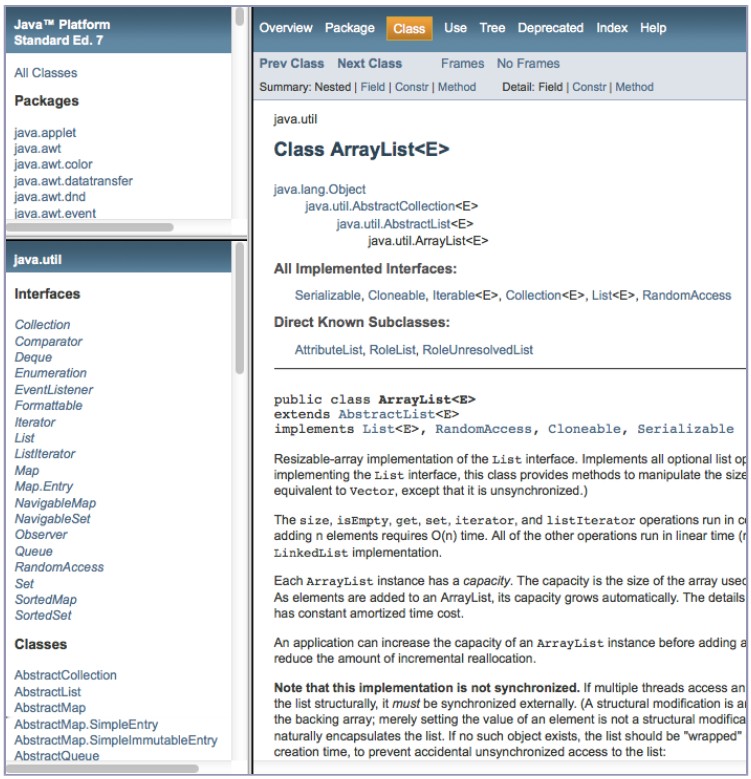


Figure 6—JavaDoc for Java

The Wiki and Tutorials

Hopefully this book has given you a good idea of where to look for specifics about Blocks made of different BlockTypes, and different kinds of Entity objects. But what about something we haven't covered?

In addition to the class-based documentation, the Canary site contains tutorials and detailed explanations of topics on its website, at http://canarymod.net/books.

There's also a forum where you can ask questions and ask for help at http://canarymod.net/forum.

And don't forget about this book's forum at the Pragmatic Bookshelf website, where you can ask questions and post comments about this book itself, along with anything really cool you've discovered that you'd like to pass on.[7]

7. https://forums.pragprog.com/forums/382

APPENDIX 3

How to Install a Desktop Server

So you have your great new custom plugins running on the server on your computer, and you want your friends to be able to connect and play along.

There are two ways to go about that: set up your own personal computer for your friends to access, or set up a permanent server in the cloud. (See how in the next appendix, Appendix 4, *How to Install a Cloud Server*, on page 229.)

To use your own computer, remember that it needs to be powered on and running your Minecraft server for that to work. If you're okay with that idea, here's what you need to do.

There are two main issues to address:

- You need a mechanism for your friends to find your machine's address on the Internet.
- You to need to make your Minecraft port, 25565, open to the world.

There are several ways to go about this. The easiest is to use a piece of software that does all the work for you. You can also take care of all the bits by hand, which is more fun but more work.

The Easy Way: LogMeIn

Hamachi is a product available from LogMeIn in a free version ("Unmanaged") for up to five users,[1] as well as a more feature-rich paid version ("Managed") for Windows and Mac computers.

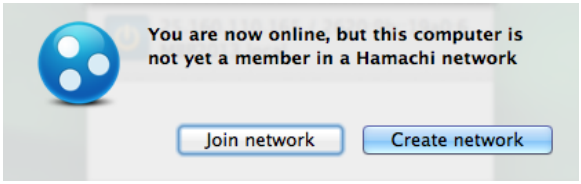
It sets up an IP address for your friends to connect to. You don't need to mess with your firewall or router or anything locally; it just works. To set it up, first download the Unmanaged version and install it.

1. https://secure.logmein.com/products/hamachi/download.aspx

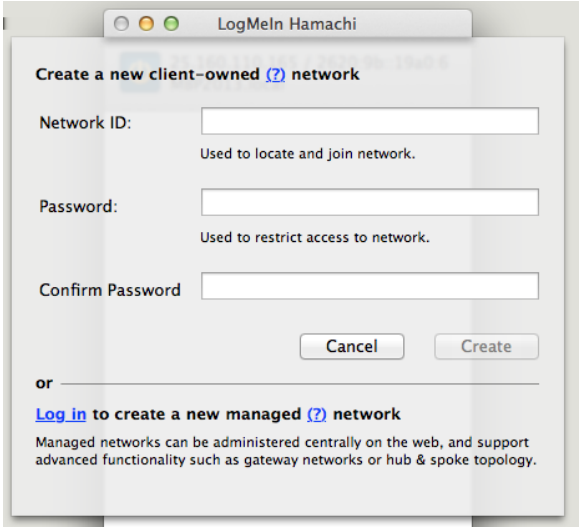
What's a Port?

In this and the next appendix, you'll see references to network ports and Minecraft's default port of 25565. A port is just an agreed-upon number that lets computers communicate over a network. For example, browsers use ports 80 and 443 to talk with web servers.

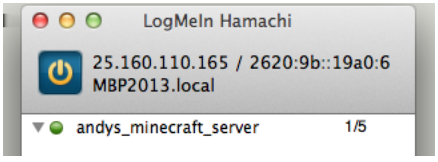
When you first run it, LogMeIn will ask you to join or create a network:



You'll want to create one. I named mine andys_minecraft_server:

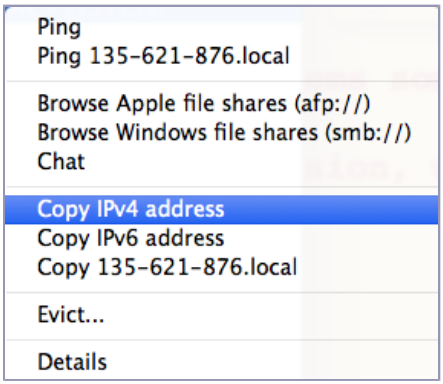


And we're up and running!



Now tell your friends to download the same software and join your network with the name you picked (like andys_minecraft_server).

They can join your network. When they right-click on your network name, they should select Copy IPv4 Address and use that in the Minecraft client to connect to you.



When you don't want your friends connected, just click the power-switch icon to turn your network off.

You're done.

The Harder Way: By Hand

To set up your computer to be open and findable on the Internet without the help of something like LogMeIn isn't magic, but it does take some work. Here are some advantages of doing it the "hard way":

- It's free.
- You can connect more than five users if you want.
- You can be available to the whole world, not just your friends.
- It's really interesting and useful stuff to learn.

You'll need to learn a little bit more about networking and how the Internet works. I'll outline the important steps here, but you may need to do some additional reading on your own. Most importantly, I can't give you exact step-by-step directions because every router (or cable modem, DSL modem, and so on) is different. But you can find that information online with help from Google or via informational sites like http://portforward.com.

Let's look at the steps.

First, your friends need to find your machine.

Static vs. Dynamic DNS

An IP address is a number that anyone in the world can use to connect to your server. An IP address is usually shown as four numbers separated by dots: 93.184.216.119 (example.com) or 127.0.0.1 (your local machine, localhost).

Your ISP probably changes your IP address every so often; you aren't guaranteed to get a the same IP address each day unless you pay extra for a *static IP*. If you knew to ask for a static IP, you probably already know how to set up a domain name and DNS server to point to your machine (if not, and you want a static IP, check out the end of Appendix 4, *How to Install a Cloud Server*, on page 229).

But you don't need a static IP for your friends to find you. There's a trick that lets you use your changing, dynamic IP just as easily.

First off, if you don't know what your IP even looks like, point your browser to this URL: http://www.checkip.org.

And that's what your IP looks like to the outside world.

If that changes every few hours or every few days, that can be annoying, as you have to keep telling your friends what IP address to use to connect to you. But you're in luck! You can get set up with a *dynamic DNS* registration for free or at a low cost from places like http://dyndns.org or http://www.noip.com.

They'll get you set up so that your friends can use a friendly name to find you on the Internet—something like andyminecraft.dns.org.

Now that they can find your machine, you need to let them in.

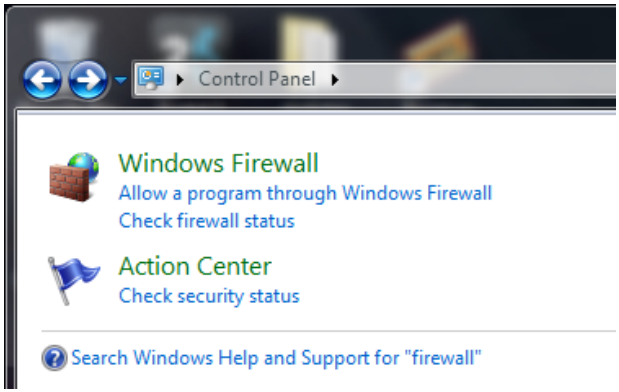
Open Up Your Firewall

Windows, Mac, and Linux machines, as well as your cable modem or ISP's router, can all run firewall software, which is designed to block exactly what we're trying to accomplish here. The idea is that twisted, evil griefers are always trying to attack your machine and break in. So most machines run some kind of firewall to lock out every port except the ones you really want to have open and are expecting.

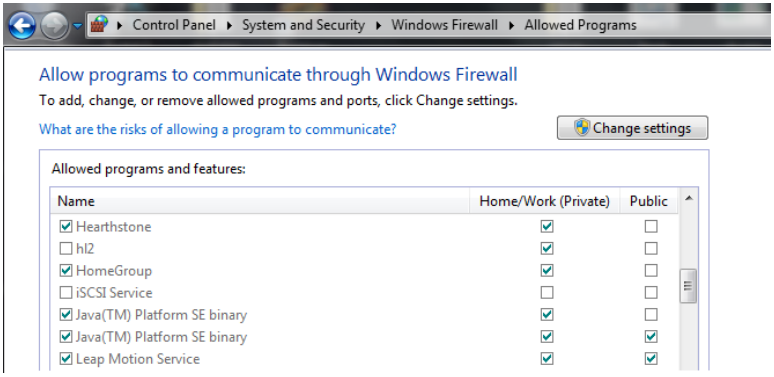
So first you need to make sure *your* computer is open to port 25565. How you do this depends on the operating system you are running.

Open Up Your Firewall on Windows

On Windows 8, start typing Firewall, and Metro will guide you through to "Allow an app or feature through Windows Firewall." See the Windows online documentation for more.[2] On Windows 7, search the Control Panel for "Firewall" and "Allow a program through Windows Firewall":



Then enable public access for Java:

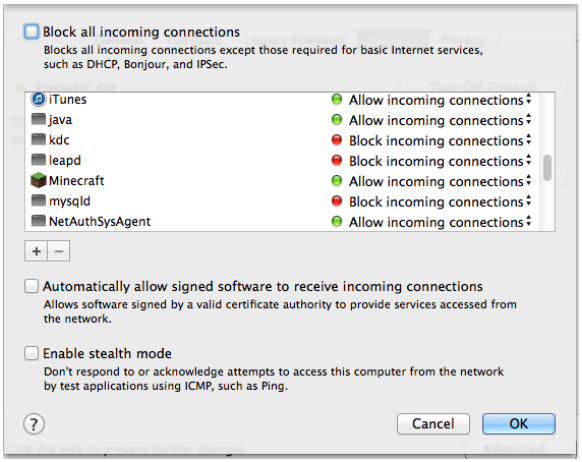


Open Up Your Firewall on Mac OS X

Open System Preferences and go to the Security & Privacy panel:

2. http://windows.microsoft.com/en-us/windows-8/windows-firewall-from-start-to-finish

If the firewall is turned off (not enabled), then you're good to go. If it's turned on, make sure Java has permission to allow incoming connections:



Open Up Your Firewall on Linux

If you've set up a firewall using IPTables, make sure port 25565 is open.

For All Operating Systems

However, opening up the firewall on your computer itself may not be enough.

To check if the Internet can get to your port, point your browser to http://canyouseeme.org. That will come up with your publicly visible IP address. Type in the port to check as 25565, and click Check Port. If it works, you're good to go. If not, there's one more thing you might need to do.

If you are connected to the Internet through a router or cable modem, then *that* device may have a firewall as well. Also, even if it has the port open to the Internet, you need to forward the port traffic from the router to your local computer.

Port Forwarding

You need to exercise caution and restraint when changing settings on your router. If you change something you didn't intend to, you could end up mucking up the device and losing your Internet access. So stick to the plan here. However, every router is different, so I can't really give you specific step-by-step instructions. Here are the basics:

1. Log on to your router. You may already know the user name and password for it. If not, it may have a default user name and password that you can Google for.

2. On some menu it will have a section for *Port Forwarding*. On my router it was listed under "Gateway" and "Forwarding."

3. You need to tell it to forward port 25565 (both TCP and UDP if it asks) from the outside world (your external IP, as reported by http://www.checkip.org) to your computer's *internal IP*.

Your computer has an internal IP address, which is used for communication between it and your router/cable modem. You can run a command on your computer to see what that address is:

Windows

Run `ipconfig`, and look for "IPv4 Address" under "Ethernet adapter" or "Wireless LAN."

Mac/Linux

Run `ifconfig`. It's probably listed under "inet" then "en1."

You might be able to Google for more specific information for your router setup, or check sites such as http://portforward.com for additional information.

One last complication: your internal IP address might change every so often, just like your ISP's IP address can change. If you're getting the IP address from a DHCP server on your router, you can also tell it to create a *DHCP reservation* so that you always get the same internal IP address. Or you can just go in and change the port forwarding when (or if) your internal IP address changes.

If you're having friends in only occasionally and don't mind the setup hassle, this is a perfectly fine way to work. But if you want more than a few folks to connect, and you want the server up 24×7, you'll need to set up a server in the cloud, which is detailed in the next appendix, Appendix 4, *How to Install a Cloud Server*, on page 229.

Quality of Service

Many modern routers have a feature where you can specify the desired Quality of Service (often called QoS) for different ports/services. The idea is that you can set a high or low priority on different kinds of network traffic.

You might find that useful when running a local Minecraft server: you can set the Minecraft port to have a very low priority.

Putting the Minecraft port at the lowest setting will ensure that your own traffic gets priority over your friends using your server.

APPENDIX 4

How to Install a Cloud Server

You can run a Minecraft server on your own computer, but there are some drawbacks to doing that. The biggest one is that your computer needs to be powered on and connected to the Internet 24×7. That's a little difficult on a laptop. Also, anything you're doing personally on the computer may slow down the Minecraft server and all your players—and anything the server is doing will slow down the other programs on your computer. So if a lot of folks want to connect to your server and play on it, you might prefer to set up a remote server in the cloud.

Setting up a Minecraft server in the cloud is very similar to setting up a server locally, with one important difference: you don't have physical access to the computer that's running your Minecraft server. That means no keyboard, no screen, no power switch or reboot button. But no worries; you know how to use the command line, and that's all you need.

We'll see what this all means and how to work with it in this appendix.

What Is the Cloud?

Some say that "the cloud" is really just a big rack of computers in Virginia. That's actually not far from the truth, as Amazon and others do maintain large data centers in that area, plus in California and many other spots around the world. "The cloud" is just a bunch of computers somewhere on the Internet.

When people talk about a computers in the cloud, they mean a computer that is accessed only over the Internet, that you might not even know (or care about) the physical location of, and that someone else owns and maintains. As you can see, that's a pretty generic and flexible definition. A huge variety of cloud services is available, with an equally huge range of pricing, reliability,

and what you get for your money. There are two major types of services you might be interested in:

- Minecraft-specific: The service provider does all the work of maintaining and administering the server. The provider will load or provide plugins, possibly handle griefers and attacks, and keep the server running. You don't have to do anything. You may be charged by the number of players online at once instead of being charged for the size of the server.
- Generic: The service provider supplies a running system and a login, but nothing else: you have to install Java and Canary, load and provide plugins, administer the server, reboot it if it gets hung up, and so on. You are usually charged based on the number of CPUs, the amount of RAM, and possibly the amount of network traffic used each month.

While you might want to look at a Minecraft-specific provider, it's a trade-off between ease of use and control. In this case you don't get much control over the server; you might have to ask to get plugins loaded, and the service might load only well-known plugins, not your development version.

Instead, what you probably want is known as a *virtual private server*, or VPS for short. There are tons of VPS providers on the Internet, with different pricing plans and packages, located in different countries with different hardware and features available. Just Google "VPS providers."

A VPS looks like an entire computer to you. You can log on and have all the files and processor to yourself. In reality, you're sharing a big piece of hardware with a bunch of other people. But everyone sees what looks like a whole computer that is all theirs. That means that as an administrator of the system, you can install any software you want. You can add user accounts, reboot, and set up your own domain name (like `example.com` or `andy.pragprog.com`).

The good news is that you can work on the remote server using a command-line shell, just like we've been doing all along, because your remote server is probably running Linux.

Remote Operating Systems

Virtually all VPS services offer one operating system for your server: Linux. Some providers, such as Microsoft with its own Windows Azure cloud service,[1] can offer Windows, and a small number offer Mac OS X. But these are rare and relatively expensive. The vast majority, including Amazon's cloud service,

1. http://www.windowsazure.com

VPS Features

While pricing and features can vary a lot for VPS offerings, here are some commonalities and things to look for:

- Most VPS systems include a web-based front end (one named `cpanel` is very popular) that allows you to reboot the system, add users, and that sort of thing.
- While a fast computer is always good to have, Minecraft is more sensitive to the amount of RAM you have available. You want a package with as much RAM as you can get—at least 2 GB (gigabytes), and often 4 GB or so.
- As I write this, several good providers have systems available for between $20 and $50 a month. That's a lot of money for a hobby, but you can help support the server by asking for donations from your players. Even a few dollars per player can fund your server. However, there might be issues with your government's laws and tax codes once you start taking money, so you should check with someone in your area who knows about such things before you start.

Some providers allow unlimited network traffic ("bandwidth"), and others set a cap. Just as with many cell-phone plans, if you exceed the bandwidth cap the provider will charge you more money.

Some providers may do the same thing with RAM or disk storage: you're allowed a certain amount, and if you exceed that they'll start charging you extra.

When comparing plans, just be sure to add up the extra fees that you might have to pay.

offer some "flavor" of Linux, a Unix-like, largely POSIX-compliant operating system.[2]

But Linux isn't just one thing. Linux is really a bundle of the core operating system plus all the extra bits: the windowing system, utilities, programming languages, and so on. A particular bundle is known as a *distribution*, and each one is a little bit different in terms of what's included, what's not included, how you install software, and how and when it gets updated.

These are some popular distributions, listed more or less in order of popularity:

- CentOS
- Ubuntu
- Fedora
- Debian
- Linux Mint

2. POSIX stands for Portable Operating System Interface, a standard for operating-system compatibility.

Before you panic over all these different bundles, realize one very important and comforting thing: the command-line shell, bash, works the same on each and every one of these distributions. All the commands we've used in this book, such as ls, cp, and cd, all work exactly the same no matter which Linux flavor you choose.

Ubuntu and CentOS seem to be the most common server distributions at the moment. Most VPS services will let you choose which distribution you want from a small selection of offerings.

The biggest difference you'll run into from one distribution to another is their package management: what tool you use to install software packages, and what packages are included by default.

Remote Access

To log in to your remote server's command-line shell, you need to use a set of programs known as SSH (which stands for "secure shell"). ssh is the command you run to connect to the server; it's the client. The server is running sshd (for "ssh daemon"), and that's what you connect to. Your server might already be running an sshd. We'll get to that shortly.

You can use ssh to connect to a server, and its companion program, scp, to copy files to the server (scp works a lot like plain cp).

ssh just needs to know the user name and server name or IP address you're connecting to, which you can specify like an email address, using the "@" sign:

```
$ ssh andy@example.com
Password:
```

Or if you don't have a name yet, you can connect using the IP address:

```
$ ssh andy@93.184.216.119
Password:
```

You may have been given an account name to use already, or you may need to make one using the web interface from your VPS provider.

You use the same sort of notation with scp to copy a file. Here we're copying a local text file named myfile.txt up to the server example.com, logging in as andy, and copying the file to my home directory ("~/").

```
$ scp myfile.txt andy@example.com:~/myfile.txt
Password:
```

(Again, you could also just use the IP, as in andy@93.184.216.119.)

Notice that when you use ssh or scp without doing anything else, it will ask you for your password on the remote machine. Every time. That gets tiresome pretty quickly, so fortunately there's a better way to set up your remote login.

SSH on Windows

Windows users might not have have access to the ssh and ssh-keygen command line programs; instead, you can use a Windows application named "PuTTY."[a] to generate keys and login to remote systems.

a. http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Set Up SSH Keys

This setup takes a couple of steps, but they are pretty straightforward. We'll go into the details next, but the overall steps are as follows:

1. Generate a special set of secret keys, including a "public key."
2. Copy the public key to the remote machine.
3. Make sure the public key is secured on the remote machine.

Once that's done, you can use ssh and scp without having to supply a password. That means you can use ssh and scp from shell scripts—which can be very handy when you need to do a bunch of things to the remote machine.

Here are the steps in detail.

First off, change to your home directory and use ssh-keygen to generate a set of RSA-style keys:[3]

```
$ cd
$ ssh-keygen -t rsa
```

When asked for a passphrase, press Enter—don't input text at this point. That command will make a subdirectory named .ssh under your home directory. Because of the leading dot (".") you won't normally see this directory listed with ls, but ls -a will show it:

```
$ ls -a
. .. .profile .ssh (and a bunch of other stuff)
$ cd .ssh
~/.ssh$ ls
id_rsa          id_rsa.pub      known_hosts
```

3. RSA is a public-key-style cryptographic system, named for its inventors.

Your public key is id_rsa.pub, and you'll need to add that to a file named ~/.ssh/authorized_keys on the server.

To do that, make the .ssh directory on the server first. I'll show the login prompt on the server as Server $ so you can tell which machine I'm on.

```
$ ssh yourname@example.com
Password:
Welcome to My Awesome Minecraft Server
Last login: Mon Feb 16 12:16:57 from xyzzy-plugh
Server$ mkdir .ssh
```

Now back on your computer, copy the id_rsa.pub file (or whatever your .pub file is named) up to the server using scp, putting it in the .ssh directory and renaming it authorized_keys:

```
$ scp id_rsa.pub yourname@example.com:~/.ssh/authorized_keys
```

That will copy the file to your .ssh directory under your home ("~"), and name it authorized_keys. If you want to later, you can add keys from other machines into this file (that's why "keys" is plural). But for now you just have this one entry.

Finally, you need to go back on the server and check and fix the file permissions. The .ssh directory should be readable and listable only by you, and the files inside should be readable by you alone. In most cases, ssh *will not work at all* if the file permissions aren't restricted. It's for your own good.

You can set the file permissions using the chmod command:

```
$ ssh yourname@example.com
Password:
Welcome to My Awesome Minecraft Server
Last login: Mon Feb 16 12:26:13 from xyzzy-plugh
Server$ cd .ssh
Server ~/.ssh$ chmod 700 .
Server ~/.ssh$ chmod 600 authorized_keys
```

Now from your computer, you should be able to ssh or scp without having to specify a password:

```
$ ssh yourname@example.com
Welcome to My Awesome Minecraft Server
Last login: Mon Feb 16 12:32:07 from xyzzy-plugh
Server$
```

And we're in!

Your next question may be, "Swell, but how do I get OUT?" Fair enough.

Changing Your Prompt

The environment variable PS1 sets your prompt string for bash. In that string, you can use \w to print out the current directory. So if you set

```
$ PS1='Server \w\$ '
```

your prompt will come back as

```
Server ~$
```

Stick this in your bash startup file to make the change permanent (as we saw back in Chapter 2, *Add an Editor and Java*, on page 15).

You can log out by pressing Ctrl-d or by typing exit. And now you're back to your local shell.

Admin with Root

On any modern system, an ordinary user doesn't usually have full permissions to do everything on the system. You usually have to type a password to get the authority of an administrator on the system. On Windows machines this account is called Administrator, and on Linux and Mac systems it's called *root*.

While you can log in directly as root with the appropriate password, that's usually frowned upon. It's too easy to make a typo and suddenly blow away half your system. Root is ultimately powerful, and with great power comes great responsibility.

But you still need to be able to execute certain commands as root without logging on as root. You can do that using the sudo command, which lets you *do* as the *super user* root would do. You preface the command you want to run with sudo, and it will prompt you for *your* normal user account password. Get it right, and then the rest of the command will be executed as root. For example, here I'm using sudo to run an adduser command to add the user "fred."

```
Remote $ sudo adduser fred
Password:
User 'fred' added.
```

This way it allows you to have the full power of root, but only in limited, controlled, well-known circumstances, and not just logged in as root.

In fact, if your SSH is set up so that it *does* allow root to log in directly, you should look up how to disable that feature. That way you have to log in using your user account and SSH key, and then provide your password via sudo.

That's the safest way to allow root access for yourself, and not for the millions of hostile attackers who are eyeing your system at this very moment.

Securing Root Access

You might already be set up to use sudo. Log in to your server and try running some simple command (like id, which just reports who you are) using sudo:

```
Remote $ sudo id
[sudo] password for andy:
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),
4(adm),6(disk),10(wheel)
```

It worked! This machine is already set up so that I can sudo. But on a machine that isn't set up, you'll get this scary warning:

```
andy is not in the sudoers file. This incident will be reported.
```

Yikes! Sounds like the cops will come after us. Have no fear: it's only logging the attempt to a log file. All it means is that you have to add your name to the permissions file for sudo.

To do that, log in as root and add the following line to the bottom of the file /etc/sudoers (nano is a convenient editor to use over SSH; we'll cover how to install that and more in the next section):

```
andy    ALL=(ALL)       ALL
```

But use your username, not "andy"—unless your username actually is Andy.

Save the file and try to use sudo again.

Once you can successfully ssh in without a password, and you can use sudo to do things as root, then you can turn off the use of passwords and not allow root to ssh in directly.

Now obviously that sounds a little dangerous, as you could accidentally lock yourself out of the computer. To help prevent that, open one window and log in to your server as root. Leave that window open and leave it alone, then open a second window to start changing settings. In case you mess up and can't log in or sudo for some reason, you have this window still open—as root, the Great and Powerful—as a backup.

In your new window, log in as root and edit the file /etc/ssh/sshd_config. You want to find these two lines and uncomment them, or change them to read no:

```
PasswordAuthentication no
PermitRootLogin no
```

Then you need to restart the SSH daemon. On many systems you can do that by using this:

```
Server $ sudo service sshd restart
```

You might also want to change the default port from 22 to some other number. Since everyone knows that port 22 is SSH, attackers will bombard your server with attacks on that port. You should be safe by turning off root login and passwords, so that you *have* to have a public key to log on, but if you want to move to a different port to avoid any attacks, here's what you do.

In that same config file (/etc/ssh/sshd_config), add or change this line:

```
Port 2345
```

But pick a number other than 2345. You want a number greater than 1024, and preferably a number that does not appear in the file /etc/services. Those are numbers that other services might already be using (including 80 for web traffic).

When using ssh to connect to the server, you'll have to specify the new port number, which you can do with the -p option:

```
$ ssh -p 2345 andy@myexample.com
```

Installing Packages

Your Linux distribution may come with everything you need. Or it might be missing a few parts. To help keep things manageable, Linux breaks up all the utilities and programs into different software *packages*. For example, if you're not doing any work with publishing, you won't need TeX or Ghostscript. If you're not programming in C or C++, you won't need GCC. If you're not on a desktop machine, you probably don't need a window manager like Gnome or KDE.

All of these packages of software can be installed, removed, updated, and listed using a package manager. Every major Linux distribution uses a different package manager. In fact, you could almost say the package manager is what distinguishes one distribution from another, although there is some sharing going on.

Popular package manager commands include yum, rpm, apt-get, and deb.

The details for each of these commands are different, but they all basically do the same thing: download and install a software package for you.

In particular, if you don't have SSH installed yet, you'll need the open-ssh package installed so you can log on to your server in a secure way. On Ubuntu, you'd run a command like this:

```
$ sudo apt-get install openssh-server
```

sudo runs the command as root; apt-get is the package-manager command under Ubuntu, and you're giving it the option to install the package named openssh-server.

On CentOS, you'd run a very similar command, but using yum instead of apt-get.

```
$ sudo yum install openssh-server
```

Similarly, you'll probably need a text editor on the server. You can't really run a full visual editor on the remote server over the Internet,[4] so you'll probably want a simple screen editor like nano:

```
$ sudo apt-get install nano
```

Installing Java

No matter what optional packages you may need or want, at a minimum you'll need Java to run your server. You can install it just like we did earlier in the book, but bear in mind one important detail: you need the official Oracle (formerly Sun) version of Java. The OpenJDK version of Java, which may be available or already installed on your system, is known to cause problems with Minecraft.

You may need to uninstall OpenJDK using your system's package manager. For instance, to remove the 1.6 version on a system that uses the Red Hat Package Manager (RPM), you'd run this:

```
$ rpm -e java-1.6.0-openjdk
```

Now go get the good Java. You'll have to download the installation program on your local computer, then scp it up to your server.

On your local machine, point your browser to http://www.java.com/en/download/manual.jsp and scroll down to the Linux section. Download the regular or 64-bit version (depending on your server) and follow the instructions for your flavor of Linux.

You'll need to set up your start_minescraft script and jars on the server just as we did locally back in Chapter 2, *Add an Editor and Java*, on page 15.

4. Actually you can, using the X Window System (XWindows), but it's fussy to set up and not a very satisfying experience.

Running Remotely

There's a slight hitch with running a script like start_minecraft when you're logged in via ssh. As soon as you log out, the script will be killed. That's not very helpful in a server environment.

Fortunately, you can use a command named screen to keep Java running even though you've logged out.[5] It takes an argument string of -d -m -S followed by a name for your server session. We'll use mcserver to name the session, so the new server-style startup script will look like this:

```
#!/bin/bash
cd "$( dirname "$0" )"
screen -d -m -S mcserver java -Xms1024M -Xmx1024M -jar CanaryMod.jar --noControl
```

(Remember, start_minecraft needs to be executable; be sure to do a chmod +x start_minecraft if you get a "permission denied" error.)

That will start the server off in its own little world, which won't be affected by you logging off. If you want to "attach" to the server so you can issue commands and such, use screen with the -r option and the session name:

```
$ screen -r mcserver
```

And you'll be attached to the server with the usual spew and prompt:

```
16:47:49 [INFO] Done (1.163s)! For help, type "help" or "?"
>
```

To detach from your server session, press Ctrl-a then d. You'll be returned to your original shell, and your server will still be running in the background.

To see what processes you have running, use the ps command:

```
Server ~$ ps
  PID TTY          TIME CMD
  337 ttys000    0:00.09 -bash
  842 ttys020    0:13.78 /usr/bin/java -jar CanaryMod.jar
```

And to see all the processes on a machine, try ps ax. (ps -? will list the options.)

In a pinch, if you need to kill off all Java processes, you can use

```
Server ~$ killall java
```

That asks Java to do some cleanup and exit in a safe and reliable manner. However, if it's stubborn and won't behave, you can use the brutal

5. There's a slightly simpler way to do this using the command nohup (named for "No Hangup"); however, you lose the ability to enter commands to the server easily.

```
Server ~$ killall -9 java
```

which will kill all Java processes dead, right now. Period.

Domain Name

When you first set up your VPS, your VPS provider will tell you the IP address for your server. That's the number anyone in the world can use to connect to your server. An IP address is usually four sets of numbers with periods in between, like 93.184.216.119 (example.com) or 127.0.0.1 (your local machine, localhost).

You can set up a domain name so that folks can find you by a name like pragprog.com or example.com instead of a number like 93.184.216.119. Your VPS provider may even be able to arrange this for you as an optional service, or you can use one of many providers on the Internet.

Typical costs are around $10–20 a year for *domain name registration*. That gives you the right to use your name. You also need someone to run a *DNS hosting* server that says your name corresponds to your IP address. Many registrars offer this service in addition to the name registration itself, or you can use a separate vendor.

To find out if the domain name you want is available, you can use the command-line tool whois. Be careful using any other tools to check for a domain name: squatters and other unsavory sorts have been known to watch for domain name searches on the Web and grab the name before you can, then try to sell it to you for a lot of money.

As with any service on the Internet, check around for reviews and comments. Some big-name DNS providers that advertise heavily during major sporting events have a very bad reputation.

What's Next

These few pages scratch the surface of what can be very deep topics.

There's a lot you can do as a Linux system administrator that we haven't covered, including running tasks at particular times, backing up data, applying system security and update patches, preventing griefers, tuning performance, and setting up your own email server—all kinds of fun.

Whole books are written on these topics, and this isn't one of them. But hopefully this is enough to get you started.

APPENDIX 5

Cheat Sheets

Enthused by reading a chapter, you're sitting down to write some code only to get stuck: how do you do that thing in Java, again? It happens to all of us when learning a new language. One of the first things I do when trying to work in a new language is to find or write a "cheat sheet" to help me remember the details.

Java Language

This is not a complete list of every Java feature—it's just the important bits you might need, shown as examples in most cases.

Literal Data Types

Whole numbers

int i = 7; i = -5;

Numbers with fractional parts

double num = 3.14; num = 0.01; num = -3e15 *(means -3 times 10 to the 15th power)*

True or false

boolean shouldGetUp=true; shouldGetUp=false *(or any Boolean expression that resolves to* true *or* false*)*

String of text characters

String s="Hello";

Array

String[] grades = {"A", "B", "C", "D", "F", "Inc"};

Math Operators

Add

`a + b`

Subtract

`a - b`

Multiply

`a * b`

Divide

`a / b`

Remainder

`a % b` *(for whole numbers—integers—only)*

Specify order of terms

`((a + b) * z)` *(a + b will be added first, then multiplied by z)*

True-or-False Comparison Operators

Equal to

`a == b`

Not equal to

`a != b`

Less than

`a < b`

Greater than

`a > b`

Less than or equal to

`a <= b`

Greater than or equal to

`a >= b`

And

`a && b` *(must be something that's true or false)*

Or

`a || b` *(must be something that's true or false)*

Not

`!a` *(must be something that's true or false)*

Java Language

Declare the package

package name.name.name;, usually using the inverse of your domain name, so a project Wonderful from the guys at pragprog.com would be in the package com.pragprog.wonderful.

Import a class

import net.canarymod.api.entity.living.humanoid.Player tells the Java compiler you want to use this class. You can use a wildcard of "*"—as in inet.canarymod.api.entity.living.humanoid.*;—but you may get more uninvited classes than you wanted.

Declare a class

public class *name* { }.

Declare a function

public return-type name (*argument types and names*), as in public int myFunction().

Declare a block of code

Use braces, { and }.

Declare variables

type name; as in int i; or Stack<Location> myLocations; generics need the specific type(s) added within the angle brackets < and >.

Assign values to variables

a = 10; s = "Bob"; x=3.1415;.

Make decisions

Decide which code to run using an if (the else {} part is optional):

```
if (true or false comparison) { // is true
    doThis();
} else { // is false
    doOtherThing();
}
```

Loops

There are three ways to make a loop around a block of code:

- for-each construct: for(*type variable* : *collection*) Example: for (Player player : playerList) { *block* }
- for loop: for (int i=0; i< *limit*; i++) { *block* }
- while loop: while (*somethingIsTrue*) { *block* }

Create a new object

`Foo thing = new Foo();`, where `Foo` is the name of the class and `thing` is the variable to be assigned to the new object.

Call a function

Use parentheses, as in `getServer()`, which returns a `Server` object for a plugin.

Java Visibility Modifiers

final

Don't let me—or anyone else—change this value once I set it.

static

Keep this data or function around outside any given object.

You can mix and match `final` and `static` as needed.

public

Anyone else can see and use this function or data.

protected

This class and other classes in this package can see and use this function or data.

private

Only this class can see and use this function or data, not subclasses or other classes in the same package.

You use either `public` or `private` per declaration, so this code would create a publicly visible, unchangeable constant outside of any object:

```
public static final int pi = 3.1415;
```

Java Data-Type Conversions

int to double

Assign it: `int now = 72; double temperature = now;`.

double to int

Cast it; the fractional part is discarded: `double when = 15.375; int day = (int)when;`.

String to int

Use `parseInt`: `int foo = Integer.parseInt("365");`.

String to double

Use `parseDouble`: `double foo = Double.parseDouble("3.1415");`.

double to String

Use valueOf: String.valueOf(3.1415);.

int to String

Use valueOf: String.valueOf(72);.

Double to String

(the class Double, not the primitive) Double.toString(3.1415);.

Integer to String

(the class Integer, not the primitive) Integer.toString(72);.

String concatenation will convert automatically, so

```
String s = "Temperature is " + 72 + " degrees."
```

will result in the string

```
"Temperature is 72 degrees."
```

APPENDIX 6

Glossary

annotation
An added command to Java source code that modifies or adds information (such as tagging a function as an event handler).

argument
A value you pass to a function for it to use.

array
A sequential list of values, indexed with an integer offset.

Array
Java class that implements an array.

binary file
A file that contains binary numbers and is not human-readable.

block
A list of code statements within a pair of braces, { and }.

boolean
A logical value that can be equal to only true or false.

cast
To change the interpretation of a value, usually from one object to a particular parent's type.

class
A recipe that tells the compiler how to make an object: what data and functions it should contain.

client
A piece of software that you run, usually with a graphical interface. It connects to a server.

command line
The terminal window where you can type in commands.

compile
To take a text file of human-readable language instructions and convert it into something the computer can run (usually a binary format). `javac` is the Java compiler.

constructor
A function in a class definition that is called when creating a new object. You can use this to set the object's variables and such.

current directory
In a shell, the directory that's current (*ha*, a tautology!)

deploy
To install a resource into a server environment.

DNS
Domain Name Service, the global system that translates a domain name like `example.com` to an IP address like 93.184.216.119.

double
A big floating-point number.

environment variable
Settings used by the shell and application programs.

event
An object that represents some real-world action, such as a mouse click.

exception
An error that interrupts the current function and starts running the top-most enclosing `catch`, or aborts the program if there isn't one.

executable
A file that the computer can run, usually a binary file with low-level machine instructions, but sometimes a text script run by a shell.

file system
The collection and organization of files and folders (directories) on the computer.

final
A Java keyword indicating that this variable can't be changed.

float
A not-so-big floating-point number.

floating point
A number with a decimal point and fractional part, like 1.25.

function
A list of Java instructions, declared with a return type, a name, parameters it takes, and a block of code between { and }.

global variable
A variable that can be used by multiple functions and/or multiple classes.

hash
A list of objects indexed by any kind of object (but usually by a string).

HashMap
Java class that implements a hash.

import
A Java keyword that lets you use a class from another package, such as `java.util.HashMap` (which is the `HashMap` class in the `java.util` package).

inherit
To use something from a parent.

integer
A whole number with no decimal point and no fractions, like 7.

I/O
Input/output: sending and receiving data from somewhere else, such as a file or over the network.

iterator
An object that lets you retrieve one value at a time from some kind of collection (like an `ArrayList` or `HashMap`).

jar
A Java Archive file that contains `.class` and configuration files.

keyword
A word defined by Java as part of the language. You can't change keywords or use their names as variables.

listener
A function that will be called when something interesting happens.

literal
A value you type in directly, like `123`, `true`, or `"Notch"`.

localhost
The computer's network name for itself.

local variable
A variable declared with a block of { and }; can be used only within that block.

long
A really big whole number, with no decimal point.

map
See *HashMap*.

null
A variable that would normally point to an object but isn't pointing to anything is set to the special value `null`.

object
A collection of live variables and functions, built from a class recipe.

object-oriented
Software based on the theory of objects that combine variables and functions into one pile of stuff.

package
A collection of Java classes that belong together.

parameter
A value in a function that has been passed in for it to use.

path
A list of directories that the computer will search to find a command or other resource.

plugin
A compiled piece of code that's added to an already-compiled piece of code.

port
An agreed-upon number that lets computers communicate over a network. For example, the Web uses port 80, and Minecraft uses port 25565.

private
A Java keyword that restricts visibility to the current class.

public
A Java keyword that opens up visibility to all classes.

script
A text file containing shell commands (or another text language, like Ruby or Python).

server
A piece of software that runs in the background, usually on another machine, that can serve multiple client connections. Can also refer to the remote machine.

shadow
A variable with the same name as another variable in its scope is said to *shadow* it. Chaos may ensue.

shell
See *command line*.

source code
Java language statements that you've typed into a file.

static
A variable or function that is not in any particular object.

string
A bunch of human-readable characters held in a variable.

symbol
A bunch of human-readable characters that has special meaning to the Java compiler as part of your program's language.

task
A piece of code running in a thread, with a well-defined purpose.

text file
A file that contains human-readable text characters.

thread
One list of functions, executed in order by the computer. Threads can be interrupted by other threads.

tick
An arbitrary unit of time. A Minecraft server tick is about 1/20 of a second.

variable
A named holder of data. Can be an immediate value, like 15, or can point to an object, like a `Player` or a `Cow`.

void

Nothing. A function that is declared to return `void` won't return any values, and doesn't need a return statement.

VPS

Virtual Private Server, a remote computer you can rent by the month or by the amount of CPU and network traffic you use.

APPENDIX 7

Common Imports

Here is a listing of all of the imported package and class names used in our plugins. If you get an error that a symbol can't be found, you may need to import the full class name. For instance, to use a HashMap you'd need to import java.util.HashMap, and to use a Block you'd need to import net.canarymod.api.world.blocks.Block.

You can always look up the full class name in the Java or Canary docs, but here are the most common ones for your convenience.

```
import com.pragprog.ahmine.ez.EZPlugin;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Stack;
import net.canarymod.BlockIterator;
import net.canarymod.Canary;
import net.canarymod.LineTracer;
import net.canarymod.api.DamageType;
import net.canarymod.api.entity.Entity;
import net.canarymod.api.entity.EntityType;
import net.canarymod.api.entity.living.EntityLiving;
import net.canarymod.api.entity.living.animal.Bat;
import net.canarymod.api.entity.living.animal.Cow;
import net.canarymod.api.entity.living.animal.Squid;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.entity.living.monster.Creeper;
import net.canarymod.api.factory.EntityFactory;
import net.canarymod.api.factory.PotionFactory;
import net.canarymod.api.inventory.ItemType;
import net.canarymod.api.potion.PotionEffect;
import net.canarymod.api.potion.PotionEffectType;
```

```
import net.canarymod.api.world.Chunk;
import net.canarymod.api.world.World;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import net.canarymod.api.world.blocks.Sign;
import net.canarymod.api.world.effects.Particle.Type;
import net.canarymod.api.world.effects.Particle;
import net.canarymod.api.world.effects.SoundEffect;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.position.Vector3D;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.commandsys.*;
import net.canarymod.database.Column.DataType;
import net.canarymod.database.Column;
import net.canarymod.database.DataAccess;
import net.canarymod.database.Database;
import net.canarymod.database.exceptions.*;
import net.canarymod.hook.HookHandler;
import net.canarymod.hook.entity.DamageHook;
import net.canarymod.hook.entity.ProjectileHitHook;
import net.canarymod.hook.player.ItemUseHook;
import net.canarymod.hook.player.TeleportHook;
import net.canarymod.hook.world.ChunkCreatedHook;
import net.canarymod.hook.world.ChunkLoadedHook;
import net.canarymod.hook.world.ChunkUnloadHook;
import net.canarymod.logger.Logman;
import net.canarymod.plugin.Plugin;
import net.canarymod.plugin.PluginListener;
import net.canarymod.tasks.ServerTask;
import net.visualillusionsent.utils.PropertiesFile;
```

Bibliography

[Swi08] Travis Swicegood. *Pragmatic Version Control Using Git*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2008.

[Swi10] Travis Swicegood. *Pragmatic Guide to Git*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2010.

Index

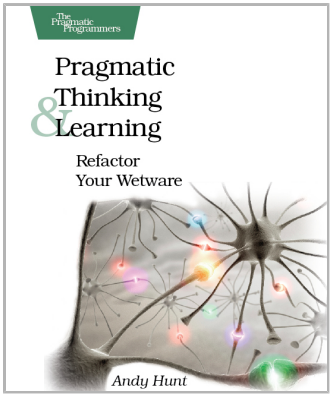
SYMBOLS

A

Other Books by Andy Hunt

Time to rewire your brain for better thinking and learning, and then see what it takes to be a modern software developer.

Pragmatic Thinking and Learning



Software development happens in your head. Not in an editor, IDE, or design tool. You're well educated on how to work with software and hardware, but what about *wetware*—our own brains? Learning new skills and new technology is critical to your career, and it's all in your head.

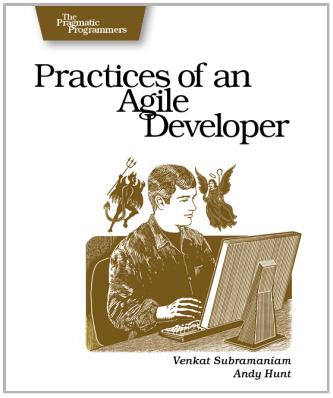
In this book by Andy Hunt, you'll learn how our brains are wired, and how to take advantage of your brain's architecture. You'll learn new tricks and tips to learn more, faster, and retain more of what you learn.

You need a pragmatic approach to thinking and learning. You need to *Refactor Your Wetware.*

Printed in full color.

Andy Hunt
(252 pages) ISBN: 9781934356050. $34.95
http://pragprog.com/book/ahptl

Practices of an Agile Developer



Want to be a better developer? This book collects the personal habits, ideas, and approaches of successful agile software developers and presents them in a series of short, easy-to-digest tips.

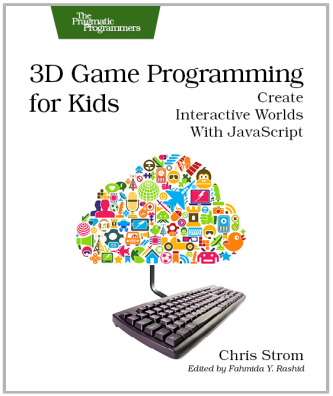
You'll learn how to improve your software development process, see what real agile practices feel like, avoid the common temptations that kill projects, and keep agile practices in balance.

Venkat Subramaniam and Andy Hunt
(208 pages) ISBN: 9780974514086. $29.95
http://pragprog.com/book/pad

JavaScript Games and Sound

Build 3D JavaScript games, and see how to add live sound to your apps.

3D Game Programming for Kids

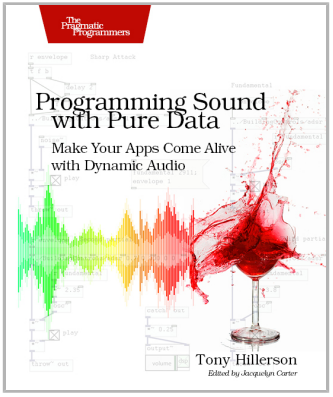


You know what's even better than playing games? Creating your own. Even if you're an absolute beginner, this book will teach you how to make your own online games with interactive examples. You'll learn programming using nothing more than a browser, and see cool, 3D results as you type. You'll learn real-world programming skills in a real programming language: JavaScript, the language of the web. You'll be amazed at what you can do as you build interactive worlds and fun games. Appropriate for ages 10-99!

Printed in full color.

Chris Strom
(250 pages) ISBN: 9781937785444. $36
http://pragprog.com/book/csjava

Programming Sound with Pure Data



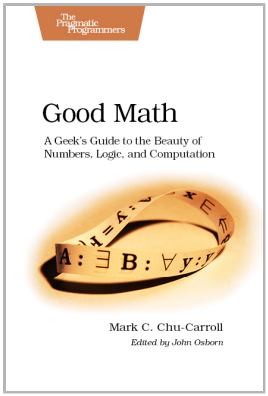
Sound gives your native, web, or mobile apps that extra dimension, and it's essential for games. Rather than using canned samples from a sample library, learn how to build sounds from the ground up and produce them for web projects using the Pure Data programming language. Even better, you'll be able to integrate dynamic sound environments into your native apps or games—sound that reacts to the app, instead of sounding the same every time. Start your journey as a sound designer, and get the power to craft the sound you put into your digital experiences.

Tony Hillerson
(196 pages) ISBN: 9781937785666. $36
http://pragprog.com/book/thsound

The Joy of Math and Health

Rediscover the joy and fascinating weirdness of pure mathematics, and see how to stay healthy as a programmer.

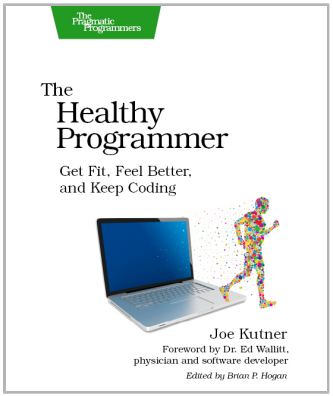
Good Math



Mathematics is beautiful—and it can be fun and exciting as well as practical. *Good Math* is your guide to some of the most intriguing topics from two thousand years of mathematics: from Egyptian fractions to Turing machines; from the real meaning of numbers to proof trees, group symmetry, and mechanical computation. If you've ever wondered what lay beyond the proofs you struggled to complete in high school geometry, or what limits the capabilities of the computer on your desk, this is the book for you.

Mark C. Chu-Carroll
(282 pages) ISBN: 9781937785338. $34
http://pragprog.com/book/mcmath

The Healthy Programmer



To keep doing what you love, you need to maintain your own systems, not just the ones you write code for. Regular exercise and proper nutrition help you learn, remember, concentrate, and be creative—skills critical to doing your job well. Learn how to change your work habits, master exercises that make working at a computer more comfortable, and develop a plan to keep fit, healthy, and sharp for years to come.

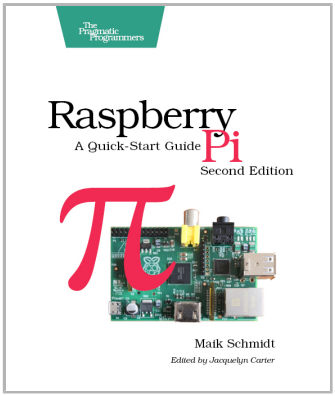
This book is intended only as an informative guide for those wishing to know more about health issues. In no way is this book intended to replace, countermand, or conflict with the advice given to you by your own healthcare provider including Physician, Nurse Practitioner, Physician Assistant, Registered Dietician, and other licensed professionals.

Joe Kutner
(254 pages) ISBN: 9781937785314. $36
http://pragprog.com/book/jkthp

Learn Hardware, Learn Ruby!

Get into the DIY hardware spirit with the Raspberry Pi, or learn to program using Ruby, the language of Ruby on Rails web applications.

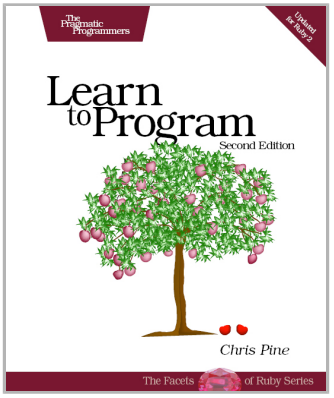
Raspberry Pi: A Quick-Start Guide (2nd edition)



The Raspberry Pi is one of the most successful open source hardware projects ever. For less than $40, you get a full-blown PC, a multimedia center, and a web server—and this book gives you everything you need to get started. You'll learn the basics, progress to controlling the Pi, and then build your own electronics projects. This new edition is revised and updated with two new chapters on adding digital and analog sensors, and creating videos and a burglar alarm with the Pi camera. *Printed in full color.*

Maik Schmidt
(176 pages) ISBN: 9781937785802. $22
http://pragprog.com/book/msraspi2

Learn to Program (2nd edition)



For this new edition of the best-selling *Learn to Program*, Chris Pine has taken a good thing and made it even better. First, he used the feedback from hundreds of reader e-mails to update the content and make it even clearer. Second, he updated the examples in the book to use the latest stable version of Ruby, and also to use code that looks more like real-world Ruby code, so that people who have just learned to program will be more familiar with common Ruby techniques.

Not only does the Second Edition now include answers to all of the exercises, it includes them twice. First you'll find the "how you could do it" answers, using the techniques you've learned up to that point in the book. Next you'll see "how Chris Pine would do it": answers using more advanced Ruby techniques, to whet your appetite as well as providing sort of a "Rosetta Stone" for more elegant solutions.

This fourth printing of *Learn to Program, 2nd edition* has been updated for Ruby 2.0.

Chris Pine
(194 pages) ISBN: 9781934356364. $24.95
http://pragprog.com/book/ltp2

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
http://pragprog.com/book/ahmine2
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *http://pragprog.com/book/ahmine2*

Contact Us

Online Orders:	*http://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764