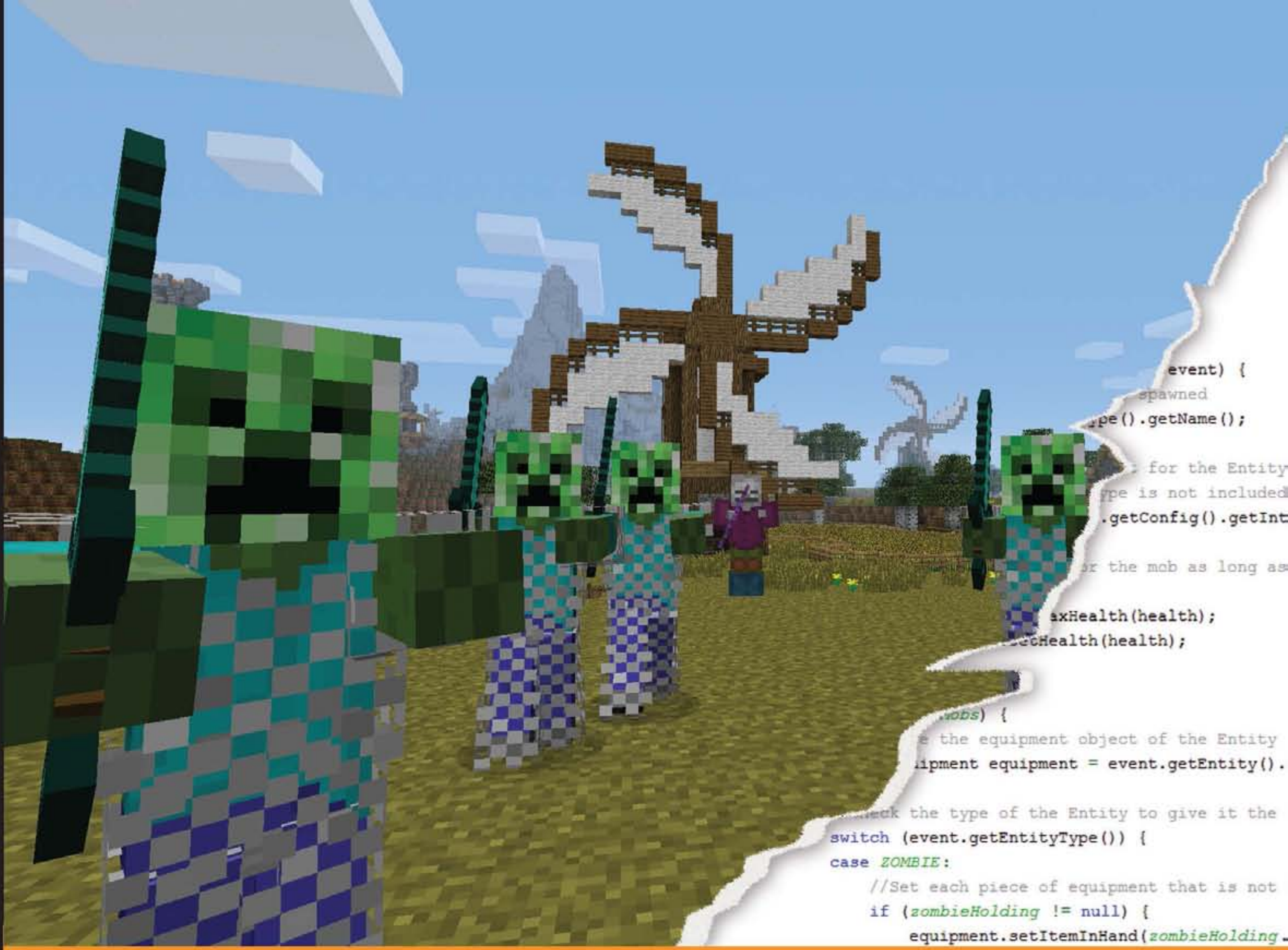

switch (event.getEntityType()) {
case ZOMBIE:
//Set each piece of equipment that is not
if (zombieHolding != null) {
equipment.setItemInHand(zombieHolding

Community Experience Distilled

# Building Minecraft Server Modifications

***Second Edition***

Create and customize your very own Minecraft server using Java and the Spigot API

**Cody M. Sommer**

PACKT PUBLISHING

# Building Minecraft Server Modifications

*Second Edition*

Create and customize your very own Minecraft server using Java and the Spigot API

**Cody M. Sommer**

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

# Building Minecraft Server Modifications

*Second Edition*

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2013

Second edition: December 2015

Production reference: 1151215

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78588-302-6

`www.packtpub.com`

Cover Image by Cody M. Sommer

# Credits

**Author**
Cody M. Sommer

**Reviewers**
Thomas E. Enebo
Pat Patterson

**Commissioning Editor**
Veena Pagare

**Acquisition Editor**
Vinay Argekar

**Content Development Editor**
Divij Kotian

**Technical Editor**
Mohit Hassija

**Copy Editor**
Vedangi Narvekar

**Project Coordinator**
Neha Bhatnagar

**Proofreader**
Safis Editing

**Indexer**
Priya Sane

**Graphics**
Kirk D'Penha

**Production Coordinator**
Shantanu N. Zagade

**Cover Work**
Shantanu N. Zagade

# About the Author

**Cody M. Sommer** graduated from SUNY Brockport with a bachelor's degree in computer science. During his time in college, he practiced his Java techniques by writing code for Bukkit plugins. The Bukkit project provided Cody with a fun way to develop software and continue to learn new things. After a few years, Cody authored the first edition of *Building Minecraft Server Modifications* in order to share his knowledge with the world. To this day, he creates new and exciting plugins for Minecraft servers. Cody has developed public plugins that are available for download, such as PhatLoots. He is also a private developer for servers such as ProspectMC. He even makes custom plugins for Minecraft events, which are hosted by him at his local library.

I would like to thank the CraftBukkit and Spigot community for making the Bukkit API available to developers for several years. They volunteer their time to update and improve the API so that others can use it to create wonderful things.

I would also like to thank my wife and daughter for supporting me while writing this book. Thank you for being patient with me while I spent busy nights typing away in my office. Teaching young people new skills through Minecraft is a passion of mine and I am glad that you understand that. I love you both.

# About the Reviewers

**Thomas E. Enebo** is the co-leader of the JRuby project and author of the Minecraft plugin project, Purugin. He has been a practioner of Java since the heady days of the HotJava browser, and he has happily been using Ruby since 2001. Thomas has spoken at many Java and Ruby conferences, co-authored *Using JRuby*, and won the Ruby Hero award. He was awarded the "Rock Star" award at JavaOne. When Thomas is not coding, he enjoys jogging, reading, and drinking a nice India pale ale (IPA).

**Pat Patterson** has been working with Internet technologies since 1997. He has built software and worked with developer communities at Sun Microsystems, Huawei Technologies, and Salesforce. At Sun, Pat was best known as the community lead for the OpenSSO open source project. At Huawei, he worked on cloud storage infrastructure software.

Since joining the developer evangelism team at Salesforce in late 2010, Pat has worked with all aspects of what is now the Salesforce App Cloud, developing a focus on identity, integration, and the Internet of Things. Describing himself as an "articulate techie", Pat has coded everything from Linux kernel drivers to a Salesforce/Minecraft integration (seriously, you can Google it!), written many online articles, and spoken at conferences on five continents.

www.PacktPub.com

## Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

PACKTLiB™

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

## Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

## Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

# Table of Contents

# Preface

This book is an introduction to programming Minecraft server plugins with the Bukkit API. Minecraft is a very versatile sandbox game, and players are always looking forward to doing more with it. Bukkit allows programmers to do just that. This book is geared towards individuals who may not have a programming background. It explains how to set up a Bukkit server and create your own custom plugins that can be run on the server. It starts with the basic features of a Bukkit plugin, such as commands and permissions, and continues towards advanced concepts, such and saving and loading data. This book will help readers create a complete Bukkit plugin, irrespective of whether they are new to Java or just new to Bukkit. The advanced topics cover portions of the Bukkit API that could even aid the current plugin developers in expanding their knowledge in order to improve their existing plugins.

## What this book covers

*Chapter 1*, *Deploying a Spigot Server*, instructs readers how to download and set up a Minecraft server that runs on Spigot, which includes the forwarding of ports to allow other players to connect. In this chapter, common server settings and commands are explained as well.

*Chapter 2*, *Learning the Bukkit API*, introduces Bukkit by teaching how to read its API documentation. In this chapter, common Java data types and Bukkit classes are discussed.

*Chapter 3*, *Creating Your First Bukkit Plugin*, guides readers through installing an IDE and creating a simple "Hello World" Bukkit plugin.

*Chapter 4*, *Testing on the Spigot Server*, discusses how to install a plugin on a Spigot server as well as simple testing and the debugging of techniques.

*Chapter 5, Plugin Commands*, instructs how to program user commands into a server plugin by creating a plugin called Enchanter.

*Chapter 6, Player Permissions*, teaches how to program permission checks in a plugin by modifying Enchanter. This chapter also guides readers through installing a third-party plugin called CodsPerms.

*Chapter 7, The Bukkit Event System*, teaches how to create complex mods that use event listeners. This chapter also helps readers learn by creating two new plugins, namely NoRain and MobEnhancer.

*Chapter 8, Making Your Plugin Configurable*, teaches readers program configuration by expanding MobEnhancer. This chapter also explains static variables and communication between classes.

*Chapter 9, Saving Your Data*, talks about how to save and load program data through the YAML file configuration. This chapter also helps create a new plugin called Warper.

*Chapter 10, The Bukkit Scheduler*, explores the Bukkit Scheduler while creating a new plugin called AlwaysDay. In this chapter, Warper is also modified to incorporate scheduled programming.

## What you need for this book

In order to benefit from this book, you will need a Minecraft account. The Minecraft game client can be downloaded for free, but an account must be bought at `https://minecraft.net/`. The other software that is used in this book includes the Spigot server.jar (this is different from the normal Minecraft server.jar) and an IDE, such as NetBeans or Eclipse. This book will walk you through the process of downloading and installing both the server and the IDE on a Windows PC.

## Who this book is for

This book is great for anyone who is interested in customizing their Minecraft server. Even if you may be new to programming, Java, Bukkit, or even Minecraft itself, this book has you covered. All that you need is a valid Minecraft account. This book is a great introduction to software development. If you have no prior knowledge of Java or writing software, you can visit `http://codisimus.com/learnjava` for some introductory teaching to prepare yourself for working through the chapters of this book. If you are interested in programming as a career or hobby, this book will get you started. If you are simply interested in playing Minecraft with your friends, then this book will help you make that experience even more enjoyable.

## Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "`<java bin path>` should be replaced with the location of the Java installation. The java path depends on the version of Java that you have on your computer."

A block of code is set as follows:

```
"<java bin path>\java.exe" -jar BuildTools.jar
```

Any command-line input or output is written as follows:

```
>op <player>
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Open the **File** menu and click on **New Project...**."

[ Warnings or important notes appear in a box like this. ]

[ Tips and tricks appear like this. ]

## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

## Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

### Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

### Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/BuildingMinecraftServerModificationsServerSecondEdition_ColorImages.pdf`.

### Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

## Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

## Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

# 1
# Deploying a Spigot Server

The first step to modifying Minecraft with the Bukkit API is to install a **multiplayer server** on your Windows PC. A **multiplayer server** is essentially the same as the single-player Minecraft server, but it allows for more customization and is not limited to the people in your home network. **Spigot** is a modified version of a Minecraft server, which will be used to load the **plugins** that you create. A **plugin** is a piece of software that hooks or plugs into another piece of software. The code that you will develop in this book will be in the form of plugins. These plugins will hook into the Minecraft code and change how Minecraft operates. We will set up a Spigot server and use it to test the plugins that you will write. By the end of this chapter, all of your friends will be able to log on to your modified Minecraft server and play together. By working through the following segments of this chapter, we will deploy a Spigot server, which will be modified in the later chapters:

- An introduction to Spigot
- Installing a Spigot server
- Understanding and modifying the server's settings
- Using the console and in-game Minecraft and Bukkit server commands
- Port forwarding

## Introduction to Spigot

As you set up your own server and begin to create plugins, you will encounter a few terms that may be new to you. The terms are **Vanilla**, **Bukkit**, **CraftBukkit**, and **Spigot**.

**Vanilla** refers to the normal Minecraft game developed by Mojang/Microsoft. The Vanilla Server is the official version of the game. It can be downloaded from `minecraft.net` and is typically named `minecraft_server.jar` or `minecraft_server.exe`. The vanilla server currently does not support any sort of mods or plugins. This is where Bukkit comes in.

**Bukkit** is an API that helps us to develop plugins. We will discuss this in detail in *Chapter 2*, *Learning the Bukkit API*. Until then, it is sufficient to know that when you hear the phrase *bukkit plugins*, it is referring to the plugins that are built against the Bukkit API.

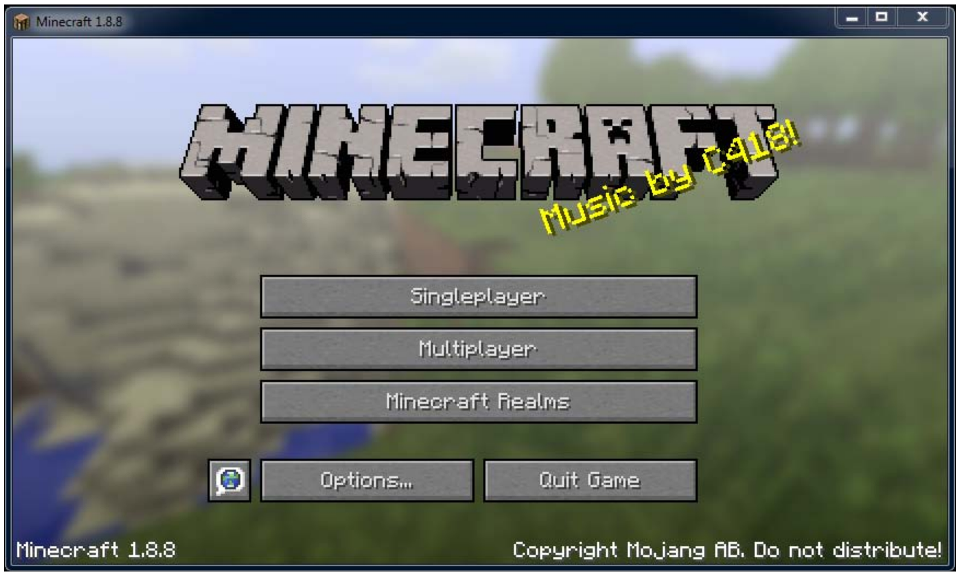
The Bukkit API was originally developed by the **CraftBukkit** team. This brings us to the next term. **CraftBukkit** is a modified Minecraft server that replaces the vanilla server. CraftBukkit and vanilla Minecraft provide us with essentially the same game. The difference is that CraftBukkit has the ability to load Bukkit plugins and execute the code within the game. CraftBukkit translates the Bukkit methods and variables into Minecraft code, which was developed by Mojang. CraftBukkit also includes additional code to aid plugin developers with completing certain tasks, such as saving/loading data, listening for server events, and scheduling the code that needs to be executed. We will not mention CraftBukkit much in this book, because it has been replaced with a project named **Spigot**.

**Spigot** completely replaces the vanilla Minecraft server, just as CraftBukkit does. Spigot was built on top of the CraftBukkit project. Therefore, they share a lot of the same code. However, Spigot is more configurable via its settings; in many ways, it is much faster. The Spigot team now maintains all three of the projects, namely Bukkit, CraftBukkit, and Spigot. You will be able to use either CraftBukkit or Spigot to run a server, since the Spigot team has been kind enough to provide us with both. I recommend running the Spigot server for the reasons mentioned earlier.

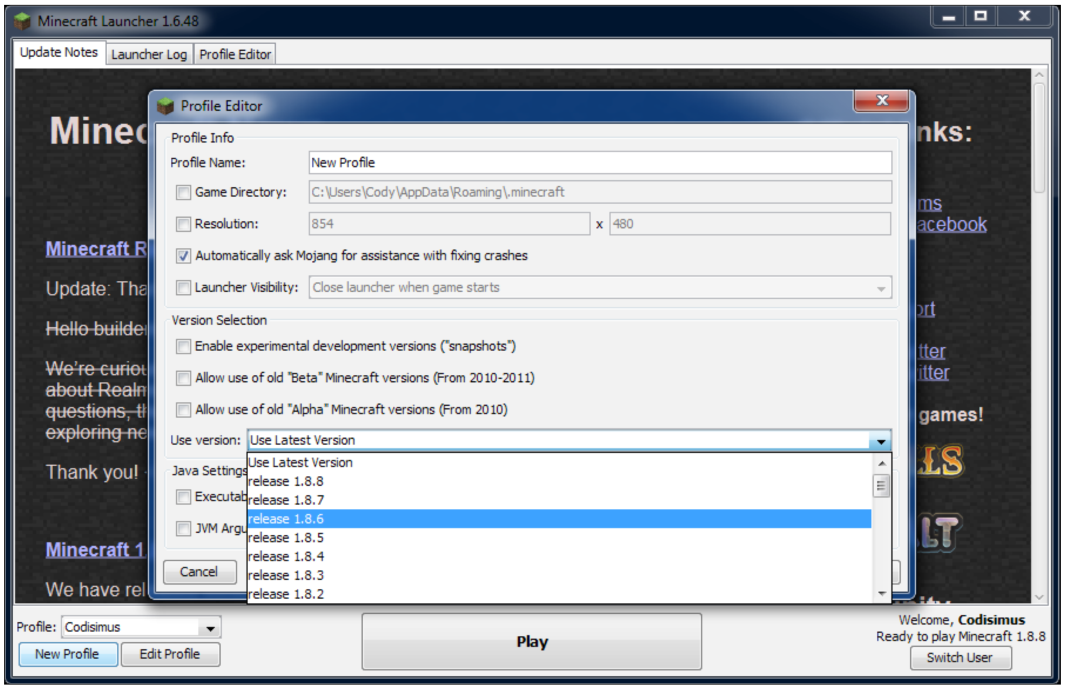
## Installing a new Spigot server

We will start from scratch to set up this new server. If you wish to use a preexisting world, you will be able to do so after creating a new Spigot server. To start, let's create a new empty folder named `Bukkit Server`. We will run the Spigot server from this newly created folder.

The main file that you will need to start a new server is `spigot.jar`. A **JAR** file is an executable Java file. Minecraft, Spigot, and every plugin that we will create are all written in Java and therefore are run from a `JAR` file. The Spigot team updates the `spigot.jar` file as Mojang releases new versions of Minecraft. Typically, when connecting to a Minecraft server, you must be playing on the same version. In case you are unsure of your Minecraft version, it is displayed in the bottom-left corner of the Minecraft **client**. A **client** is the program that you use to play Minecraft, as shown in the following screenshot:



You can choose the version of Minecraft that you want to play by creating a new profile in the Minecraft launcher, as shown in the following screenshot:



Profile Editor

Profile Info

Profile Name: New Profile

Game Directory: C:\Users\Cody\AppData\Roaming\.minecraft

Resolution: 854 x 480

[✓] Automatically ask Mojang for assistance with fixing crashes

Launcher Visibility: Close launcher when game starts

Version Selection

Enable experimental development versions ("snapshots")

Allow use of old "Beta" Minecraft versions (From 2010-2011)

Allow use of old "Alpha" Minecraft versions (From 2010)

Use version: Use Latest Version

Use Latest Version
release 1.8.8
release 1.8.7
release 1.8.6
release 1.8.5
release 1.8.4
release 1.8.3
release 1.8.2

Profile: Codisimus

Welcome, **Codisimus**
Ready to play Minecraft 1.8.8

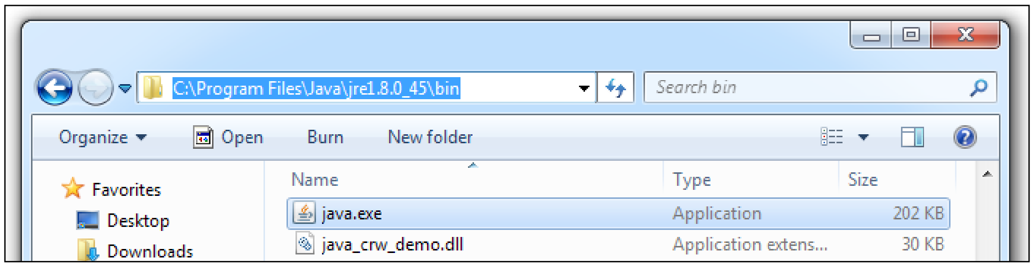
For legal reasons, the Spigot team is not allowed to let you download `spigot.jar`. However, it does provide you with tools and instructions so that you can build the `JAR` file yourself. The Spigot team continues to improve this process by providing the latest instructions as well as a troubleshooting guide at `https://www.spigotmc.org/threads/buildtools-updates-information.42865/`. This chapter includes simplified instructions on how to obtain the needed `JAR` file. However, in case you run into problems while building these `jar` files, refer to the instructions provided on `spigotmc.org`.

You will need Git for Windows in order to run the build tools. You can download it from `http://msysgit.github.io/`. When installing Git for Windows, the default installation settings will be fine. You will also need to download the build tools `JAR` file, which can be found at `https://hub.spigotmc.org/jenkins/job/BuildTools/lastSuccessfulBuild/artifact/target/BuildTools.jar`. Create a new folder to place `BuildTools.jar` in. Name this folder `Spigot Build Tools`. Also create a new text file within the same folder. Name this text file `update.txt`. Open the `text` file and write the following line of code in it:

```
"<java bin path>\java.exe" -jar BuildTools.jar
```

`<java bin path>` should be replaced with the location of the Java installation. The java path depends on the version of Java that you have on your computer. Look for the `Java` folder inside the `Program Files` or `Program Files (x86)` directory. The `Program Files` directory can be found at the root of the main hard drive, which is typically `C:\`. If you do not see the Java folder, then you will have to install Java. You can download it at `https://java.com/download/`.

Once you are in the `Java` folder, you will see one or more java installation folders, such as `jre7` or `jre8`. Open the installation folder. In case you see more than one, open the one with the higher version number. Within the java installation folder, open the `bin` directory. Here you should see `java.exe`, though it may be displayed as simply **java**. Copy the path from the top of the explorer window; this is the java bin path, as shown in the following screenshot:



| Name | Type | Size |
|---|---|---|
| java.exe | Application | 202 KB |
| java_crw_demo.dll | Application extens... | 30 KB |

If you have Java 8, then the line of code in the update file will be similar to the following code:

```
"C:\Program Files (x86)\Java\jre1.8.0_45\bin\java.exe" -jar
BuildTools.jar
```

On most PCs, you can reference the java variable in place of the java.exe path. Therefore the line of code in the update file will be as follows: `java -jar BuildTools.jar`

Save the file and then rename it from `update.txt` to `update.sh`. If you don't see the `.txt` extension on the file, then you will have to adjust your folder options by performing the following steps:

1. Open the **View** tab to the upper left
2. Select **Folder and search options**
3. Uncheck **Hide extensions for known file types**
4. Click on **OK**

Now, you can rename `update.txt` to `update.sh`. Run `update.sh` by double-clicking on it. This will execute the build tools, downloading all the code and applying changes until it is up to date. It will take several minutes to complete. Once it's complete, you will have `spigot.jar`, `craftbukkit.jar`, and `bukkit.jar`. The two server jars, namely spigot and craftbukkit, will be found in the `Spigot Build Tools` directory, where you placed `BuildTools.jar`. The `bukkit.jar file` is located in the `Bukkit/target` directory in the same folder. Each file will have a version number appended to it, such as `spigot-1.8.8.jar` and `bukkit-1.8.8-R0.1-SNAPSHOT.jar`. Note the location of these files, as you will need them within this chapter as well as throughout the book.

It is recommended to run update.sh once a week so that you have the latest version of Spigot.

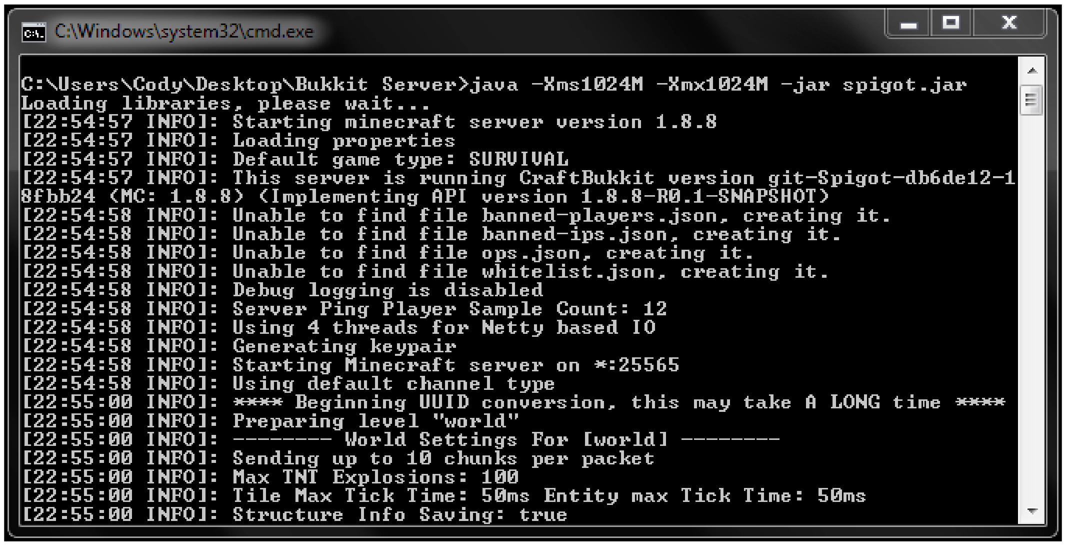
Copy the `spigot.jar` file and place it in the `Bukkit Server` folder that was created by you at the beginning of this segment. For simplicity, we will remove the version number and rename the file `spigot.jar`.

Now, we will create a batch file that we can double-click on every time we wish to start the server. In a new text document, type the following two lines:

```
java -Xms1024M -Xmx1024M -jar spigot.jar
PAUSE
```

`1024` tells how much of the computer's RAM the server will be allowed to use. You can change this number if you want the server to use more or less RAM. `spigot.jar` is the name of the `spigot.jar` file. This must match the name of your file. We renamed the file to exclude the version number so that we will not need to edit this batch file every time we update the Spigot server to the latest version. `java` indicates that we are using Java to run the server. In case the server does not start during the following step, you may need to replace `java` with the full java path that you copied earlier. The rest of the code in the batch file should not concern you, and it should remain unchanged.

Save the text document as `Start Server.bat` and ensure that it is in the same folder as that of `spigot.jar`. Now, you will be able to run the server. Double-click on the batch file that you just created. It will open the command prompt and start creating server files. It will look like the following screenshot and should display the Minecraft server version that you are using:



```
C:\Users\Cody\Desktop\Bukkit Server>java -Xms1024M -Xmx1024M -jar spigot.jar
Loading libraries, please wait...
[22:54:57 INFO]: Starting minecraft server version 1.8.8
[22:54:57 INFO]: Loading properties
[22:54:57 INFO]: Default game type: SURVIVAL
[22:54:57 INFO]: This server is running CraftBukkit version git-Spigot-db6de12-1
8fbb24 (MC: 1.8.8) (Implementing API version 1.8.8-R0.1-SNAPSHOT)
[22:54:58 INFO]: Unable to find file banned-players.json, creating it.
[22:54:58 INFO]: Unable to find file banned-ips.json, creating it.
[22:54:58 INFO]: Unable to find file ops.json, creating it.
[22:54:58 INFO]: Unable to find file whitelist.json, creating it.
[22:54:58 INFO]: Debug logging is disabled
[22:54:58 INFO]: Server Ping Player Sample Count: 12
[22:54:58 INFO]: Using 4 threads for Netty based IO
[22:54:58 INFO]: Generating keypair
[22:54:58 INFO]: Starting Minecraft server on *:25565
[22:54:58 INFO]: Using default channel type
[22:55:00 INFO]: **** Beginning UUID conversion, this may take A LONG time ****
[22:55:00 INFO]: Preparing level "world"
[22:55:00 INFO]: -------- World Settings For [world] --------
[22:55:00 INFO]: Sending up to 10 chunks per packet
[22:55:00 INFO]: Max TNT Explosions: 100
[22:55:00 INFO]: Tile Max Tick Time: 50ms Entity max Tick Time: 50ms
[22:55:00 INFO]: Structure Info Saving: true
```

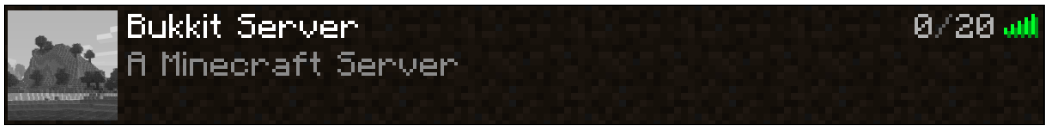
If you do not see the **Starting minecraft server** message, then there may be something wrong with the batch file. If a window similar to what's shown in the previous screenshot does not appear, then make sure that the batch file is named `Start Server.bat` and not `Start Server.bat.txt`. When you first start the server, you will see a few warnings. Most of them shouldn't worry you as they are expected. However, you may see a message that explains that **you need to agree to the EULA in order to run the server**. If you look in the `Bukkit Server` folder, you will now see a new file named `eula.txt`. Open this file, set `eula=true`, and save it to agree to the terms, which are outlined by Mojang at `https://account.mojang.com/documents/minecraft_eula`. Once you do so, you can start the server again. This time, you will see the server loading and generating a new world.

## Setting up a new server

You will see the server folder populated with several files and folders. The purpose of some of these are explained in this section, but most of the files should not concern you at present:

- `plugins`: This folder is where you will place all the Bukkit plugins that you wish to use on the server.
- `world`: The folders that begin with `world`, such as `world`, `world_nether`, and so on, include all the information for the new world of the server. If you already have a Minecraft world that you wish to use, then replace these new folders with the old world folders. Do not attempt to do this while the server is running as it will cause problems.
- `server.properties`: This file holds several options that allow you to change how a Minecraft server operates. You can open it with a text editor. There are many settings, and most of them are pretty self-explanatory. I will go over a few settings in the following list that you may want to modify. For a full list of property explanations, you can visit `www.minecraftwiki.net/wiki/Server.properties`. Changing any of these settings will require you to restart the server.
    - `pvp=true`: The `pvp` property can be set to a `boolean` value. **PvP** (short for **Player versus Player**) determines whether players can attack and harm each other. You will want to set this to `true` or `false`, depending on whether you want PvP to be on or off respectively.
    - `difficulty=1`: The difficulty property can be set to a number from `0` to `3`, where `0` means *Peaceful*, `1` means *Easy*, `2` means *Normal*, and `3` means *Hard*. Everyone on the server will play at this difficulty level.

- `gamemode=0`: This property determines which game mode players will start in, where `0` means *Survival*, `1` means *Creative*, and `2` means *Adventure*.
- `motd=A Minecraft Server`: **MOTD** (short for **Message Of The Day**. This message will be displayed when viewing your server in the Minecraft multiplayer server list, as shown in the following screenshot. It is a good idea to set this to a short description of your server. An example of this is `Bukkit plugin testing`.



- `online-mode=true`: This can be set to `false` to allow players to connect to the server when in the offline mode. This is useful in case `http://minecraft.net/` is unavailable or your computer is not connected to the Internet. Running your server in the offline mode can cause security issues, such as other players logging in to your account.

• `bukkit.yml`: This file contains many server options. These are the options that a vanilla server does not offer and are only available when you run a modified server. Note that this file is a YMAL (`.yml`) file and not a PROPERTIES (`.properties`) file. When you open it, you will see how the two file types are formatted differently. The first difference that you will see is that certain lines are indented. You do not need to fully understand the YMAL formatting, as it will be explained further as we progress through creating Bukkit plugins. There are a few settings in this file that I will bring to your attention, as shown in the following list. For a full list of these Bukkit settings, you can visit `wiki.bukkit.org/Bukkit.yml`. Like `server.properties`, changing any of these settings will require you to restart the server.

- `allow-end: true`: A vanilla Minecraft server allows you to disable the nether world from functioning. A Bukkit server allows you to disable the end world as well. Set this to `false` to prevent players from traveling to the end.

- `use-exact-login-location: false`: Vanilla Minecraft contains a feature that will prevent players from spawning inside a block. The player will instead be spawned above the block so that they will not be stuck when they join the server. This can be easily exploited to climb onto blocks that a player could normally not reach. Bukkit can prevent this from occurring by spawning the player exactly where they logged out. Set this property to `true` if you wish to prevent this.
- `spawn-limits`: Bukkit allows a server admin to modify the number of monsters and animals that are allowed to spawn within a given **chunk**. If you are unfamiliar with the term **chunk**, it is a group of `16 x 16` blocks from bedrock to the highest point of the sky. The following is a picture of a single chunk in Minecraft; if you feel that there are too many (or too few) mobs, then you will want to adjust these values accordingly:

    - `ticks-per: autosave: 0`: Unlike vanilla Minecraft, a Bukkit server will not periodically save your player/world data. Automatically saving data may prevent the server from losing the changes that were made in the game in case it crashes or shuts down for some reason, such as the computer losing power. Vanilla has set this to `6000` by default. This value is provided in **ticks**. There are 20 **ticks** every second. We can determine how long 6,000 ticks is with this math: 6000 ticks / 20 ticks/second = 300 seconds and 300 seconds / 60 seconds/minute = 5 minutes. From this calculation, you should be able to calculate an appropriate time period after which you want your server to autosave your progress. If your server lags whenever it saves your changes, then you may want to increase this number. A setting of `72000` will cause a lag only once every hour. However, if the server crashes right before it saves, you may lose any progress that you made in the past hour.

- `spigot.yml`: This file is similar to `bukkit.yml`. It has many settings and configurations that are only available when running a Spigot server. If you wish to configure any of these options, refer to the documentation at the top of the file.

## Minecraft/Bukkit server commands

We now have all the custom options set. Next, let's log on to the server and take a look at the in-game server commands.

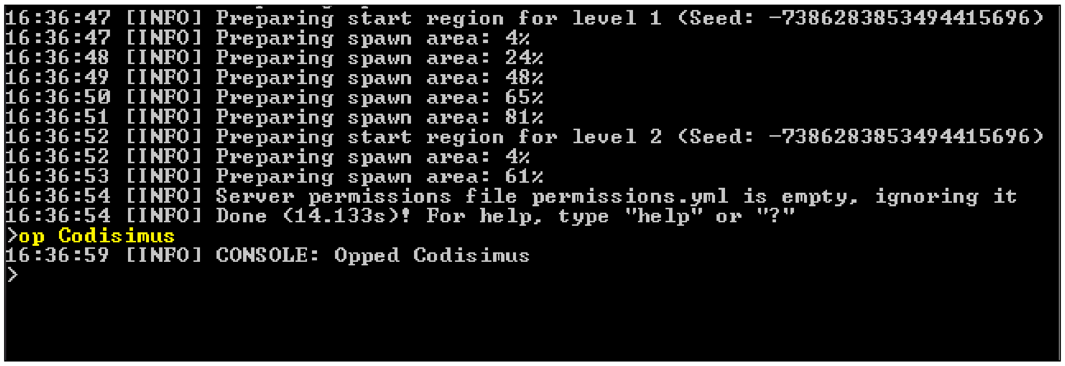
To log in to your server, you will need to know the IP address of your computer. Later in this chapter, we will work through finding this essential information. However, I will assume that for now, you will be playing Minecraft on the same machine on which you are running your server. In this case, for the IP of the server, simply type `localhost`. Once you are connected to the server, you will see that the Spigot server is essentially the same as the vanilla server because you do not have any plugins installed yet. The first indication that the server is running Bukkit is that you will have a few extra commands at your disposal.

Bukkit inherits all the Minecraft server commands. If you have ever played on a Minecraft server, then you have probably already used some of these commands. In case you have not, I will explain some of the useful ones. These commands can be typed into the **console** or an in-game console. By "console", I am referring to the command prompt that is running your server. Bukkit has a built-in permissions system that limits players from using specific commands.

They cannot use a command if they do not have the necessary permissions. We will discuss this in detail in a later chapter, but for now, we will make your player an **operator**, or **op** for short. An **operator** automatically has all the permissions and will be able to execute all the commands that will be presented. To make yourself an operator, issue the `op` command to the console, as follows:

```
>op <player>
```

Replace `<player>` with your player name. See the highlighted command in the following screenshot for an example:



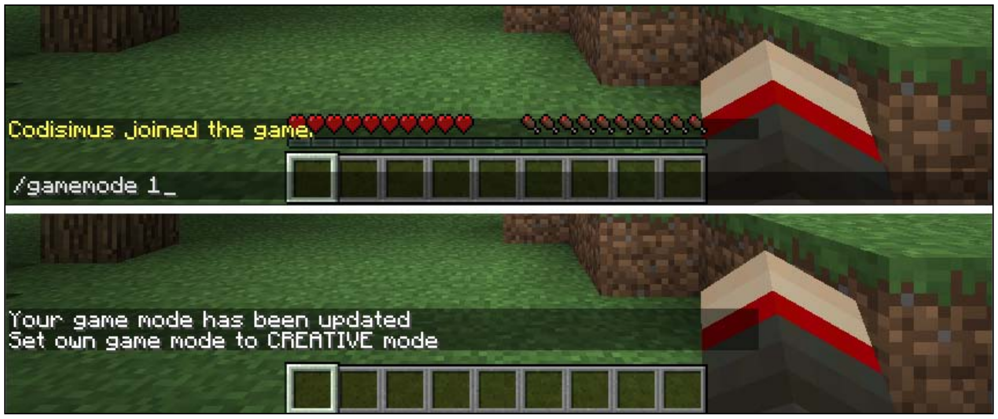
```
16:36:47 [INFO] Preparing start region for level 1 (Seed: -7386283853494415696)
16:36:47 [INFO] Preparing spawn area: 4%
16:36:48 [INFO] Preparing spawn area: 24%
16:36:49 [INFO] Preparing spawn area: 48%
16:36:50 [INFO] Preparing spawn area: 65%
16:36:51 [INFO] Preparing spawn area: 81%
16:36:52 [INFO] Preparing start region for level 2 (Seed: -7386283853494415696)
16:36:52 [INFO] Preparing spawn area: 4%
16:36:53 [INFO] Preparing spawn area: 61%
16:36:54 [INFO] Server permissions file permissions.yml is empty, ignoring it
16:36:54 [INFO] Done (14.133s)! For help, type "help" or "?"
>op Codisimus
16:36:59 [INFO] CONSOLE: Opped Codisimus
>
```

Once you have been *Opped*, you are ready to test some of the in-game server commands. In order to understand how to use commands properly, you must understand the command syntax. Let's look at the `gamemode` command as an example:

```
gamemode <0 | 1 | 2 | 3> [player]
```

- `< >` indicates that it is a required argument.
- `[ ]` indicates that it is an optional parameter. For this command, if the player parameter is not included, then the game mode of your own player will be set.
- | is a known symbol for the word *or*. So, `<0 | 1 | 2 | 3>` indicates that 0, 1, 2, or 3 can be entered. They represent *Survival*, *Creative*, *Adventure*, and *Spectator* respectively.
- Parameters must always be typed in the same order in which they are displayed. Usually, if you enter an incorrect command, a help message will appear, reminding you of how to use the command properly.

Note that when you issue an in-game command, you must start with `/`, but when issuing a command from the console, `/` must be left out. A proper use of the `gamemode` command will be `/gamemode 1`, which will set your game mode to **Creative**, as shown in the following screenshot:



```
Codisimus joined the game.
/gamemode 1_
```

```
Your game mode has been updated
Set own game mode to CREATIVE mode
```

Another example of this command is `/gamemode 2 Steve`, which will find the player whose username is `Steve` and change his game mode to *Adventure*.

Now that you understand the basic syntax for commands, you can learn how to use some other useful server commands from the following list. Most of these commands are also present in vanilla Minecraft. Only a few of them are specific to Bukkit servers:

- `gamerule <rule> [true | false]`

  An example of this is `/gamerule mobGriefing false`.

  The rule parameter can be set to any of the following:

  - `doMobSpawning`: This determines whether mobs will naturally spawn
  - `keepInventory`: This determines whether players will keep their items if they die
  - `mobGriefing`: This determines whether mobs, such as creepers, can destroy blocks
  - `doFireTick`: This determines whether fire should be spread
  - `doMobLoot`: This determines whether mobs should drop items
  - `doDaylightCycle`: This determines whether the day/night cycle is in effect

- `give <player> <item> [amount [data]]`
    - For example, `/give Codisimus wool 3 14` gives Codisimus 3 red wool.
- `plugins` (applicable only in Bukkit)
    - For example, `/plugins` or `/pl` displays a list of all the plugins that are installed on the server.
- `reload` (applicable only in Bukkit)
    - For example, `/reload` or `/rl` disables all the plugins and re-enables them. This command is used to load new settings for a plugin without shutting down the entire server.
- `spawnpoint [player] [x y z]`
    - For example, `/spawnpoint` allows you to spawn where you are standing when you die.
- `stop`
    - For example, `/stop` saves your progress and shuts down the server. This is how you should stop the server to ensure that data is saved. You will lose data if you simply close the command prompt by clicking on **X.**
- `tell <player> <message>`
    - For example, `/tell Steve my secret base is behind the waterfall` sends a message that only Steve will be able to see. Note that these messages will also be printed to the console.
- `time set <day | night>`
    - For example, `/time set day` sets the time of the server to `0` (daytime).
- `toggledownfall`
    - For example, `/toggledownfall` stops or starts rain/snowfall.
- `tp [player] <targetplayer>`
    - For example, `/tp Steve Codisimus` teleports Steve to the location of Codisimus.

For more information regarding these and other commands, visit `minecraft.gamepedia.com/Commands` and `wiki.bukkit.org/CraftBukkit_commands`. The commands and property files mentioned earlier give you a lot of control over how the server functions.

## Port forwarding

Where's the fun in running your own Minecraft server when no one else can log on to it? I will now explain how to allow your friends to connect to your server so that they can play with you. In order to do this, we must first find your IP address. Just like your place of residence has a street address, your computer has an address on the Internet. This is what your friends will type into their Minecraft client to find your server. To find the IP address, simply search `IP` on Google. At the top of the results will be a line that states, "**Your public IP address is XX.XX.XXX.XX**" (the X signs will be replaced by numbers, and its overall length may be different). You can also visit `www.whatismyip.com` to find out your IP address.

Once you have your IP address, try using it to connect to your server rather than using `localhost`. If you are able to connect, then your friends will be able to do so too. If not, you will have to take additional steps to allow other players to connect to your server. This will be the case if your computer is attached to a router. We must let the router know that it should point other Minecraft players towards your computer, which is running the server. This process is called **port forwarding.** To do so, we will first need some additional information.

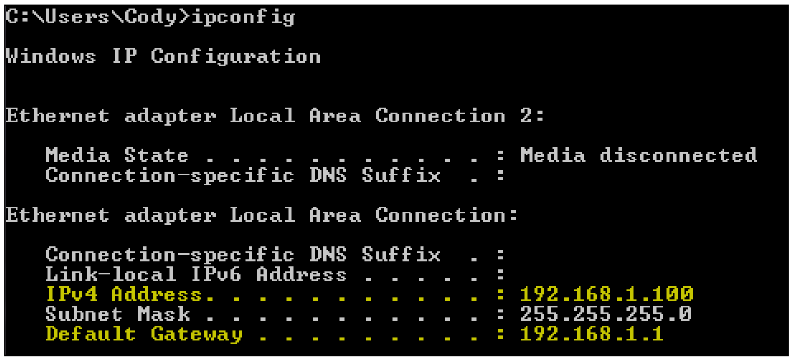
We need to know the IP address of your computer on your local network. This IP address will be different from the address that we obtained earlier. We will also need to know the IP address of your router. To find out this information, open a new command prompt window. The command prompt can be found at the following path:

```
Start Menu/All Programs/Accessories/Command Prompt
```

You can also search for `cmd.exe` to find it. Once the command prompt is open, type in the following command:

```
>ipconfig
```

Then, press the *Enter* key. A screen will be displayed, which will be similar to the one shown in the following screenshot:



```
C:\Users\Cody>ipconfig

Windows IP Configuration


Ethernet adapter Local Area Connection 2:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . :
   IPv4 Address. . . . . . . . . . . : 192.168.1.100
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1
```

In the previous image, the two IP addresses that you were looking for have been highlighted. The numbers will most likely be very similar to these sample numbers. **IPv4 Address** is the address of your computer, and **Default Gateway** is the address of your router. Make a note of both of these IPs.

Next, we will log in to your router. In a web browser, type the IP address of the router, which is **192.168.1.1** in our example. If you do this correctly, then you will be prompted with a login form asking for a username and password. If you do not know this information, you can try to input `admin` for both the fields. If this is unsuccessful, you will have to find the default username and password, which can be found in the paperwork that was provided with your router. This information can usually be found online as well by searching for the name of your router along with the term `default login`.

Once we are logged in to the router, we must find the area that includes the settings for port forwarding. There exist many brands and models of routers in the world, and all of them present this option differently. Therefore, I cannot provide a specific walkthrough of how this page is found. However, you will want to look for a tab that says something that includes the terms **Forwarding**, **Port Forward**, or **Applications & Gaming**. If you do not see any of these, then expand your search by exploring the advanced settings. Once you find the correct page, you will most likely see a table that looks like the following one:

| **Application Name** | **External Port** | **Internal Port** | **Protocol** | **IP Address** |
|---|---|---|---|---|
| Bukkit Server | 25565 | 25565 | TCP and UDP | 192.168.1.100 |

Fill in the fields, as shown in the previous table. The layout and formatting will of course differ depending on your router, but the important details are that you forward port `25565` to the IP address that you found earlier, which is **192.168.1.100** in our example. Ensure that you save these new settings. If you have done this correctly, then you should now be able to connect to the server by using your public IP address.

## Summary

You now have a Spigot server running on your PC. You can inform your friends of your IP address so that they can play on your new server with you. In this chapter, you became familiar with in-game commands and how to use them, and your server is ready to have Bukkit plugins installed onto it as soon as we program them. To prepare ourselves for programming these plugins, we will first become familiar with the Bukkit API and how it can be used.

# 2
# Learning the Bukkit API

In this chapter, you will be introduced to the **Bukkit API** and learn what it allows you to accomplish through programming plugins for a Spigot server. By the end of the chapter, you will most likely have numerous ideas for plugins that you will eventually be able to create yourself. This chapter will cover the following topics in detail:

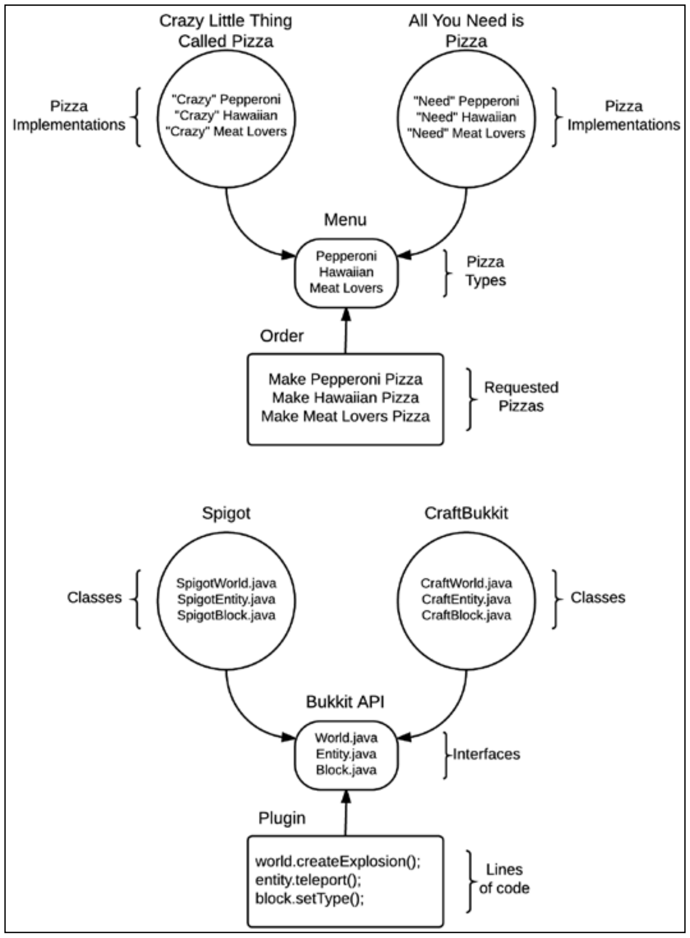
- Understanding the purpose of an API
- Finding the documentation of the Bukkit API
- Navigating through Javadocs to find specific information
- Reading and understanding the documentation
- Exploring and learning the features of the Bukkit API

## Introduction to APIs

**API** stands for **Application Programming Interface**. An API helps control how various software components are used. As mentioned in the previous chapter, Spigot includes the Minecraft code in a form that is easy for developers to utilize when creating plugins. Spigot has a lot of code that we need not access in order to create plugins. It also includes code that we should not tamper with, as it may cause the server to become unstable. Bukkit provides us with the interfaces that we can use to properly modify the game while restricting access to other portions of the code. An **interface** is essentially a shell of a class. It includes methods, but the methods are empty. The Spigot server contains classes for each interface. The classes implement the interfaces and fill each method with the appropriate code.

To explain this better, let's imagine the Bukkit API as a menu to a pizza shop. The menu contains different types of pizza, such as pepperoni, Hawaiian, and meat lovers. These menu items represent the interfaces within the API, with each interface having a method named `makePizza`. At this point, these pizzas cannot be eaten, because they are merely a concept. They are just items on a menu. But let's say that a pizza shop named *All You Need is Pizza* decides to open up and they use this menu, or API. This pizza shop can represent CraftBukkit. The pizza shop creates recipes for each item on the menu. This is equivalent to writing code for each `makePizza` method within the three interfaces. Thus, these recipes are the classes that implement the interfaces. However, these classes are still just a concept. It is not until the `makePizza` method is called that you have an instance of that class. This instance, or object, will be the tangible pizza that you can actually eat. Now, imagine that there is another pizza shop named *Crazy Little Thing Called Pizza*, which opens across the street from *All You Need is Pizza*. This new pizza shop will represent Spigot. Crazy Little Thing Called Pizza uses the exact same menu, or API, as All You Need is Pizza. However, its recipes, or implementations of the methods, may be different.

Using this same analogy, we can see the benefit of the API. As a customer, I can take a look at the menu and assemble an order. For example, I want to order a pepperoni pizza and a meat lovers pizza. Since I created my order based on the menu and both pizza shops implemented the same menu, either restaurant is able to fulfill my order. Likewise, a developer creates a plugin based on the Bukkit API. Both **CraftBukkit** and **Spigot** utilize the **Bukkit API**. Therefore, they will both support the plugin. The following diagram explains this relation between pizza and code:



Basically, Bukkit acts as a bridge between a plugin and the Spigot server. The Spigot team adds new classes, methods, and so on to the API as new features develop in Minecraft, but the preexisting code rarely changes. This ensures that Bukkit plugins will still function correctly months, or even years, from now even though new versions of Minecraft/Spigot are released. For example, if Minecraft were to change how an entity's health is handled, we would not see any difference.

The Spigot jar will account for this change by filling the `getHeath` method with the updated code. Then, when the plugin calls the `getHealth` method, it will function exactly as it had before the update. The addition of new Minecraft features, such as new items, is another example of how great the Bukkit API is. Let's say that we've created a plugin that gives food an expiration date. To check whether an item is food, we'll use the `isEdible` method. Minecraft continues to create new items. If one of these new items is **Pumpkin Bread**, Spigot will flag that type of item as edible and will therefore be given an expiration date by our plugin. A year from now, new food items will still be given expiration dates without us needing to change any of our code.

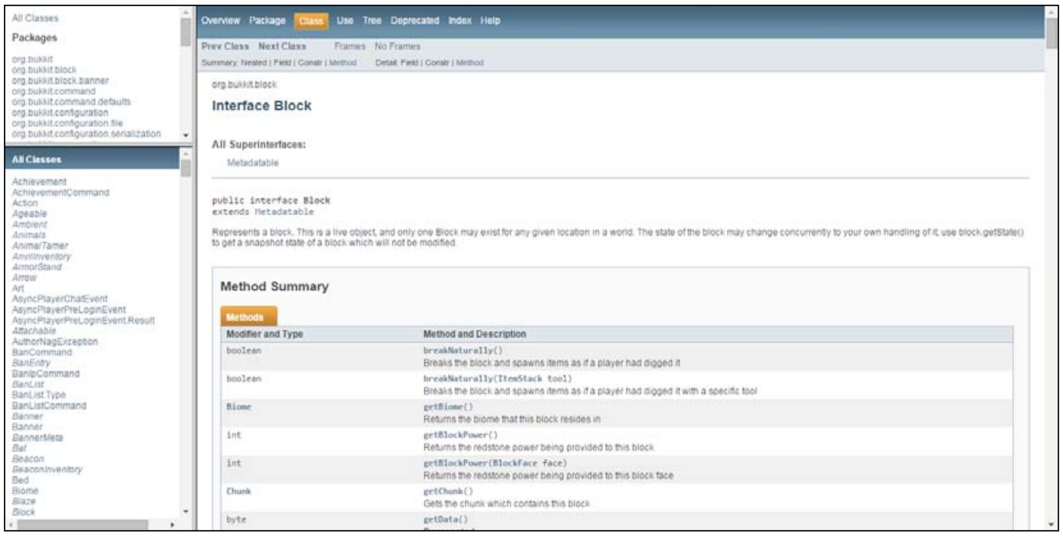
## The Bukkit API documentation

Documentation of the Bukkit API can be found at `hub.spigotmc.org/javadocs/bukkit/`. The Bukkit.jar file that you built in *Chapter 1*, *Deploying a Spigot Server* also contains the Spigot API, which can be found at `hub.spigotmc.org/javadocs/spigot/`. The Spigot API is a **superset** of the Bukkit API, which means that it contains all the classes, interfaces, and so on that are present in the Bukkit API, as well as some additional classes that are unique to the Spigot project. If you want your plugin to support Spigot and CraftBukkit servers, then you will want to develop by using the Bukkit API. If you choose to only support Spigot servers, then you can develop using the Spigot API. In this book, we will refer to the Bukkit API. However, using the Spigot API will yield the same results.

## Navigating through the Bukkit API documentation

We can go through the Bukkit API documentation to get a general idea of what we can modify on a Spigot server. Server-side plugins are different from client-side mods in that we are limited with what we are able to modify in the game using server-side plugins. For example, we cannot create a new type of block, but we can make lava blocks rain from the sky. We cannot make zombies look and sound like dinosaurs, but we can put a zombie on a leash, change its name to Fido, and have it not burn in the daylight. For the most part, you cannot change the visual aspect of the game, but you can change how it functions. This ensures that everyone who connects to the server with a standard Minecraft client will have the same experience.

For some more examples on what we can do, let's have a look at the various pages of the API's documentation:



All Classes

Packages

org.bukkit
org.bukkit.block
org.bukkit.block.banner
org.bukkit.command
org.bukkit.command.defaults
org.bukkit.configuration
org.bukkit.configuration.file
org.bukkit.configuration.serialization

All Classes

Achievement
AchievementCommand
Action
*Ageable*
*Ambient*
*Animals*
*AnimalTamer*
*AnvilInventory*
*ArmorStand*
*Arrow*
Art
AsyncPlayerChatEvent
AsyncPlayerPreLoginEvent
AsyncPlayerPreLoginEvent.Result
*Attachable*
AuthorNagException
BanCommand
*BanEntry*
BanIpCommand
*BanList*
BanList.Type
BanListCommand
*Banner*
Banner
*BannerMeta*
*Bat*
*Beacon*
*BeaconInventory*
Bed
Biome
*Blaze*
*Block*

Overview Package Class Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

org.bukkit.block

Interface Block

All Superinterfaces:

Metadatable

```
public interface Block
extends Metadatable
```

Represents a block. This is a live object, and only one Block may exist for any given location in a world. The state of the block may change concurrently to your own handling of it; use block.getState() to get a snapshot state of a block which will not be modified.

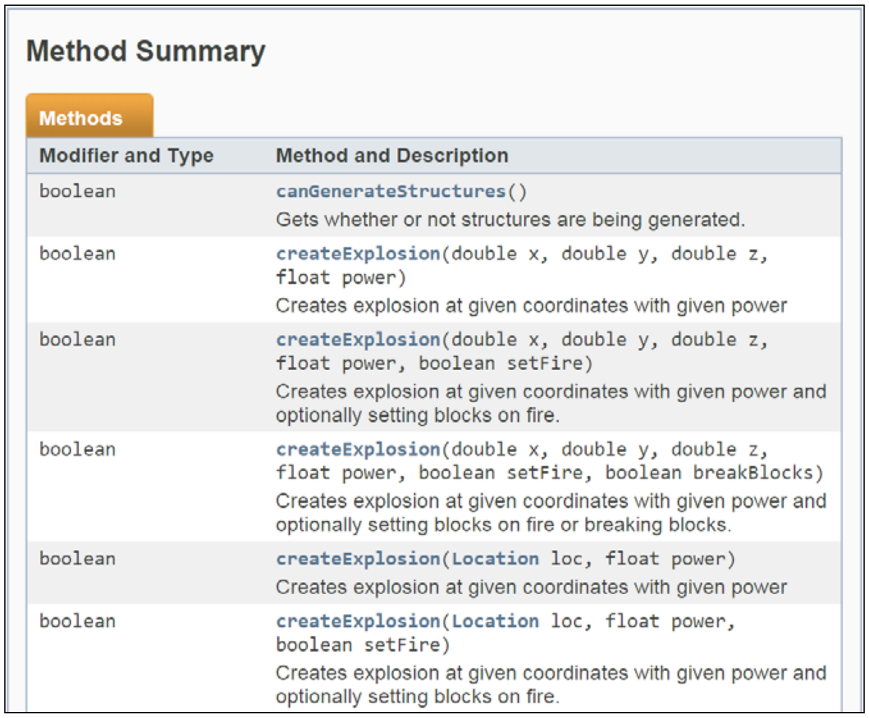
Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| `boolean` | `breakNaturally()`<br>Breaks the block and spawns items as if a player had digged it |
| `boolean` | `breakNaturally(ItemStack tool)`<br>Breaks the block and spawns items as if a player had digged it with a specific tool |
| `Biome` | `getBiome()`<br>Returns the biome that this block resides in |
| `int` | `getBlockPower()`<br>Returns the redstone power being provided to this block |
| `int` | `getBlockPower(BlockFace face)`<br>Returns the redstone power being provided to this block face |
| `Chunk` | `getChunk()`<br>Gets the chunk which contains this block |
| `byte` | `getData()` |

You will see that the classes and interfaces within the API are selectable in the lower left section of the **Javadoc**. Selecting a package in the upper left narrows the choices in the section below it. Each type, such as a class or interface, is organized into a package. These packages help group similar classes together. For example, `Cow`, `Player`, and `Zombie` are all types of entities and thus can be found in the `org.bukkit.entity` package. So, if I were to say that the `World` interface can be found at `org.bukkit.World`, then you will know that you can find `World` within the `org.bukkit` package. Knowing this will help you find the classes or interfaces that you are looking for. You can always use *Ctrl* + *F* to search for a specific word on the webpage. This can help in finding a specific class in a long list.

Let's look at the `World` class and see what it has to offer. The classes are listed in alphabetical order. So, we will find **World** near the end of the list within the `org.bukkit` package. When you click on the `World` class link, all of its methods will be displayed in the main column of the site under the **Method Summary** header, as shown in the following screenshot:



Method Summary

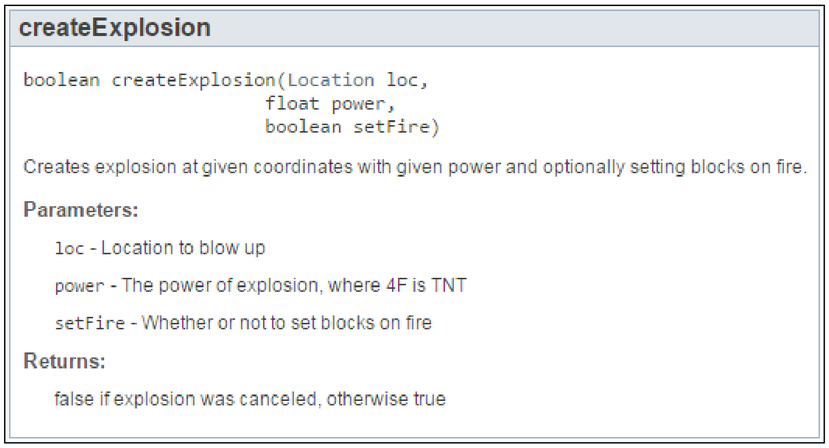
Methods

| Modifier and Type | Method and Description |
|---|---|
| boolean | `canGenerateStructures()`<br>Gets whether or not structures are being generated. |
| boolean | `createExplosion(double x, double y, double z, float power)`<br>Creates explosion at given coordinates with given power |
| boolean | `createExplosion(double x, double y, double z, float power, boolean setFire)`<br>Creates explosion at given coordinates with given power and optionally setting blocks on fire. |
| boolean | `createExplosion(double x, double y, double z, float power, boolean setFire, boolean breakBlocks)`<br>Creates explosion at given coordinates with given power and optionally setting blocks on fire or breaking blocks. |
| boolean | `createExplosion(Location loc, float power)`<br>Creates explosion at given coordinates with given power |
| boolean | `createExplosion(Location loc, float power, boolean setFire)`<br>Creates explosion at given coordinates with given power and optionally setting blocks on fire. |

A `World` object is an entire world on your server. By default, a Minecraft server has multiple worlds, including the main world, nether world, and end world. Spigot even allows you to add additional worlds. The methods that are listed in the `World` class can be applied to the specific world object. For example, the `Bukkit.getWorlds` method will give you a list of all the worlds that are on the server; each one is unique. Therefore, if you call the `getName` method on the first world, it may return `world` while, calling the same method on the second world may return `world_nether`.

## Understanding the Java documentation

Let's look at a method that is included in the `World` class to see what information it provides us. Click on the link to view the `createExplosion(Location loc, float power, boolean setFire)` method. You will be brought to a method description that is similar to the one shown in the following screenshot:



createExplosion

```
boolean createExplosion(Location loc,
                        float power,
                        boolean setFire)
```

Creates explosion at given coordinates with given power and optionally setting blocks on fire.

**Parameters:**

`loc` - Location to blow up

`power` - The power of explosion, where 4F is TNT

`setFire` - Whether or not to set blocks on fire

**Returns:**

false if explosion was canceled, otherwise true

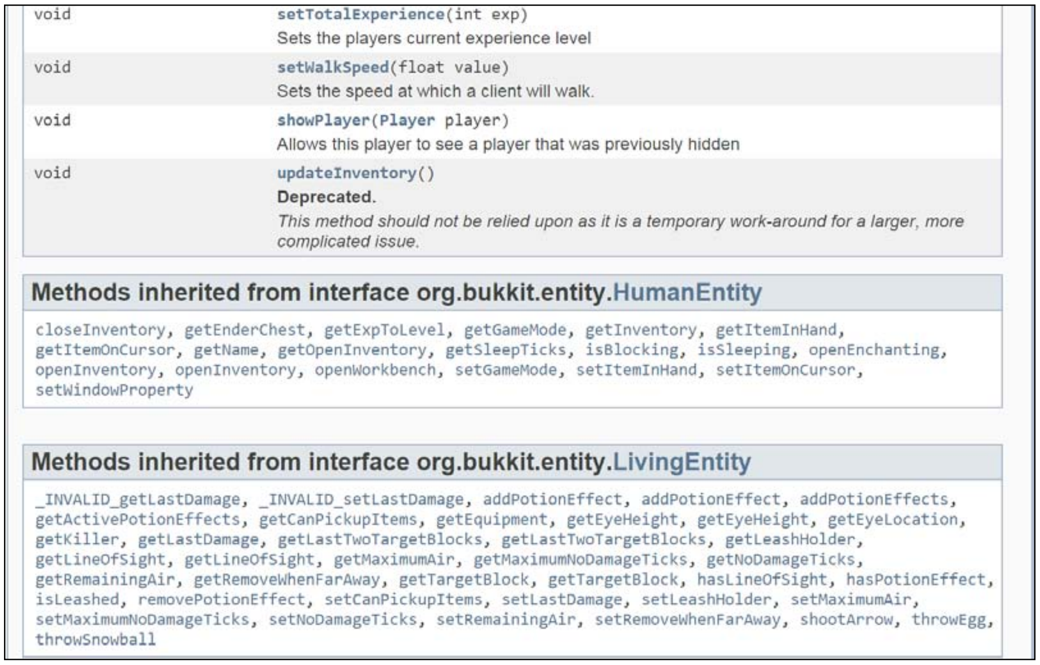
The screenshot explains each parameter and the return value for the method. This method requires that we pass three parameters to it, which is explained as follows:

- Where the explosion should take place
- How powerful the explosion should be
- Whether the explosion should cause the surrounding blocks to ignite in flames

If the returned value is `void`, then the method will not send any information back to us. In this example, the method returns a `boolean` value. On reading the documentation, you will learn that the returned value is whether or not the explosion actually occurred. If another plugin prevented the explosion from happening, then the `createExplosion` method will return `false`.

## Exploring the Bukkit API

Now that you are familiar with the Bukkit API documentation, I advise you to look through it on your own. You will find interesting methods; many of these methods will spark ideas for cool plugins that you may want to make. Note that there may be additional links to view more methods for an object. For example, a `Player` is a type of **LivingEntity**. Therefore, you can call a **LivingEntity** method on a `Player` object. This inheritance is shown after the method summary, as shown in the following screenshot:



| | |
|---|---|
| void | **setTotalExperience**(int exp)<br>Sets the players current experience level |
| void | **setWalkSpeed**(float value)<br>Sets the speed at which a client will walk. |
| void | **showPlayer**(**Player** player)<br>Allows this player to see a player that was previously hidden |
| void | **updateInventory**()<br>**Deprecated.**<br>*This method should not be relied upon as it is a temporary work-around for a larger, more complicated issue.* |

**Methods inherited from interface org.bukkit.entity.HumanEntity**

closeInventory, getEnderChest, getExpToLevel, getGameMode, getInventory, getItemInHand, getItemOnCursor, getName, getOpenInventory, getSleepTicks, isBlocking, isSleeping, openEnchanting, openInventory, openInventory, openWorkbench, setGameMode, setItemInHand, setItemOnCursor, setWindowProperty

**Methods inherited from interface org.bukkit.entity.LivingEntity**

_INVALID_getLastDamage, _INVALID_setLastDamage, addPotionEffect, addPotionEffect, addPotionEffects, getActivePotionEffects, getCanPickupItems, getEquipment, getEyeHeight, getEyeHeight, getEyeLocation, getKiller, getLastDamage, getLastTwoTargetBlocks, getLastTwoTargetBlocks, getLeashHolder, getLineOfSight, getLineOfSight, getMaximumAir, getMaximumNoDamageTicks, getNoDamageTicks, getRemainingAir, getRemoveWhenFarAway, getTargetBlock, getTargetBlock, hasLineOfSight, hasPotionEffect, isLeashed, removePotionEffect, setCanPickupItems, setLastDamage, setLeashHolder, setMaximumAir, setMaximumNoDamageTicks, setNoDamageTicks, setRemainingAir, setRemoveWhenFarAway, shootArrow, throwEgg, throwSnowball

If you are ever going to try and think up an idea for a plugin, browsing through the API documentation will surely give you some ideas. I suggest reading the class pages, listed as follows, as they will be the classes that you will frequently use in your future plugins:

| **Class** | **Package** | **Description** |
|---|---|---|
| `World` | `org.bukkit` | A world on the server |
| `Player` | `org.bukkit.entity` | A person who is playing on the server |
| `Entity` | `org.bukkit.entity` | A player, mob, item, projectile, vehicle, and so on |
| `Block` | `org.bukkit.block` | A specific block in the world, such as a dirt block or a chest |
| `Inventory` | `org.bukkit.inventory` | The inventory of a player, chest, furnace, and so on |
| `ItemStack` | `org.bukkit.inventory` | An item that is in an inventory, which includes how many items are present |
| `Location` | `org.bukkit` | The location of an entity or a block |
| `Material` | `org.bukkit` | The type of a block or item, such as `DIRT`, `STONE`, or `DIAMOND_SWORD` |
| `Bukkit` | `org.bukkit` | Contains many useful methods that can be called from anywhere in your code |

Now that you understand how to read the Bukkit Java documentation, you can find answers to the various questions that you may have. For example, what if you want to find out which methods you would call to get the Block that is at `x:20 y:64 z:14` in the world that is named "world"?

First, you will need to retrieve the correct `World` object. The initial place that you may check is the Bukkit class, as listed in the previous table. You may check there because you can call these methods from anywhere in your code. Another option is to view the uses of the `World` class. This can be done by clicking on the **Use** link at the top of the **World** page. There, you can see all the methods that return a `World` object as well as methods that accept a `World` object as a parameter. To aid in searching through a page, remember that you can use *Ctrl* + *F*. Searching for *name* will lead you to the `Bukkit.getWorld` method, which takes the name of the world as a parameter and returns the actual `World` object.

Once you have the `World` object, you will want to find a method that will give you the `Block` at a specific location. You could navigate to the World page and use *Ctrl* + *F* to search for *block*, *location*, *x*, *y*, or *z*. If none of these help you find a useful method, then you can always view the uses of Block in a way that is similar to how we viewed the uses of World. Either way, you will find the `World.getBlockAt` method, which can be called on the `World` object that you discovered in the previous step.

The following are a few additional challenges to guide you while exploring the Bukkit API on your own and becoming familiar with it:

- Which method would you call to check what time it is in a world?
- Which methods would you call to send a message to the player whose name is Steve?
- Which methods would you call to check whether the material of a block is flammable?
- Which method would you call to check whether a player has diamonds in their inventory?
- Which methods would you call to check whether a player is holding an item that is edible?

## Summary

If you have any trouble figuring out any of the problems mentioned in the challenges or with any other portion of the Bukkit API you can ask help from the Spigot forums (`www.spigotmc.org/forums`), the official IRC channel for Spigot (`www.spigotmc.org/pages/irc`), and the Minecraft forums (`www.minecraftforum.net`).

You can also contact me directly or visit my website at `www.codisimus.com`. I am always interested in helping out a fellow developer.

You now have the knowledge required to begin programming your own Bukkit plugins. As we did in this chapter, we will have to refer to the documentation to find the required information. Being able to navigate and understand the API documentation will speed up the process of coding. If you are ever unsure of a section of the API, you now know how to find the information that you need. In the next chapter, we'll use the Bukkit API to begin writing code to create your first Bukkit plugin.

# 3
# Creating Your First Bukkit Plugin

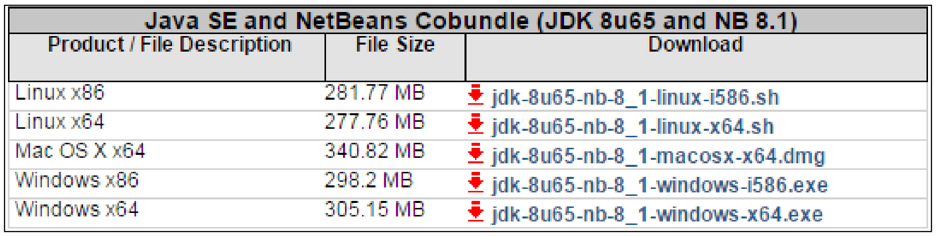
The Bukkit plugins that we will program will be written in the Java programming language. We will use an **IDE** (short for **Integrated Development Environment**) to write the plugins. An IDE is software that will aid us in writing the Java code. It has many tools and features that make programming much easier. For example, it automatically detects errors in code, it often tells us how to fix these errors and sometimes even does it for us, and it provides us with many shortcuts, such as a keystroke to compile code and build a JAR file so that the code can be executed. In this chapter, we will download and install an IDE and prepare it to create a new Bukkit plugin. We will cover the following topics and by the end of this chapter, we will have written our first plugin, which will be ready to be tested on our server:

- Installing an IDE
- Creating a new project
- Adding Bukkit as a library
- The `plugin.yml` file
- The plugin's `main` class
- Creating and calling new methods
- Expanding the code

## Installing an IDE

In this book, we will use NetBeans as our IDE. There are other popular IDEs too, such as Eclipse and IntelliJ IDEA. You can use a different IDE if you wish. However, in this chapter, it will be assumed that you are using NetBeans. No matter which IDE you choose, the Java code will be the same. Therefore, as long as you set up the code properly, you can use any IDE for the remaining chapters. If you are fairly new to programming, then I suggest that you use NetBeans for now, and after you are more comfortable with programming, try other IDEs and choose the one that you prefer.

The NetBeans IDE can be downloaded from `http://www.oracle.com/technetwork/java/javase/downloads/`. Downloading the program from Oracle will also allow us to download the required **Java Development Kit** (**JDK**) at the same time. You will see several download links. Click on the NetBeans link to visit the **JDK 8 with NetBeans** download page. Once you select **Accept License Agreement**, you will be allowed to download the software. The download link is located in a table that is similar to the one shown in the following screenshot:



| Java SE and NetBeans Cobundle (JDK 8u65 and NB 8.1) | | |
|---|---|---|
| Product / File Description | File Size | Download |
| Linux x86 | 281.77 MB | jdk-8u65-nb-8_1-linux-i586.sh |
| Linux x64 | 277.76 MB | jdk-8u65-nb-8_1-linux-x64.sh |
| Mac OS X x64 | 340.82 MB | jdk-8u65-nb-8_1-macosx-x64.dmg |
| Windows x86 | 298.2 MB | jdk-8u65-nb-8_1-windows-i586.exe |
| Windows x64 | 305.15 MB | jdk-8u65-nb-8_1-windows-x64.exe |

If your PC has a 64-bit Windows operating system, then you will want to use the link corresponding to **Windows x64**. If your PC has a 32-bit Windows operating system or you are unsure about whether it's a 64-bit or 32-bit Windows operating system, then download the **Windows x86** version.

If you wish to check whether you are running a 64-bit version of Windows, then you can check it by viewing the **System** window in **Control Panel**.

Once it has finished downloading, install the software. During the installation process, you may be asked about whether you'd like to install **JUnit**. We will not be using **JUnit**. Therefore, you should select **Do not install JUnit**. In the next few screens of the installer, you will be asked about where you would like to install the two types of software. The default settings are fine. You can simply click on **Next**.

## Creating a new project

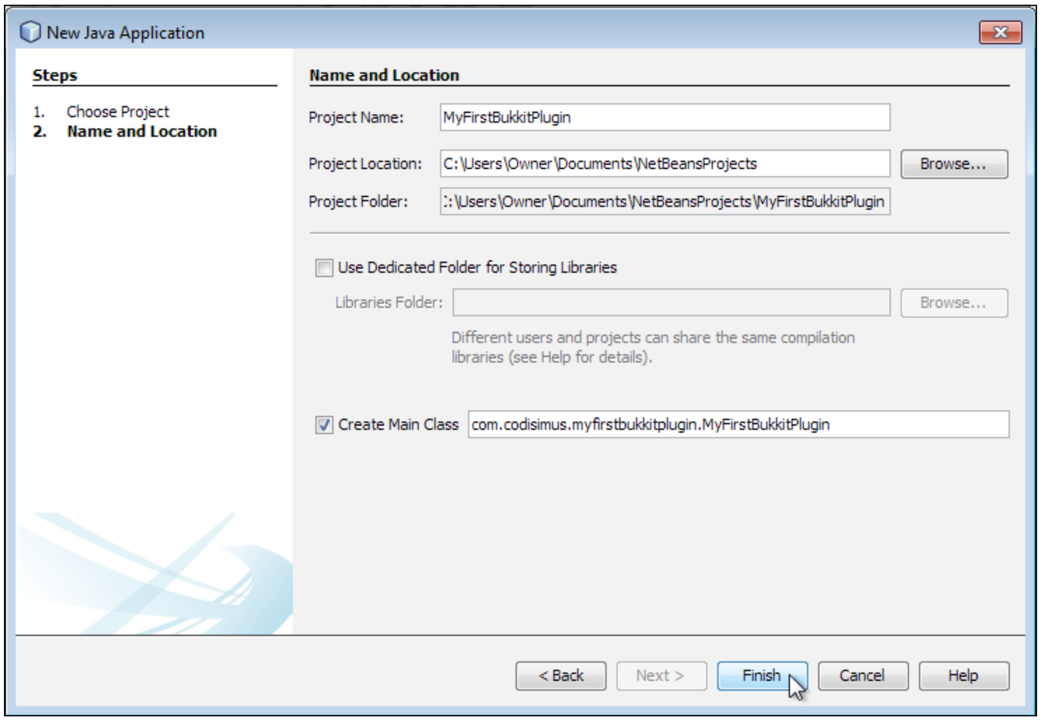
Once it is installed, open NetBeans to begin the creation of the first project. You can create a new project by performing the following steps:

1. Open the **File** menu and click on **New Project...**.
2. We want to create a new **java application. Java Application** is already selected by default. Therefore, simply click on **Next**.
3. We must now name the first project. It is a good idea to avoid using spaces in a name. Let's name this project `MyFirstBukkitPlugin`.
4. Unless you want to store your project in another location, you can leave the default value of **Project Location**.
5. Ensure that **Create Main Class** is checked.

The `main` class is where we will put the code that is needed to enable the plugin that we want to create. For this field, you must determine the package of your project. This usually involves your website's domain name in the reverse order. For example, Bukkit uses `org.bukkit` and I use `com.codisimus`. Assuming that you don't have your own domain name, you can use your e-mail address, such as `com.gmail.username`. You need to use something that will be unique. If two plugins were to have the same package, it might cause collisions in class names, and Java will have no way of knowing which class you are referring to. Using an e-mail address or a domain name that you own is a good way to ensure that other developers don't use the same package. For this same reason, you should exclude `bukkit` or `minecraft` from your package name. The package should also be in lowercase, as shown in the previous examples.

Once you have a package, you need to name your main class. To avoid confusion, most Bukkit plugin developers use the project name as the main class name. The name of the main class should start with a capital letter.

The following screenshot is an example of how your forms should appear before clicking on **Finish**:



New Java Application

Steps
1. Choose Project
2. **Name and Location**

**Name and Location**

Project Name: MyFirstBukkitPlugin

Project Location: C:\Users\Owner\Documents\NetBeansProjects

Project Folder: :\Users\Owner\Documents\NetBeansProjects\MyFirstBukkitPlugin

☐ Use Dedicated Folder for Storing Libraries

Libraries Folder:

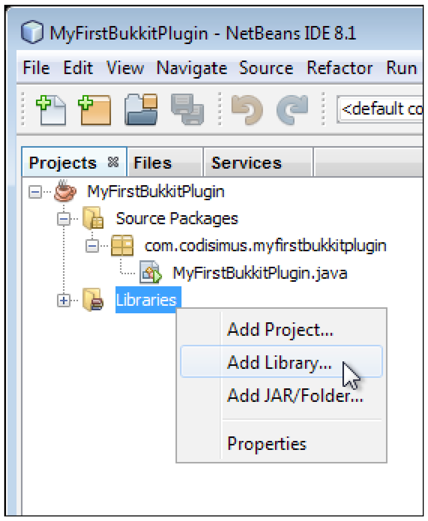
Different users and projects can share the same compilation libraries (see Help for details).

☑ Create Main Class com.codisimus.myfirstbukkitplugin.MyFirstBukkitPlugin

## Adding Bukkit as a library

Now that we have created the main class, we need to add the Bukkit API as the library for the project. You may recall, as discussed in the previous chapter, that the API includes the code that we can access to modify the Spigot server. You built the API JAR file when you built the Spigot jar in *Chapter 1, Deploying a Spigot Server*. If needed, refer to this chapter to retrieve the `Bukkit.jar file`. You will want to move it to a more permanent location. I suggest that you create a folder named `Libraries` and place the JAR file in this folder. The filename will most likely have a version appended to it. We will rename this file, which is similar to what we did for `spigot.jar`. This will help us easily update it in the future. Therefore, the new location of the `bukkit.jar file` will be similar to `C:\Users\Owner\Documents\NetBeansProjects\Libraries\bukkit.jar`. Remember your file location because, now that we have the Bukkit API, we can create a library for it in NetBeans.

In NetBeans, inside the **Projects** tab, you will see a **Libraries** folder. When you right-click on it, you will be presented with the **Add Library...** option. Click on it to bring up a list of the current libraries, as shown in the following screenshot:



For the first project, we need to create the Bukkit library by performing the following steps. For future projects, it will already be present and we can simply select it:

1. Click on **Create...** and type **Bukkit** as the **Library Name**.
2. In the next window, there is an **Add JAR/Folder...** button. Click on it to locate and add the `bukkit.jar` file.
3. Leave the **Sources** tab empty and then click on the **Javadoc** tab.
4. Add `https://hub.spigotmc.org/javadocs/spigot/` in this tab and click on **OK**. This allows us to read some parts of the API documentation directly in the IDE.

Now, you will be able to select **Bukkit** as a library and add it to the project.

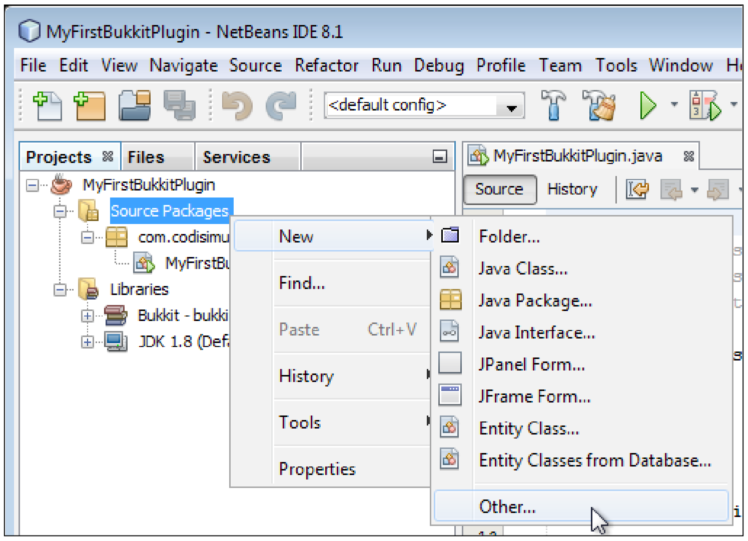
Note that in order to update to a newer version of Bukkit, you can simply replace the current `bukkit.jar` file with the new one, just as you would do to update the `spigot.jar file` on your server. No additional modifications need to be done to your existing projects. However, you will have to compile the code in these projects to check whether any new errors are presented.

## The essentials of a Bukkit plugin

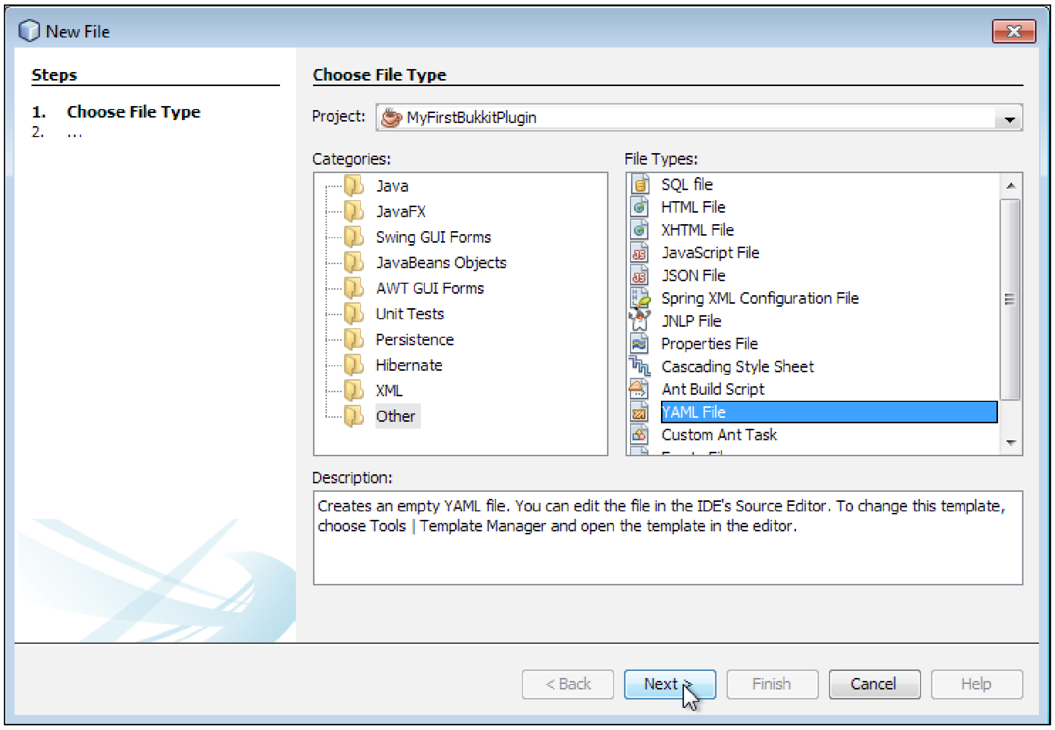
Each Bukkit plugin requires two specific files. These files are `plugin.yml` and the main class of the plugin. We will begin by creating the most basic versions of each of these files. All of your future projects will start with the creation of these two files.

## The plugin.yml file

We are ready to start programming a Bukkit plugin. The first file that we will create is `plugin.yml`. This is the file that the Spigot server will read to determine how to load a plugin. Right-click on **Source Packages** and click on **New | Other...**, as shown in the following screenshot:

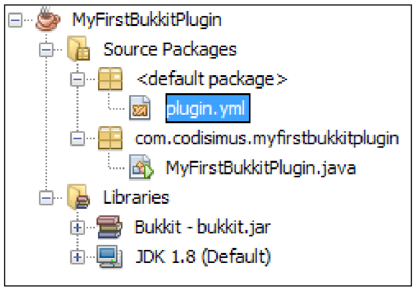


In the window that appears, select **Other** under **Categories**. Then, select **YAML File** under **File Types**, as shown in the following screenshot, and click on **Next**:



Creates an empty YAML file. You can edit the file in the IDE's Source Editor. To change this template, choose Tools | Template Manager and open the template in the editor.

Set the **File Name** as `plugin`, let the name of the folder be `src`, and click on **Finish**. Your project's tree structure should now look like in the following screenshot:



The `plugin.yml` file was created in the default package. This is where it needs to be so that Spigot can find it. For now, we will fill in the `plugin.yml` file with the most basic settings. The `plugin.yml file` must include the name of your plugin, its version, and its `main` class. We have already determined the name and `main` class, and we will give it a `Version number of 0.1`.

If you wish to learn more about version numbers, Wikipedia has a great article on this at `http://en.wikipedia.org/wiki/Software_versioning`.

The simplest form of `plugin.yml` is as follows:

```
name: MyFirstBukkitPlugin
version: 0.1
main: com.codisimus.myfirstbukkitplugin.MyFirstBukkitPlugin
```

That is all that you need in this file, but some other fields that you may wish to add are `author`, `description`, and `website`. We are done with this file. You can save and close `plugin.yml`.

## The plugin's main class

We need to modify the main class. Open `MyFirstBukkitPlugin.java` if it is not already open. We do not use the `main` method in plugins. Hence, we will delete that section of the code. Now, you will have an empty Java class, as shown in the following code:

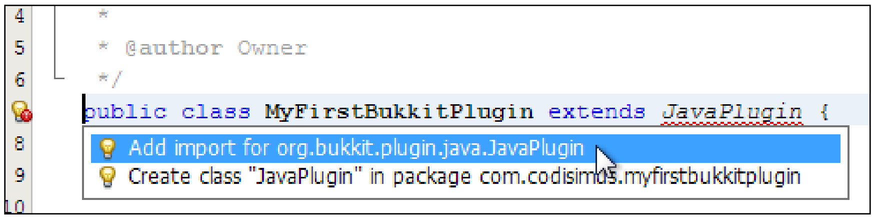
```
package com.codisimus.myfirstbukkitplugin;

/**
 *
 * @author Owner
 */
public class MyFirstBukkitPlugin {

}
```

You may see additional comments, but they will not affect how the program is executed They are there for anyone who may be reading the code to help them understand it. It is always a good idea to comment on the code that you write. If someone ends up reading your code, whether it is a fellow developer or yourself a week from now, they will thank you for adding comments.

The first thing that we need to do is tell the IDE that this class is a Bukkit plugin. To do so, we will extend the `JavaPlugin` class by adding `extends JavaPlugin` immediately following the class name. The modified line will look like the following piece of code:

```
public class MyFirstBukkitPlugin extends JavaPlugin {
```

You will see that a squiggly line and a light bulb appear. This will happen a lot, and it usually means that you need to import something from the Bukkit API. The IDE will do this for you if you ask it to do so. Click on the light bulb and import `JavaPlugin` from the Bukkit library, as shown in the following screenshot:



```
 *
 * @author Owner
 */
public class MyFirstBukkitPlugin extends JavaPlugin {
```

Add import for org.bukkit.plugin.java.JavaPlugin
Create class "JavaPlugin" in package com.codisimus.myfirstbukkitplugin

This will automatically add a line of code near the top of your class. Right now, you can install this plugin on your server, but it will of course not do anything. Let's program the plugin to broadcast a message to the server once it is enabled. This message will show up when the plugin is enabled as we test it. To do this, we will override the `onEnable` method. This method is executed when the plugin is enabled. Mimic the following code to add the method:

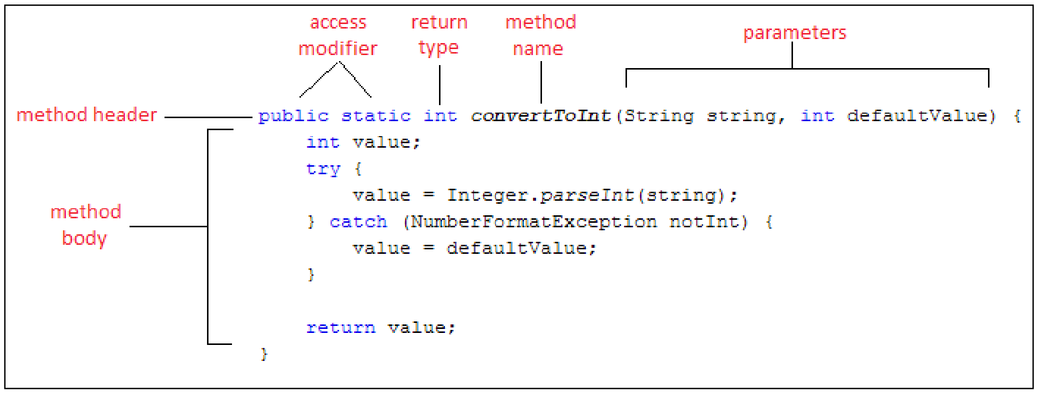
```
public class MyFirstBukkitPlugin extends JavaPlugin {
  public void onEnable() {

  }
}
```

You will see another light bulb that will ask you to add the `@Override` annotation. Click on it to automatically add the line of code. If you were not prompted to add the override annotation, then you may have misspelled something in the method header.

We now have the base of all of your future plugins.

## Making and calling new methods

Let's create a new method that will broadcast a message to the server. The following diagram labels the various parts of a method in case you are not familiar with them:



```
public static int convertToInt(String string, int defaultValue) {
    int value;
    try {
        value = Integer.parseInt(string);
    } catch (NumberFormatException notInt) {
        value = defaultValue;
    }

    return value;
}
```

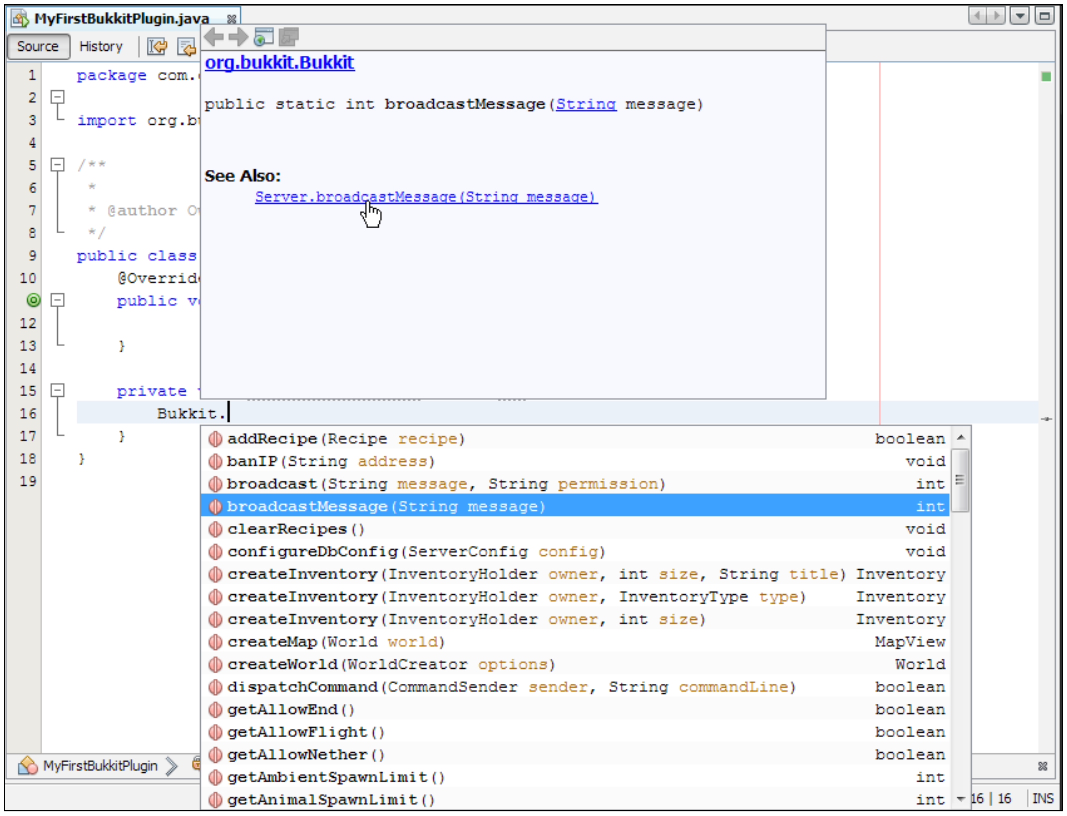
Create a new method named `broadcastToServer`. We will place it in the `MyFirstBukkitPlugin` class under the `onEnable` method. We only want to call this method from inside the `MyFirstBukkitPlugin` class so that the access modifier will be `private`. If you want to call this method from other classes in the plugin, you can remove the modifier or change it to `public`. The method will not return anything and thus will have a return type of `void`. Finally, the method will have one parameter, namely a string named `msg`. After creating this second method, your class will look like the following code:

```
public class MyFirstBukkitPlugin extends JavaPlugin {
  @Override
  public void onEnable() {

  }

  private void broadcastToServer(String msg) {

  }
}
```

**Downloading the example code**

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

We will write the code within the body of the new method to accomplish its task. We want to broadcast a message to the server. We could call the `getServer` method on the plugin. However, for convenience, the `Bukkit` class contains a number of server methods in a static context. You may have come across the methods that we need when you were going through the `Bukkit` class of the API in the previous chapter; if you have not stumbled upon them, browse through the methods in the `Bukkit` class at `https://hub.spigotmc.org/javadocs/spigot/index.html?org/bukkit/Bukkit.html` until you find the `broadcastMessage(String message)` method. We will call the `broadcastMessage` method from our own `broadcastToServer` method. In the IDE, type `Bukkit`, which is the name of the class, to indicate that you will be accessing the `Bukkit` class from a static context. Continue by typing a period (.) in order to call a method from that class. You will see that a list of available methods will appear, and we can simply scroll through them and choose the one that we want. This is shown in the following screenshot:



```
package com.
import org.b
/**
 *
 * @author O
 */
public class
    @Overrid
    public v
    }
    private
        Bukkit.
    }
}
```

**org.bukkit.Bukkit**

```
public static int broadcastMessage(String message)
```

**See Also:**

`Server.broadcastMessage(String message)`

```
addRecipe(Recipe recipe)                                        boolean
banIP(String address)                                              void
broadcast(String message, String permission)                        int
broadcastMessage(String message)                                    int
clearRecipes()                                                     void
configureDbConfig(ServerConfig config)                             void
createInventory(InventoryHolder owner, int size, String title) Inventory
createInventory(InventoryHolder owner, InventoryType type)     Inventory
createInventory(InventoryHolder owner, int size)               Inventory
createMap(World world)                                          MapView
createWorld(WorldCreator options)                                 World
dispatchCommand(CommandSender sender, String commandLine)       boolean
getAllowEnd()                                                   boolean
getAllowFlight()                                                boolean
getAllowNether()                                                boolean
getAmbientSpawnLimit()                                              int
getAnimalSpawnLimit()                                               int
```

Click to select the `broadcastMessage` method. The API documentation for the method will be displayed. Note that to the right of the method, it says **int**. This informs us that this method returns an `integer` type value. When you click on the **See Also:** link, as shown in the preceding screenshot, the documentation will tell us that the number that is returned is the number of players that the message was sent to. We don't really care about this number. Therefore, we will not assign it to a variable.

After selecting the method from the list, the IDE fills the parameters with variables that it believes we will use. In this case, it should use `msg` as the parameter. If not, simply type `msg` as parameter in `broadcastMessage method` This completes the broadcast method. Now, we can call it from the `onEnable` method. We will pass the `Hello World!` string as an argument.

Adding this line of code will result in the class containing the following code:

```
public class MyFirstBukkitPlugin extends JavaPlugin {
  @Override
  public void onEnable() {
    broadcastToServer("Hello World!");
  }

    /**
    * Sends a message to everyone on the server
    *
    * @param msg the message to send
    */
  private void broadcastToServer(String msg) {
    Bukkit.broadcastMessage(msg);
  }
}
```

If we test this plugin, then it will print `Hello World!` once it is enabled.

## Expanding your code

Before testing the code, let's improve on the `onEnable` method by implementing an `if` statement. If there is only one player online, then why not say hello to that specific player? We can get a collection of all the players that are online by calling `Bukkit.getOnlinePlayers`. If we wish to check whether the size of the collection is equal to 1, we can accomplish this by using an `if/else` statement. This is demonstrated in the following code:

```
if (Bukkit.getOnlinePlayers().size() == 1) {
   //Only 1 player online
  //Say 'Hello' to the specific player
} else {
  //Say 'Hello' to the Minecraft World
  broadcastToServer("Hello World!");
}
```

Within the `if` statement, we will now get the first and the only object in the collection of players. Once we have it, we can continue by broadcasting `Hello` along with the player's name. After completing the `if` statement, the entire class will look like the following code:

```
package com.codisimus.myfirstbukkitplugin;

 importorg.bukkit.Bukkit;
 importorg.bukkit.entity.Player;
 importorg.bukkit.plugin.java.JavaPlugin;

    /**
    * Broadcasts a hello message to the server
    */
public class MyFirstBukkitPlugin extends JavaPlugin {
  @Override
  public void onEnable() {
    if (Bukkit.getOnlinePlayers().size() == 1) {

      //Loop through the collection to access the single player
      for (Player player : Bukkit.getOnlinePlayers()) {
        //Say 'Hello' to the specific player
        broadcastToServer("Hello " + player.getName());
      }
```

```
        } else {
            //Say 'Hello' to the Minecraft World
            broadcastToServer("Hello World!");
          }
        }

        /**
        * Sends a message to everyone on the server
        *
        * @param msg the message to send
        */
        private void broadcastToServer(String msg) {
          Bukkit.broadcastMessage(msg);
        }
      }
```

If you do not fully understand the `if` statement or the code provided, then I suggest that you go to my website to learn the basics of Java, which is a prerequisite that was mentioned in the preface of this book.

## Summary

Your first plugin is complete and ready to be tested on your server. In the next chapter, we will install your new plugin, learn how to test it, and discover when the `onEnable` method is executed by the server. Now that you are familiar with writing and calling methods, you can now create basic plugins. Each plugin that you create from now on will always start in a way that is similar to the way this one was started, that is, first create a new project and add Bukkit as a library then fill out the `plugin.yml` file and finally set up your main class as a `JavaPlugin` class with the `onEnable` method.

# 4 Testing on the Spigot Server

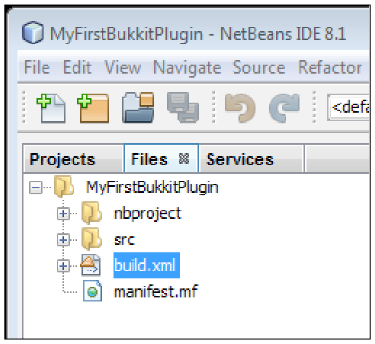
Bukkit plugins are designed to run on a CraftBukkit or Spigot server. At this point, you have a Spigot server and a simple plugin. After completing this chapter, you will have your new plugin installed on your server. In this chapter, you will make changes to your plugin's code, and you will quickly see it being reflected on your server. This will help you develop the plugin much faster and allow you to accomplish more as you create new plugins. You will also learn how to troubleshoot the code in order to fix it when it is not working properly. This chapter will cover the following topics:

- Building a JAR file for your plugin
- Installing the plugin on your server
- Testing the plugin
- Testing new versions of the plugin
- Debugging the code

## Building a JAR file

In order to install a plugin on a server, we need a `.jar` file. A `JAR` file is a Java executable. It contains all the written code, which is bundled together in a ZIP file format. This code needs to be translated so that a computer can understand and run it.

In NetBeans, there is a single button that we can click on to build a project. This will generate the `.jar` file that we need. Let's add a block of code to our project to automatically copy the created `.jar` file to a more convenient location. In NetBeans, click on the **Files** tab to access the `build.xml file` for your project, as shown in the following screenshot:


MyFirstBukkitPlugin - NetBeans IDE 8.1
File Edit View Navigate Source Refactor
Projects
Files
Services
MyFirstBukkitPlugin
nbproject
src
build.xml
manifest.mf

Open `build.xml` and add the following block of code after the `import file` line:

```
<target name="-post-jar">
  <copy file="${dist.jar}" todir="../Plugin Jars"
    failonerror="true"/>
</target>
```

This additional code will be executed after the JAR file is successfully built. It will copy the JAR file from the `dist` directory to the specified location. You can change `"../Plugin Jars"` to whichever directory you want. Here, `..` means to go up one folder. Therefore, if your NetBeans project is located at `C:\Users\Owner\Documents\NetBeansProjects\MyFirstBukkitPlugin`, then the `.jar` file will be copied to `C\Users\Owner\Documents\NetBeansProjects\Plugin Jars\MyFirstBukkitPlugin.jar`. Adding this code to each of your plugins will keep them organized in a central folder. After adding this new code, your file will look similar to the following piece of code:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MyFirstBukkitPlugin" default="default" basedir=".">
  <description>Builds, tests, and runs the project
    MyFirstBukkitPlugin.</description>
  <import file="nbproject/build-impl.xml"/>
  <target name="-post-jar">
    <copy file="${dist.jar}" todir="../Plugin Jars"
      failonerror="true"/>
  </target>
</project>
```

> In the preceding code, `failonerror` is set to `true`. This means that an error will be presented when building in case the JAR file fails to be copied to the defined location. An error such as this may indicate that the location does not exist or you have insufficient privileges. You can set this value to `false` if you do not wish to see these warnings.

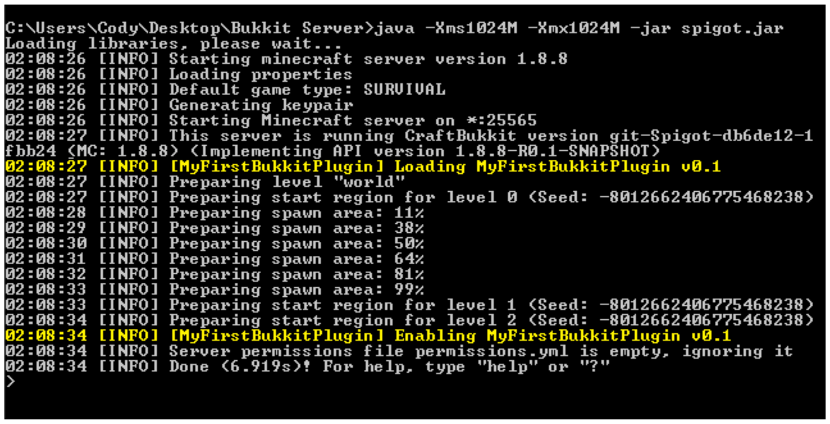
Note that you will have many additional lines between `<!--` and `-->`. These are comments, and I encourage you to read them if you wish to learn more about what you can add to the `build.xml` file. Once you save this file, you are ready to build your project. You can do so by clicking on the hammer icon or using the *F11* shortcut key. The hammer icon, which is present on your toolbar, looks like the following image:

If NetBeans fails to successfully build a jar, then you may have errors in your code.

These errors are most likely shown by the red lines and light bulbs, as seen in *Chapter 3, Creating Your First Bukkit Plugin*. You can usually find these errors by hovering over or clicking on the bulbs for help. If you are not able to do so, refer to the previous chapter to check whether your code is correct or not. If you still have doubts, refer to *Chapter 2, Learning the Bukkit API* for help.

## Installing the plugin

The installation of the new plugin is quite simple. You need to create a copy of the `.jar` file from the directory that you chose earlier in your server's `plugins` folder. Then, start your server as you normally would. You will see the output of the console informing you that the plugin is loaded, as shown in the following screenshot:



```
C:\Users\Cody\Desktop\Bukkit Server>java -Xms1024M -Xmx1024M -jar spigot.jar
Loading libraries, please wait...
02:08:26 [INFO] Starting minecraft server version 1.8.8
02:08:26 [INFO] Loading properties
02:08:26 [INFO] Default game type: SURVIVAL
02:08:26 [INFO] Generating keypair
02:08:26 [INFO] Starting Minecraft server on *:25565
02:08:27 [INFO] This server is running CraftBukkit version git-Spigot-db6de12-1
fbb24 (MC: 1.8.8) (Implementing API version 1.8.8-R0.1-SNAPSHOT)
02:08:27 [INFO] [MyFirstBukkitPlugin] Loading MyFirstBukkitPlugin v0.1
02:08:27 [INFO] Preparing level "world"
02:08:27 [INFO] Preparing start region for level 0 (Seed: -8012662406775468238)
02:08:28 [INFO] Preparing spawn area: 11%
02:08:29 [INFO] Preparing spawn area: 38%
02:08:30 [INFO] Preparing spawn area: 50%
02:08:31 [INFO] Preparing spawn area: 64%
02:08:32 [INFO] Preparing spawn area: 81%
02:08:33 [INFO] Preparing spawn area: 99%
02:08:33 [INFO] Preparing start region for level 1 (Seed: -8012662406775468238)
02:08:34 [INFO] Preparing start region for level 2 (Seed: -8012662406775468238)
02:08:34 [INFO] [MyFirstBukkitPlugin] Enabling MyFirstBukkitPlugin v0.1
02:08:34 [INFO] Server permissions file permissions.yml is empty, ignoring it
02:08:34 [INFO] Done (6.919s)! For help, type "help" or "?"
>
```

If you do not see the `Hello World!` message when your server initially starts, don't worry. This behavior is normal because at this point, there will never be players online for the message to be broadcasted to. For now, we are only concerned with the messages that were highlighted in the previous screenshot.

Every time you make changes to the code, you will have to build a new JAR file and install the new version. To install the newer version, you can simply copy and paste it into the server's `plugin` folder and overwrite the old file. This can be usually done without even shutting down the server. However, if the server is running, you will need to use the `reload` command to load the new version.

If you do not wish to manually copy the `plugin .jar` file and paste it in the server's plugin folder every time you make changes in the code, then you can automate it in `build.xml`. In addition to copying the `jar` file and pasting it in the `Plugin Jars` directory, you can also copy and paste it directly in your server's `plugins` directory. To do so, add a second `copy file` tag and set `todir` to your server's `plugin` directory. The following code is an example of what this will look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MyFirstBukkitPlugin" default="default" basedir=".">
  <description>Builds, tests, and runs the project
    MyFirstBukkitPlugin.</description>
  <import file="nbproject/build-impl.xml"/>
  <target name="-post-jar">
    <copy file="${dist.jar}" todir="../ Plugin Jars"
      failonerror="true"/>
    <copy file="${dist.jar}" todir="C:/Users/Owner/Desktop/Bukkit
      Server/plugins" failonerror="true"/>
  </target>
</project>
```

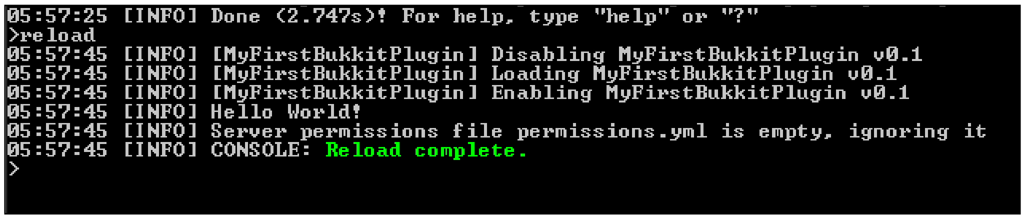
Again, you should do this for every plugin that you want to automatically install on your server.

## Testing your plugin

As you may recall, the purpose of the first plugin was to discover when a plugin is loaded. Issue a `reload` command by typing the following command into the console:

```
>reload
```

You will see that Spigot will automatically disable and re-enable the plugin, as shown in the following screenshot:

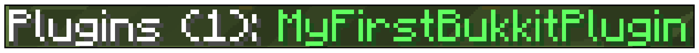


```
05:57:25 [INFO] Done (2.747s)! For help, type "help" or "?"
>reload
05:57:45 [INFO] [MyFirstBukkitPlugin] Disabling MyFirstBukkitPlugin v0.1
05:57:45 [INFO] [MyFirstBukkitPlugin] Loading MyFirstBukkitPlugin v0.1
05:57:45 [INFO] [MyFirstBukkitPlugin] Enabling MyFirstBukkitPlugin v0.1
05:57:45 [INFO] Hello World!
05:57:45 [INFO] Server permissions file permissions.yml is empty, ignoring it
05:57:45 [INFO] CONSOLE: Reload complete.
>
```

This time, you will see the **Hello World!** message once your plugin is enabled. If exactly one player is online, then it will say hello to that player. Let's observe this by logging onto the server and issuing the reload command from in-game. Open your Minecraft client and connect to your server. From in-game, first issue the following command:

```
/plugins
```

You will be given a list of all the plugins that are installed. For now, there is only one plugin, as shown in the following screenshot:

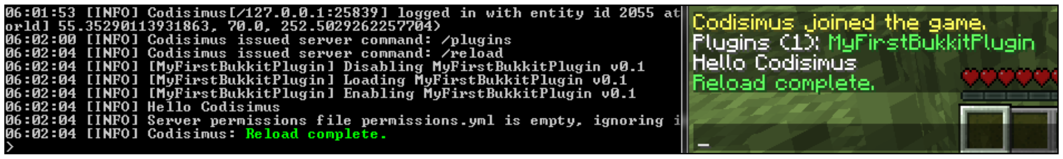


Plugins (1): MyFirstBukkitPlugin

Now that there is one player on the server, we can test the plugin by reloading the server. Issue the following in-game command:

```
/reload
```

Note that in both the game and console, you will see the **Hello Codisimus** message, as shown in the following screenshot, to indicate that the plugin is working as intended:



```
06:01:53 [INFO] Codisimus[/127.0.0.1:25839] logged in with entity id 2055 at
orld] 55.35290113931863, 70.0, 252.5029262257704)
06:02:00 [INFO] Codisimus issued server command: /plugins
06:02:04 [INFO] Codisimus issued server command: /reload
06:02:04 [INFO] [MyFirstBukkitPlugin] Disabling MyFirstBukkitPlugin v0.1
06:02:04 [INFO] [MyFirstBukkitPlugin] Loading MyFirstBukkitPlugin v0.1
06:02:04 [INFO] [MyFirstBukkitPlugin] Enabling MyFirstBukkitPlugin v0.1
06:02:04 [INFO] Hello Codisimus
06:02:04 [INFO] Server permissions file permissions.yml is empty, ignoring i
06:02:04 [INFO] Codisimus: Reload complete.
>
```

```
Codisimus joined the game.
Plugins (1): MyFirstBukkitPlugin
Hello Codisimus
Reload complete.
```

## Testing new versions of the plugin

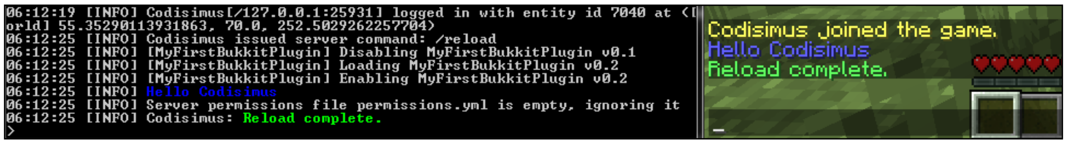
The plugin works as intended, but there is always room for improvement. Let's continue working with this plugin by adding code to it.

A player may not see the **hello** message when it is white in color. We can change the color of the message using `ChatColor Enum`. This `Enum constants` has all the color codes that are supported in-game so that we can easily add them to messages. Let's modify the plugin and install the newly modified version on the server. Choose your favorite color and place it before the message in the `broadcastToServer` method, as shown in the following code:

```
Bukkit.broadcastMessage(ChatColor.BLUE + msg);
```

Before you build a new JAR file, change the version in `plugin.yml` to `0.2` to indicate that this is an updated version. Each time you make a revision to the code, you create a new version. Changing the version number to reflect the code change will ensure that the new code will have a unique version number assigned to it. This is valuable if you need to know the code changes that are included within a specific edition of the project.

Build a new JAR file using either the build icon or the *F11* key. Copy and paste the new version into the `plugins` folder if you did not set up `build.xml` to do so automatically. Issue the `reload` command again to view the results, as shown in the following screenshot:



```
06:12:19 [INFO] Codisimus[/127.0.0.1:25931] logged in with entity id 7040 at ([
orld] 55.35290113931863, 70.0, 252.5029262257704)
06:12:25 [INFO] Codisimus issued server command: /reload
06:12:25 [INFO] [MyFirstBukkitPlugin] Disabling MyFirstBukkitPlugin v0.1
06:12:25 [INFO] [MyFirstBukkitPlugin] Loading MyFirstBukkitPlugin v0.2
06:12:25 [INFO] [MyFirstBukkitPlugin] Enabling MyFirstBukkitPlugin v0.2
06:12:25 [INFO] Hello Codisimus
06:12:25 [INFO] Server permissions file permissions.yml is empty, ignoring it
06:12:25 [INFO] Codisimus: Reload complete.
>
```

```
Codisimus joined the game.
Hello Codisimus
Reload complete.
```

The plugin has been reloaded and the message is now in color. Also, note how the version number changes when the plugin is disabled, and again when it is loaded and enabled. This makes it clear that the new version of the plugin was successfully installed on the server.

Try to expand this plugin more on your own to test different code. The following list contains a few challenges for you:

- Program the plugin to display the actual name of the world rather than the word **World**. A hint for this challenge is that you can get a list of all the worlds and then use the first world in the list. Note that this will broadcast `Hello world!`, unless you have renamed the world in `server.properties`.

- Send a message to the player rather than broadcasting the message to the entire server.
- If more than one player is online, send a unique hello message to each player. A hint for this is that you can use a `for` loop.
- If no players are online, send a unique hello message for each world.

## Debugging the code

As you develop this plugin as well as other Bukkit plugins, there will be times when the code that you have written does not function as you would expect. This is caused by an error that lies somewhere in the code. These errors are referred to as **bugs**, and the process of finding these bugs and removing them is called **debugging**.

## Learning from your mistakes

Do not be discouraged when your plugin does not work on the first try. Even experienced programmers encounter bugs throughout their code. Just because your software does not work perfectly does not mean that you are a poor developer. Being able to discover bugs and fix them is a huge part of software development. The more mistakes you make, the more you can learn from them. This is apparent in the following example.

Some day, you may write a plugin that has a list of players in it. You may then write the following `for` loop to loop through each player and remove those who are in the `CREATIVE` mode:

```
for (Player player : playerList) {
  if (player.getGameMode() == GameMode.CREATIVE) {
    playerList.remove(player);
  }
}
```

When you test this code, an error will likely occur. The error that is thrown will be a `ConcurrentModificationException method`. The name of the exception may not mean much to you, but it will help you narrow down the issue.

Developers need not know how to fix every error, but they should know where to find information about those errors so that they can figure out how to fix them. This is usually found in the software's documentation or public message boards. Most developers will make a note of the error message and search for information about it using Google. The top results will often be the official documentation or posts from other people who have encountered the same issue.

To know more about the error, you can search for `ConcurrentModificationException`; you may find the following statement from Oracle's **Javadoc**:

> "*For example, it is not generally permissible for one thread to modify a Collection while another thread is iterating over it.* "

The official javadoc can prove to be useful. But sometimes, they are still difficult to understand. Luckily, there exist websites such as `stackoverflow.com`, which allow programmers to help each other debug code. If you refer back to the search results, you will see links to Stack Overflow questions as well as posts from similar websites. These links can be very helpful because often there are people just like you who have run into the same error. If you look at the questions and the answers provided by others, you can learn why this error occurs and how to remedy it.

After reading through the questions related to the concurrent modification bug, you will eventually learn that in most cases, the exception occurs when you attempt to modify a list or collection while looping through it. You will also find that you must use an iterator in order to avoid this scenario. Usually, there are examples of how to fix the bug too. In this case, there are explanations of how to correctly use an iterator to remove objects from a list. If an explanation is not present, then you can research iterators within Oracle's javadoc just as you would use Bukkit's javadoc. We can fix the previous code by using an iterator, as follows:

```
Iterator<Player> itr = playerList.iterator();
while (itr.hasNext()) {
  Player palyer = itr.next();
  if (player.getGameMode() == GameMode.CREATIVE) {
    itr.remove();
  }
}
```

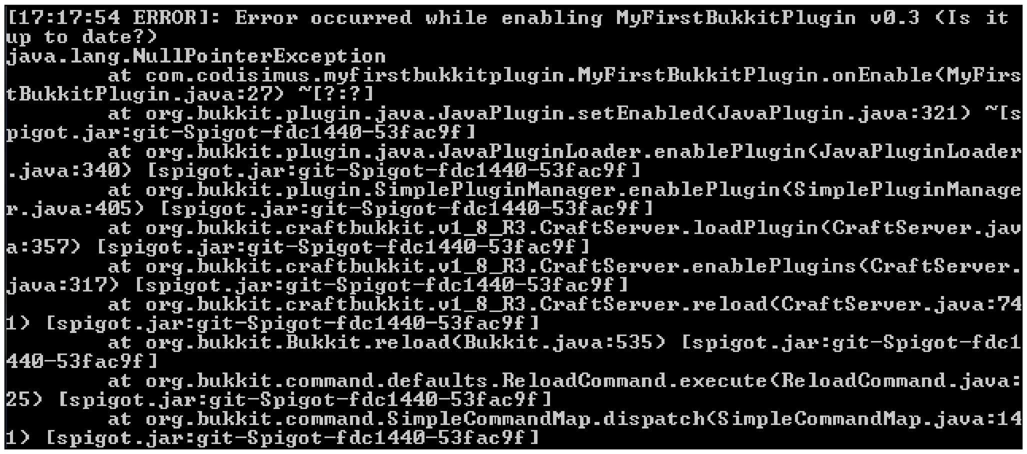
After fixing the concurrent modification bug that was present in the code, you are now a more experienced programmer. You will know how to avoid this issue in the future and you have even learned how to use an iterator in the process.

## When researching is not enough

Sometimes, looking through documentation and reading message boards is not enough to fix a bug. Errors such as **NullPointerException** are very common and can be caused by a variety of things. By researching, you will discover that **NullPointerException** occurs when you attempt to access a method or field of a **null** object. "Null" refers to the lack of a value. Therefore, a null object is an object that does not exist. However, knowing this does not always help you find exactly which object has a null value and why it is a null value in the first place. To aid in finding bugs, here are some steps to follow to pinpoint the troublesome code.

## Reading the stack trace

Most errors in Java are presented in the form of a **stack trace**. A stack trace informs you of the lines of code that were being executed before the error occurred. On a Spigot server, these stack traces will appear similar to the following screenshot:



```
[17:17:54 ERROR]: Error occurred while enabling MyFirstBukkitPlugin v0.3 (Is it
up to date?)
java.lang.NullPointerException
        at com.codisimus.myfirstbukkitplugin.MyFirstBukkitPlugin.onEnable(MyFirs
tBukkitPlugin.java:27) ~[?:?]
        at org.bukkit.plugin.java.JavaPlugin.setEnabled(JavaPlugin.java:321) ~[s
pigot.jar:git-Spigot-fdc1440-53fac9f]
        at org.bukkit.plugin.java.JavaPluginLoader.enablePlugin(JavaPluginLoader
.java:340) [spigot.jar:git-Spigot-fdc1440-53fac9f]
        at org.bukkit.plugin.SimplePluginManager.enablePlugin(SimplePluginManage
r.java:405) [spigot.jar:git-Spigot-fdc1440-53fac9f]
        at org.bukkit.craftbukkit.v1_8_R3.CraftServer.loadPlugin(CraftServer.jav
a:357) [spigot.jar:git-Spigot-fdc1440-53fac9f]
        at org.bukkit.craftbukkit.v1_8_R3.CraftServer.enablePlugins(CraftServer.
java:317) [spigot.jar:git-Spigot-fdc1440-53fac9f]
        at org.bukkit.craftbukkit.v1_8_R3.CraftServer.reload(CraftServer.java:74
1) [spigot.jar:git-Spigot-fdc1440-53fac9f]
        at org.bukkit.Bukkit.reload(Bukkit.java:535) [spigot.jar:git-Spigot-fdc1
440-53fac9f]
        at org.bukkit.command.defaults.ReloadCommand.execute(ReloadCommand.java:
25) [spigot.jar:git-Spigot-fdc1440-53fac9f]
        at org.bukkit.command.SimpleCommandMap.dispatch(SimpleCommandMap.java:14
1) [spigot.jar:git-Spigot-fdc1440-53fac9f]
```

> If your server is hosted somewhere else and you are viewing the console through an online browser tool, the stack trace may be printed in the reverse order.

Whenever there is an exception on your server, Spigot logs the error along with the plugin that caused it. Sometimes, there are even details of which event was in progress when the error occurred. With the stack trace shown in the preceding screenshot, we can see that the error was caused by `MyFirstBukkitPlugin version 0.3`. If the version does not match the version that you have in your **IDE**, you will want to update the server with the latest version of the plugin. This way, you can be certain that the code running on the server is the same code that you have in NetBeans. We can also see that the exception was thrown when the plugin was being enabled. On the next line, we see the specific error, which was **NullPointerException**. On the line following that, we are told the exact line of code that caused the error. It happened within the `onEnable` method of the `MyFirstBukkitPlugin` class. In parenthesis, it states **MyFirstBukkitPlugin.java:27**. This tells us that the error was on line 27 of the `MyFirstBukkitPlugin` class, which of course is in the `onEnable` method. The first three lines of the stack trace are the most useful to us. You rarely have to look at the further lines for explanation. Sometimes, you will not see any of the class names in the code at the start of the stack trace. However, if you look further, you will probably see a familiar class and method name.

Now that you have the class name and line number, you can look back at your code to see if you notice why you are getting **NullPointerException**. Within NetBeans, I can see that line 27 is as follows:

```
Bukkit.getPlayer("Codisimus").sendMessage("Hello, there are " +
Bukkit.getOnlinePlayers().size() + " player(s) online and " + Bukkit.
getWorlds().size() + " world(s) loaded.");
```

## Breaking down the code

The troublesome line is a very long line of code. So, it is not apparent which object has a null value. If you find yourself in a similar situation, I recommend that you break down the code into multiple lines. This will give us the following code:

```
Player player = Bukkit.getPlayer("Codisimus");
int playerCount = Bukkit.getOnlinePlayers().size();
int worldCount = Bukkit.getWorlds().size();
player.sendMessage("Hello, there is " + playerCount + " player(s)
online and " + worldCount + " world(s) loaded.");
```

After installing and running this new code, you should see the same error in the console, but it will point you to a different line of code. With this new code, you will now see that the exception is thrown on line 30, which is the last line of the preceding code segment.

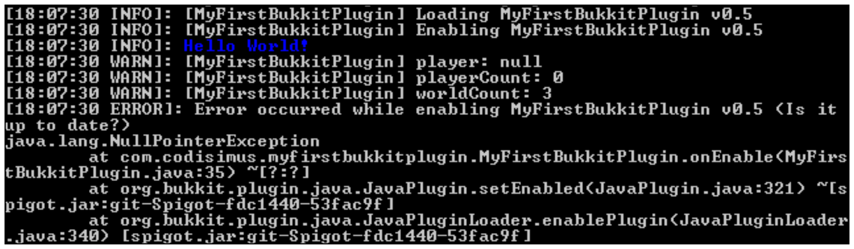
## Adding debug messages

There is still a lot going on in that single line of code. Therefore, you may not be sure about which variable is null. Is it `player`, `playerCount`, or `worldCount`? If you need some additional help, you can add what we call debug messages to your code. These messages can print information to the console log to indicate what is happening throughout the code. There are a few ways to log messages in a Bukkit plugin. The easiest way is to use the `System.out.println(String string)` method. However, it is a better practice to utilize the logger that the Spigot server assigns to your plugin. The logger can be retrieved with the `getLogger` method. This method is in the `JavaPlugin` class. You can access it from within the `onEnable` method. These debug messages will only be temporary. Therefore, you can use whichever method you prefer. But I do suggest that you try using the logger as it also prints out plugin information. In our example, we will use the logger.

Now that we know how to print messages, let's log the values of each variable, as follows:

```
Player player = Bukkit.getPlayer("Codisimus");
int playerCount = Bukkit.getOnlinePlayers().size();
int worldCount = Bukkit.getWorlds().size();
Logger logger = getLogger();
logger.warning("player: " + player);
logger.warning("playerCount: " + playerCount);
logger.warning("worldCount: " + worldCount);
player.sendMessage("Hello, there is " + playerCount + " player(s)
online and " + worldCount + " world(s) loaded.");
```

Note that we have added the debug messages before the faulty line of code. Once an exception is thrown, the computer stops executing the code. Therefore, if the debug messages were after the `sendMessage` call, the messages would never be printed.



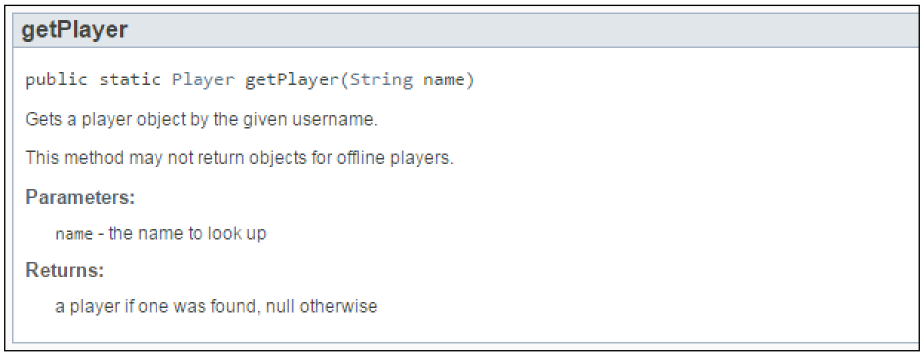
```
[18:07:30 INFO]: [MyFirstBukkitPlugin] Loading MyFirstBukkitPlugin v0.5
[18:07:30 INFO]: [MyFirstBukkitPlugin] Enabling MyFirstBukkitPlugin v0.5
[18:07:30 INFO]: Hello World!
[18:07:30 WARN]: [MyFirstBukkitPlugin] player: null
[18:07:30 WARN]: [MyFirstBukkitPlugin] playerCount: 0
[18:07:30 WARN]: [MyFirstBukkitPlugin] worldCount: 3
[18:07:30 ERROR]: Error occurred while enabling MyFirstBukkitPlugin v0.5 (Is it
up to date?)
java.lang.NullPointerException
        at com.codisimus.myfirstbukkitplugin.MyFirstBukkitPlugin.onEnable(MyFirs
tBukkitPlugin.java:35) ~[?:?]
        at org.bukkit.plugin.java.JavaPlugin.setEnabled(JavaPlugin.java:321) ~[s
pigot.jar:git-Spigot-fdc1440-53fac9f]
        at org.bukkit.plugin.java.JavaPluginLoader.enablePlugin(JavaPluginLoader
.java:340) [spigot.jar:git-Spigot-fdc1440-53fac9f]
```

Once you install and run the updated code, you will see the debug messages within the console:

Now, we can clearly see that `player` has a `null` value.

## Referring back to the Javadoc

If you look back at the line of code where `player` is set and then read the Bukkit javadoc, you will learn that player has a value of  null because the requested player, Codisimus, is not online. If the player cannot be found, `null` is returned.


**getPlayer**

```
public static Player getPlayer(String name)
```

Gets a player object by the given username.

This method may not return objects for offline players.

**Parameters:**

`name` - the name to look up

**Returns:**

a player if one was found, null otherwise

As you can see, the bug in the code may not be exactly at the line given in the stack trace. In this scenario, the null value is set a few lines earlier.

## Fixing the bug only after you understand it

Now that the bug is exposed, we can fix it. In the case of **NullPointerException**, there are two solutions. Do not simply fix the bug a certain way because you can. You should strive to understand why the bug is present and how the code should function instead. Maybe, the variable player is never supposed to have a null value. If I know that the player Codisimus will always be online, then perhaps, I misspelled the username. However, we know that Codisimus will not always be online. So, within this plugin, the player variable will sometimes have a null value. In that scenario, we do not want to try to send a message to the player, since it will throw **NullPointerException**. To remedy this, we can put the line of code within an `if` statement, which developers typically refer to as a null check:

```
Player player = Bukkit.getPlayer("Codisimus");
int playerCount = Bukkit.getOnlinePlayers().size();
int worldCount = Bukkit.getWorlds().size();
//Logger logger = getLogger();
//logger.warning("player: " + player);
//logger.warning("playerCount: " + playerCount);
//logger.warning("worldCount: " + worldCount);
if (player != null) {
    player.sendMessage("Hello, there is " + playerCount + " player(s) 
online and " + worldCount + " world(s) loaded.");
}
```

Note that I have changed the debug messages to comments by prepending them with // so that those messages are not printed to the log. Alternatively, I can remove these lines completely if I feel that I will never need them again.

Now that we have added the null check, the message will only be sent if the player is not null.

## Summary

You now know how to create a JAR file from a NetBeans project. For the plugins that you will create in the future, you can follow this simple process in order to install and run your new plugin, whether it is for testing or for a finished product. You also know how to update a plugin that is already installed on your server and fix the bugs that are exposed in your code. In the following chapters, we will create increasingly complex plugins. The first step to this is creating commands for plugins that players will be able to execute in game.

# 5
# Plugin Commands

The nice thing about the Bukkit API is that it has the basic features already built into its framework. As programmers, we need not go out of our way to implement these basic features into plugins. In this chapter, we will discuss one of these features, namely the in-game commands that can be executed by a player. These are similar to the commands that you are already familiar with, such as `/reload`, `/gamemode`, or `/give`. We will create a plugin that will *enchant* an item. By the end of this chapter, once the plugin is complete, you will be able to type `/enchant` to add your favorite enchantments to the item in your hand.

Commands are one of the easiest ways for players to communicate with a plugin. They also allow players to trigger the execution of a plugin's code. For these reasons, most plugins will have some sort of command. The Bukkit development team realized this and provided us with a simple way to register commands. Registering commands through Bukkit ensures that a plugin will know when a player types a command. It also prevents a plugin from having conflicts with another plugin's commands. The following are the three steps that we will cover to add a command to a plugin:

- Informing Bukkit that a plugin will be using a command
- Programming what a plugin will do when someone types a command
- Assigning newly written code to a specific command

## Adding a command to plugin.yml

Create a new Bukkit plugin as you did in *Chapter 3*, *Creating Your First Bukkit Plugin*, but name it `Enchanter`. Alternatively, you can create a copy of your existing project and modify the name, package, and so on in order to create a new plugin. This will eliminate the need to add the required libraries and configure the build script.
A project can be copied by performing the following steps:

1. Right-click on the project that you wish to copy and select **Copy…** from the menu.
2. Set the project name. The project location should remain unchanged.
3. Open `build.xml`, as discussed in *Chapter 4*, *Testing on the Spigot Server*, and change the project's name to match what was set in step 2.
4. Update the package in your new project so that it is unique by right-clicking on the package and selecting **Rename…** in the **Refactor** menu item.
5. Rename the main class, if necessary. You can also remove the methods or classes that you know will not be reused.
6. Finally, modify the `plugin.yml` file with the new plugin information, including name, main, version, and description.

Next, we will inform Bukkit that we will use a command by modifying the `plugin.yml` file of the plugin. As mentioned in *Chapter 2*, *Learning the Bukkit API*, Spigot reads the YAML file in order to find out necessary information about the plugin. This information includes all the commands that your plugin will handle. Each command can have a description, a proper usage message, and aliases, which is similar to how `rl` is an alias for `reload`. The command that we will use for the plugin will be `enchant`. It is typical to use lowercase letters for commands so that players do not have to worry about capitalization when typing the in-game command.
The following code is a sample of how `plugin.yml` will appear after the `enchant` command is added:

```
name: Enchanter
version: 0.1
main: com.codisimus.enchanter.Enchanter
description: Used to quickly put enchantments on an item
commands:
  enchant:
    aliases: [e]
    description: Adds enchantments to the item in your hand
    usage: Hold the item you wish to enchant and type /enchant
```

Note how the lines are indented. This indentation must be spaces and not tabs. NetBeans helps us automatically indent the necessary lines as you type them. In addition to this, NetBeans will automatically use spaces even if you use the *Tab* key. Indentation is very important in YAML files as this determines the hierarchy of keys. The `enchant command` is indented under `commands` to indicate that it is a command for the plugin. The `aliases`, `description`, and `usage` commands are indented under `enchant` to indicate that they belong to the `enchant` command.

[ The order of these three settings does not matter and they are optional. ]

The usage message will be displayed in case an error occurs or a player uses a command incorrectly. The description message can be viewed by issuing the help command for the plugin, that is, `/help Enchanter`.

For `aliases`, we have `e` as a value. This means that we can type `/e` if we feel that `/enchant` is too long to type. You may have more aliases, but they must be put in a YAML list format. Lists in a YAML file can be created in two different ways. The first format involves separating each item by a comma and a space and enclosing the entire list in square brackets, as shown in the following piece of code:

```
aliases: [e, addenchants, powerup]
```

The second format involves placing each item on a new line, which starts with a hyphen and a space, as shown in the following piece of code:

```
aliases:
  - e
  - addenchant
  - powerup
```

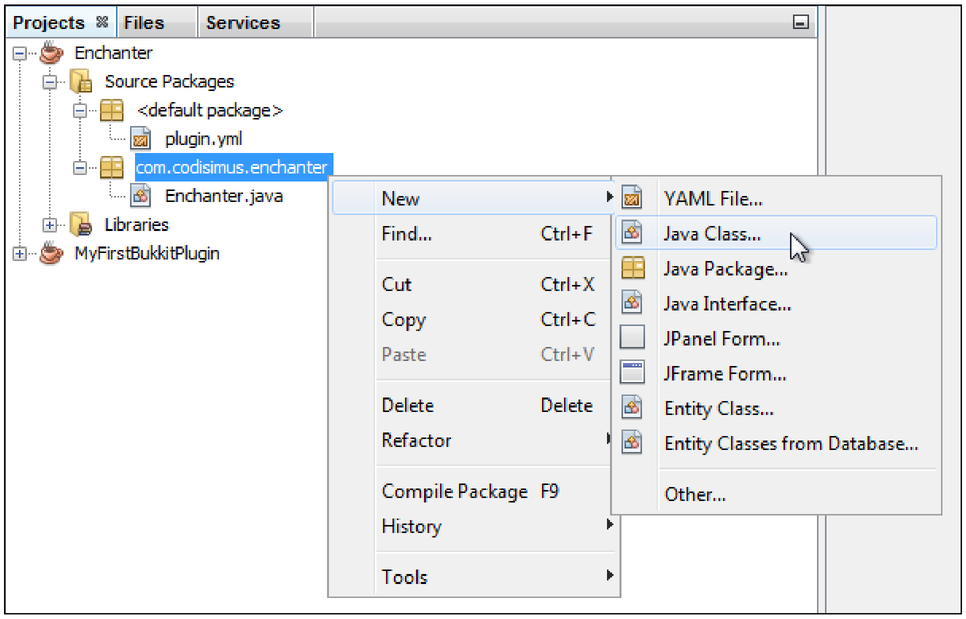
The preferred method is usually determined by the length of the list. The second format is easier to read when lists are long. However, be careful not to have extra or missing spaces before the hyphen, as it will cause problems when a program tries to read the list. In general, ensure that your lists line up. For more information about the YAML language, visit `http://www.yaml.org/spec/1.2/spec.html`.

Multiple commands can be easily added to a plugin. The following code is an example of `plugin.yml` with several commands:

```
name: Enchanter
version: 0.1
main: com.codisimus.enchanter.Enchanter
description: Used to quickly put enchantments on an item
commands:
  enchant:
    aliases: [e, addenchants]
    description: Adds enchantments to the item in your hand
    usage: Hold the item you wish to enchant and type /enchant
  superenchant:
    aliases:
      - powerup
  disenchant:
    description: Removes enchantments from the item in your hand
    usage: Hold the item you wish to disenchant and type /disenchant
```

## Programming the command actions

Once you have added the command to the `plugin.yml` file, you can begin working on the code that the command will trigger. Create a new class in the NetBeans project. This new class will be called `EnchantCommand`. You can name the class something else if you wish, but keep in mind that the name of a class should give you an idea of how the class is used without you having to open it. Place this class in the same package as that of `Enchanter`, the main plugin class, as shown in the following screenshot:

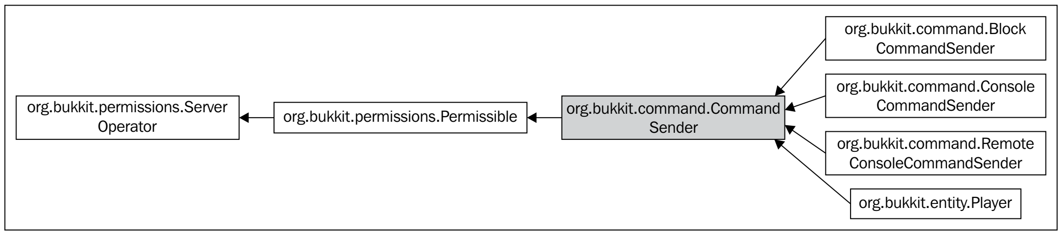


Keep in mind that though the packages are structured similarly, you will use your own unique namespace, not `com.codisimus`.

This new class will execute the `enchant` command. Thus, it must implement the `CommandExecutor` interface. We will append code to the class header to do this. This is similar to adding `extends JavaPlugin` to the `Enchanter` class. `JavaPlugin` is a class. Therefore, we extended it with our class. `CommandExecutor` is an interface, which means that we must implement it. Once we add `implements CommandExecutor` to the class header of `EnchantCommand`, a light bulb will appear to notify us of the need to import the `CommandExecutor` class. Import the class, and the light bulb will still be there. It is now informing us that because we implemented an interface, we must implement all of its abstract methods. Click on the light bulb to do so, and the method that we need appears. This new method will be called when a player executes the `enchant` command. The method provides us with the following four parameters:

- `CommandSender sender`
    - This command can be named `cs` by default, but we will name it `sender` because it is easy to forget what `cs` stands for
    - This is who sent the command
    - It may be a player, the console, a command block, or even a custom `CommandSender` interface that was created by another plugin
- `Command cmnd`
    - This is the `Command` object that the sender executes
    - We will not need this as this class will be used for only a single command
- `String alias`
    - This is which alias the sender typed
    - For example, it might be `enchant`, `e`, `addenchant`, or `powerup`
- `String[] args`
    - This is an array of strings
    - Each string is an argument that the sender type
    - Arguments follow the alias and are separated by a space
    - The command itself is not considered as an argument
    - For example, if they type `/enchant knockback 5`, then `knockback` will be the first argument (`args[0]`) and `5` will be the second and final argument (`args[1]`)
    - We do not need to worry about the arguments at this point, because the `enchant` command will not need any

As mentioned before, there are different kinds of `CommandSenders`. The following image is an inheritance diagram for `CommandSender`:


org.bukkit.permissions.Server Operator
org.bukkit.permissions.Permissible
org.bukkit.command.Command Sender
org.bukkit.command.Block CommandSender
org.bukkit.command.Console CommandSender
org.bukkit.command.Remote ConsoleCommandSender
org.bukkit.entity.Player

In this diagram, you can see that `Player`, `ConsoleCommandSender`, and a couple of other classes are all subtypes of `CommandSender`. The purpose of the enchant command is to allow a player to enchant the item that they are holding. Therefore, a `CommandSender` object that isn't a player will have no use for this command. In the `onCommand` method, the first code that we write will be to check whether a player has executed the command. If we do not perform this check, then the plugin will crash when a nonplayer attempts to issue the `enchant` command. We will check this by using an `if` statement and the `instanceof` keyword. The code for this is as follows:

```
if (sender instanceof Player)
```

This code can be translated to this:

```
if the command sender is a Player
```

This `if` statement will let us know if it was a player who sent the command. If the command sender is not a player, then we want to stop executing the code. We will do this by using the `return` keyword. However, the `return` type for this method is `boolean`. We must return a `boolean` value, which will tell Bukkit whether the usage message needs to be shown to the command sender. Typically, for the `onCommand` method, you want to return `false` if the command is not executed successfully. In this case, it was not. Therefore, we will use the `return false;` code. So far, inside the method, we have constructed the following code:

```
if (sender instanceof Player) {
  return false;
}
```

However, this is not quite right. This asks Bukkit to return `false` if the command sender is a player, but we want to return `false` when the opposite is the case. We can accomplish this by adding an exclamation point. If you don't already know, in Java, an exclamation point is a `NOT` operator and can be used to invert a `boolean` value. We will correct the previous code by inverting the resulting value, as shown in the following code:

```
if (!(sender instanceof Player)) {
  return false;
}
```

Note the extra set of parentheses. This is very important. Parentheses allow expressions to be grouped together. We want to invert the `boolean` value that results from the `sender instanceof Player` code. Without the parentheses, we would be attempting to invert the sender object, which does not make sense. As a result of this, the code would not be compiled.

Up to this point, the `EnchantComand` class code is as follows:

```
package com.codisimus.enchanter;

import org.bukkit.command.Command;
import org.bukkit.command.CommandExecutor;
import org.bukkit.command.CommandSender;
import org.bukkit.entity.Player;

/**
 * Enchants the item that the command sender is holding
 */
public class EnchantCommand implements CommandExecutor {

  @Override
  public boolean onCommand(CommandSender sender, Command cmnd,
     String alias, String[] args) {
    //This command can only be executed by Players
    if (!(sender instanceof Player)) {
      return false;
    }
  }

}
```

Now that we have taken care of the nonplayers, we are certain that the `CommandSender` object is a player. We will want to work with the `Player` object rather than the `CommandSender` object because the `Player` object will have a specific item in its hand. We can get the `Player` object by *casting* the `CommandSender` object to `Player`. By casting, we are telling Java that we know that the command sender is actually a `Player object` rather than a `ConsoleCommandSender object` or one of the other subtypes. Casting is done by using the following syntax:

```
Player player = (Player) sender;
```

If you are not already familiar with casting, I again suggest that you learn some of these programming concepts at `codisimus.com/learnjava`.

Now that we have the `Player object`, we need the item that they are holding. Looking at the Bukkit API documentation for the `Player` class, which is available at `https://hub.spigotmc.org/javadocs/bukkit/org/bukkit/entity/Player.html`, you can see that there exists a `getItemInHand` method, which is inherited from `HumanEntity`. It will return an `ItemStack` class, which is exactly what we want. This is demonstrated in the following piece of code:

```
ItemStack hand = player.getItemInHand();
```

Before doing anything with this item, we have to ensure that there actually is an item to enchant. If the player runs the command when they have no item in their hand, we do not want the plugin to crash. We will check whether the value of `ItemStack class` is `null` and the type of the item is `AIR`. In either case, we will return `false`, as follows, because the command wasn't executed:

```
if (hand == null || hand.getType() == Material.AIR) {
  return false;
}
```

If we do not include the `null` check (hand == null) here in the code, we may encounter a `NullPointerExceptions error`, as discussed in *Testing on the Spigot Server*.

Now, we have a reference to the player and the item that they are holding. Our end goal is to enchant this item. Again, looking at the API documentation, we can find several methods to add enchantments to an `ItemStack class` at `https://hub.spigotmc.org/javadocs/bukkit/org/bukkit/inventory/ItemStack.html`. Read through the descriptions to find out which one is right for us.

Two of the methods are used to add multiple enchantments at once. We may want to add more than one enchantment, but to simplify the code, we will only add one at a time. The two remaining methods are `addEnchantment(Enchantment ench, int level)` and `addUnsafeEnchantment(Enchantment ench, int level).`

The description for the unsafe method states that: *This method is unsafe and will ignore level restrictions or item type. Use at your own discretion.* This warning is provided because these unsafe enchantments have not been tested and could produce undesirable results. You shouldn't let this deter you from using the method but you will want to test the enchantment before using it with friends to ensure that it doesn't crash the server.

Therefore, if we choose to go with `unsafe`, we can create powerful enchantments, such as a sharpness level of 10. Without a plugin, a sword is limited to a sharpness of level 5. With unsafe enchantments, we can also enchant items that were previously unenchantable, such as a fish with `KNOCKBACK` or `FIRE_ASPECT`. Now, you will start to discover all the fun and cool things that you can do with plugins and which could not be done with a vanilla game.

From a personal experience, I found out that the `KNOCKBACK` enchantment is quite entertaining. In my example, I will apply `KNOCKBACK` to the item, but you should of course choose whichever enchantment you prefer. For a full list of enchantments and what they do, visit the API docs at `https://hub.spigotmc.org/javadocs/bukkit/org/bukkit/enchantments/Enchantment.html` or the Minecraft Wiki at `http://minecraft.gamepedia.com/Enchanting#Enchantments`. Bukkit does warn us that using an unsafe method can cause problems. To avoid conflicts, try to keep the enchantment levels at 10 or below. With most enchantments, you will not even notice a difference after level 10. We have decided that we will use `addUnsafeEnchantment(Enchantment ench, int level)`. This method takes an `Enchantment` and an `int` value as parameters. This `int` value is of course the enchantment's level, as stated in the API documentation. We have decided what we want each of these parameters to be. We can complete the line of code, as shown in the following piece of code:

```
hand.addUnsafeEnchantment(Enchantment.KNOCKBACK, 10);
```

For added fun, we will add the `FIRE_ASPECT` enchantment as well, as shown in the following piece of code:

```
hand.addUnsafeEnchantment(Enchantment.FIRE_ASPECT, 1);
```

At this point, everything will be executed successfully. Before we return `true`, we should send a message to the player to let them know that everything worked as planned. We will use the `sendMessage` method to send the message to only this player by using the following line of code. No one else on the server, including the console, will see the message:

```
player.sendMessage("Your item has been enchanted!");
```

The completed class is shown in the following lines of code. Remember to comment your code as you type it. Some of the comments in the following code may seem unnecessary, as the code is easy enough to read. We will refer to this code by the term self-documentation. You only need to leave comments for the code that may be difficult to understand in the future or which may need clarification. While you are still learning, I encourage you to overuse comments. They won't hurt anything by being present, and they will clearly explain the code for you in case you need it:

```
package com.codisimus.enchanter;

import org.bukkit.Material;
import org.bukkit.command.Command;
import org.bukkit.command.CommandExecutor;
import org.bukkit.command.CommandSender;
import org.bukkit.enchantments.Enchantment;
import org.bukkit.entity.Player;
import org.bukkit.inventory.ItemStack;

/**
 * Enchants the item that the command sender is holding
 */
public class EnchantCommand implements CommandExecutor {

  @Override
  public boolean onCommand(CommandSender sender, Command cmnd,
   String alias, String[] args) {
  //This command can only be executed by Players
    if (!(sender instanceof Player)) {
    return false;
  }

  //Cast the command sender to a Player
  Player player = (Player) sender;

  //Retrieve the ItemStack that the Player is holding
  ItemStack hand = player.getItemInHand();

  //Return if the Player is not holding an Item
  if (hand == null || hand.getType() == Material.AIR) {
    return false;
  }

  //Add a level 10 Knockback enchantment
  hand.addUnsafeEnchantment(Enchantment.KNOCKBACK, 10);

  //Add a level 1 Fire Aspect enchantment
  hand.addUnsafeEnchantment(Enchantment.FIRE_ASPECT, 1);

  player.sendMessage("Your item has been enchanted!");
  return true;
  }

}
```

The preceding code implements the enchant command. It verifies that the command sender is a player and that the player is holding an item. It then adds defined enchantments to the item. This concludes the work that needed to be done in the `EnchantCommand` class.

## Assigning the executor for the enchant command

We are almost ready to start using the command on the server. The only remaining step is to assign the class that we just wrote to the `enchant` command. This is typically referred to as registering a command. In the `onEnable` method of the `Enchanter` class, we will get the `enchant` command using the `getCommand("enchant")` code.

[ The name of the command must be exactly as it is in `plugin.yml`. This also means that this code will only retrieve commands that are specific to this plugin. ]

Once we have the enchant command, we can set a new instance of `EnchantCommand` as the executor for the command. All of this can be done in one line, as shown in the following piece of code:

```
getCommand("enchant").setExecutor(new EnchantCommand());
```

All that you will have in the `main` class is shown in the following code:

```
package com.codisimus.enchanter;

import org.bukkit.plugin.java.JavaPlugin;

  /**
  * Enchants the item that the command sender is holding
  */
public class Enchanter extends JavaPlugin {
  @Override
  public void onEnable() {
    //Assign the executor of the enchant command
    getCommand("enchant").setExecutor(new EnchantCommand());
  }
}
```

## Summary

You now have a useful plugin to play with on your own server. You can build this plugin, as discussed in the previous chapter, and put it on your server to test. Try it with different items and observe how it works. There are many plugins that can be created and which solely function by using commands. With this knowledge, you have the potential to create numerous plugins. You can try a few plugins, like a plugin that teleports you to the spawn location of the world using `/spawn`, a plugin that plays the Creeper Hiss sound to a specific player using `/scare <player>`, and a plugin that strikes a player with lightning using `/strike <player>` by yourself.

> There is a `strikeLightning` method within the `World` class.

For the plugin, you will have to use arguments. First, you will need to check whether you were given the correct number of arguments. Then, you will have to get the first argument, as explained earlier in this chapter. This argument will be the name of a player. There is a method in the `Bukkit` class to find a player with a given name.

If you are ever searching for a plugin idea, remember that the API documentation is a great source of inspiration. Also, people are always looking for plugins to be made on the Bukkit, Spigot, and Minecraft forums. In the next chapter, we will expand on the `Enchanter` plugin by adding permissions to it. This will ensure that only privileged players will be able to enchant items using the `enchant` command.

# 6
# Player Permissions

Player Permissions is one feature that nearly every Bukkit server administrator wants to have on their server. In vanilla Minecraft, you are either an **OP** (**operator**), or simply a regular player. With permissions, you can create an infinite number of ranks between the two. There are several permission plugins available, which can be found on the Bukkit or Spigot websites. In the past, developers had to write their own code in order to support one or more of these permission systems. Luckily, the Bukkit API now has a basis for player permissions, which makes our job easier. We no longer need to learn a new API for every permissions plugin that exists. We only need to support Bukkit's universal permissions system, which we can be sure will not change drastically at any moment. In this chapter, you will do just that and install a permissions plugin that helps you organize each player's permissions. By the end of this chapter, you will be able to control your server in a way that ensures that untrusted players will not be able to spoil the fun for everyone else. We will cover the following topics in this chapter:

- The benefits of using permissions on your server and in plugins
- What a permission node is and how it is used by developers and server administrators
- Adding a permission node to the `plugin.yml` file
- Assigning a permission node to one of your plugin's commands
- Testing player permissions in-game
- Installing and configuring a third-party permissions plugin
- Using permission nodes throughout your plugin

## The benefits of permissions

**Permissions** give you more control over the players on your server. They allow you to prevent abuse from untrusted players. With permissions, you can give each player a specific rank based on their role in the server and how trustworthy they are. Let's say that you want to give a specific player the ability to teleport to the location of some other players. With permissions, you can do so without giving that same player the ability to spawn items, kick/ban other players, and even stop your server completely! The simplest example of a useful permission is not giving new players the permission to build. This prevents someone from logging on to your server with the sole intention of defacing the world. They will be unable to destroy your or other players' buildings.

When programming plugins, you can assign certain permissions to specific commands or actions. This allows you to give the benefits of your plugins to privileged people only. For example, you may want only your good friend and yourself to have the option of enchanting your items using the `enchant` command. The first step to accomplishing this is to know what permission nodes are and how they work.

## Understanding permission nodes

A **permission node** is a `string` that usually contains multiple words separated by periods. These permission nodes are given to players to give them special privileges on the server. An example of this is `minecraft.command.give`, which is the permission node that is needed to execute the `give` command. As you can see, it can be broken down into three parts, namely, the creator (Minecraft), the category (command), and the specific privilege (the `give` command). You will find most permission nodes structured this way. For a plugin, its permission nodes begin with the name of the plugin. This helps prevent any collision of nodes. If two plugins were to use the same permission node, then an administrator cannot limit access to one node and not the other node. You will also find that many plugins' permission nodes are only two words long. This is done when the plugin does not have many permissions. Therefore, there is no need for categories. On the other hand, for large plugins, you may wish to include numerous nested categories.

To help you understand permission nodes, we will create a permission node for the Enchanter plugin. The first word of the permission node will be the name of the plugin, while the second word will be the name of the command. If the permission node is directly related to a specific command, then it is wise to use the command name within the permission node. This will make your permissions simple to understand and easy to remember. The permission node for the enchant command will be enchanter.enchant. If we expect this plugin to have several permissions, then we can use enchanter.command.enchant instead. Either permission node is fine, but we will use the former in our example. Note that most developers tend to keep their permission nodes in lowercase. This is optional, but it usually prevents errors when typing in the node later. Once we have decided upon a permission node, we must add it to plugin.yml in order to use it with a plugin.

## Adding a permission node to plugin.yml

In the Enchanter project, open the plugin.yml file. Adding permission nodes is similar to how commands are added. On a new line, add permissions:. Ensure that this line is not indented at all. On the lines that follow, add each permission node that our plugin will use, followed by a colon. The next few lines will provide the attributes of the permission, such as its description. The following code is an example of how the plugin.yml file will look with the enchant permission node added. Ensure that the indentations are similar. Note that the version attribute should also be updated to indicate that this is a new and improved version of the Enchanter plugin:

```
name: Enchanter
version: 0.2
main: com.codisimus.enchanter.Enchanter
description: Used to quickly put enchantments on an item
commands:
  enchant:
    aliases: e
    description: Adds enchantments to the item in your hand
    usage: Hold the item you wish to enchant and type /enchant
permissions:
  enchanter.enchant:
    description: Needed to use the enchant command
    default: op
```

The default attribute can be set to `true`, `false`, `op`, or `not op`. This determines who will have this permission; `true` means that everyone will have this permission, `false` means that no one will have it, `op` means that only operators will have it, and `not op` means that everyone except for the operators will have it. Who has this permission can be further modified by using a permission plugin, which will be discussed later in this chapter.

Just like with commands, you can assign multiple permissions to a plugin. For more information on the `plugin.yml` file, visit `http://wiki.bukkit.org/Plugin_YAML`.

## Assigning a permission node to a plugin command

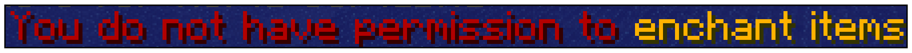
Now that we have created the permission node, we want to prevent players from using the `enchant` command if they do not have the `enchanter.enchant` node. This process is simple, as it only requires adding a few more lines to the `plugin.yml` file.

For the `enchant` command, we will add two attributes, namely `permission` and `permission-message`. The `permission` attribute is simply the permission node that is needed to execute the command. The `permission-message` attribute is a message that the player will see if they do not have the necessary permissions. After these additions, the `plugin.yml` file will look like this:

```
name: Enchanter
version: 0.2
main: com.codisimus.enchanter.Enchanter
description: Used to quickly put enchantments on an item
commands:
  enchant:
    aliases: [e]
    description: Adds enchantments to the item in your hand
    usage: Hold the item you wish to enchant and type /enchant
    permission: enchanter.enchant
    permission-message: You do not have permission to enchant items
permissions:
  enchanter.enchant:
    description: Needed to use the enchant command
    default: op
```

You may want to add colors to the permission message. This can be done by using the § symbol. This is the character that Minecraft uses to indicate a color code. This symbol can be easily typed by holding *Alt* while pressing *2* and then *1*. A list of all the colors and their corresponding code can be found at `http://www.minecraftwiki.net/wiki/Formatting_codes`. An example of the `permissions-message` line with color support looks like this:

```
permission-message: §4You do not have permission to §6enchant items
```


You do not have permission to enchant items

## Testing player permissions

You can test the new addition to the plugin by building the `jar` file and installing it on your server, as discussed in *Chapter 4*, *Testing on the Spigot Server*. Ensure that you reload or restart the server so that the newest version of the plugin is used. Remember that the version number is printed on the console when the plugin is enabled.

By testing on your server, you will find out that you can enchant items through the plugin. Since you are an OP, you have the `enchanter.enchant` node by default. *De-OP* yourself by using the following console command:

```
>deop Codisimus
```

Now, you will no longer be able to use the `/enchant` command.

## Using a third-party permissions plugin

You will most likely have trusted players on your server with whom you wish to share the use of the `/enchant` command. However, these players are not trusted enough to be an OP. In order to share the use of this command, you will need to use a permissions plugin. The permissions plugin will allow you to create multiple groups of players. Each group will have different permissions assigned to it. Each player that plays on your server can then be assigned to a specific group. As an example, you can have four permission groups, namely *default*, *trusted*, *mod*, and *admin*. The *default* group will have the basic permissions. A new player who joins the server will be put into the *default* group. The *trusted* group will have a few more privileges. They will have access to specific commands, such as setting the time of day in the server world and teleporting players. The *mod* group stands for "moderator", and it will have access to many other commands, such as kicking or banning a player. Finally, the `admin` group, which stands for "administrator", will have the `/give` command and the `/enchant` command.

There are several permission plugins available at `dev.bukkit.org`. Each permission plugin is created by a different developer. They have various features depending on how the developer decided to program it. Most of the popular permissions plugins that are used today were actually created before permissions were added to the API. Because of this, they may not utilize all of Bukkit's features. They also include additional features that are no longer needed, such as permission groups. The plugin that we will use is the one that I have developed myself, and it is called `CodsPerms`. `CodsPerms` is a simple and basic permissions plugin. Because `CodsPerms` follows the rules of the Bukkit API, the group configuration that you will learn in this chapter can also be utilized for other permission plugins. Instructions on where to download `CodsPerms` can be found at `http://codisimus.com/codsperms`.

Once you have the `jar` file of the plugin, install it on your server as you would install one of your own plugins. With the plugin installed, the `permission` command will be available to you. Executing the `/perm` command will inform you of the various commands that are now at your disposal.

> You need to have the `permissions.manage` node in order to use the permission command. Until we fully set up the permission plugin, you can either run these commands from the console, or give yourself the OP status.

You will see that there are commands that can be used to give permission nodes to players as well as remove them. This is useful if you want to add single nodes, such as giving yourself the `permissions.manage` node, but you will not want to use those commands for everyone who joins your server. To resolve this, we will configure the groups that were presented earlier.

These groups will be created as a permission node that contains several other child permission nodes. This will allow us to give a player a single group node, and they will then inherit all of its children. We can create these parent nodes within the `permissions.yml` file located in the `root` directory (the same folder in which you placed `spigot.jar`). The `permissions.yml` file is a `YAML` file like `plugin.yml`. Therefore, you should be familiar with the formatting. You can edit this file with a text editor. If you wish to use NetBeans, you can open the file by navigating to **File | Open File...** or by dragging and dropping the file in the NetBeans window.

> Editing a `YAML` file incorrectly will cause it to not load completely. The issue that you will most likely face with the `YAML` files is having a *tab* in your document rather than *spaces*. This will cause your file to not load properly.

The following code is a sample of how `permissions.yml` might look after creating the groups that were specified earlier:

```
group.default:
  description: New Players who may have joined for the first time
  default: true
  children:
    minecraft.command.kill: true
    minecraft.command.list: true
group.trusted:
  description: Players who often play on the server
  default: false
  children:
    group.default: true
    minecraft.command.weather: true
    minecraft.command.time: true
    minecraft.command.teleport: true
group.mod:
  description: Players who moderate the server
  default: false
  children:
    group.trusted: true
    minecraft.command.ban: true
    minecraft.command.pardon: true
    minecraft.command.kick: true
group.admin:
  description: Players who administer on the server
  default: false
  children:
    group.mod: true
    minecraft.command.ban-ip: true
    minecraft.command.pardon-ip: true
    minecraft.command.gamerule: true
    minecraft.command.give: true
    minecraft.command.say: true
    permissions.manage: true
    enchanter.enchant: true
```

Every group can inherit the permission nodes of another group by simply adding that group permission node as one of their children. In this example, the `admin` group inherits all the permissions from the `mod` group, the `mod` group inherits all the permissions from the `trusted` group, and the `trusted` group inherits all the permissions from the `default` group. Therefore, the `admin` group also inherits the `default` group's permissions through parentage. In this sample file, we have `default` set to `true` for the `group.default` parent node. This means that each player on the server will automatically have the `group.default` permission node. Because of the child nodes, each player will also have `minecraft.command.kill` and `minecraft.command.list`. Adding permissions to the default group will eliminate the need to hand out permissions to each player who joins your server.

As you can see, the permission nodes earlier included permissions for some Minecraft commands as well as the permission for the `Enchanter` plugin. There are many more permissions than have already been listed. These are a few that are commonly used. The rest of the permissions for the Minecraft and Bukkit commands can be found at `wiki.bukkit.org/CraftBukkit_commands`.

Once you have populated the permissions `YAML` file, you will have to reload the server in order for the changes to take effect. Now, you can assign players to different groups. Use the following command with your own username to add yourself to the trusted group:

```
>perm give Codisimus group.trusted
```

You will have the permissions defined in `group.trusted` within the `permissions.yml` file. Try putting yourself in various groups and using the `/enchant` commands and various other commands. Ensure that you are not an OP, as it will give you all the permissions regardless of which group you are in. Also, keep in mind that you must manually remove yourself from groups. If a player in the `admin` group is added to the `trusted` group, they will still have administrator privileges until they are removed from the admin group.

## Using permission nodes throughout your plugins

In some cases, you may want to check whether a player has a specific permission from within your code. With the addition of a universal permission system within Bukkit, this is very easy, regardless of the permission plugin that you are using. Looking at the Bukkit API documentation, you will see that the `Player` object contains a `hasPermission` method, which returns a Boolean response. The method requires a `string` value, which is the permission node that is being checked. We can place this method in an `if` statement, as shown in the following code:

```
if (player.hasPermission("enchanter.enchant")) {
  //Add a level 10 Knockback enchantment
  Enchantment enchant = Enchantment.KNOCKBACK;
  hand.addUnsafeEnchantment(enchant, 10);
  player.sendMessage("Your item has been enchanted!");
} else {
  player.sendMessage("You do not have permission to enchant");
}
```

This block of code is unnecessary for the plugin because Bukkit can automatically handle player permissions for commands. To have a look at how this is properly used, let's go back to `MyFirstBukkitPlugin` and add a permission check. The following code is the modified `onEnable` method, which will only say `Hello` to the players who have the necessary permission:

```
@Override
public void onEnable() {
  if (Bukkit.getOnlinePlayers().size() >= 1) {
    for (Player player : Bukkit.getOnlinePlayers()) {
      //Only say 'Hello' to each player that has permission
      if (player.hasPermission("myfirstbukkitplugin.greeting")) {
        player.sendMessage("Hello " + player.getName());
      }
    }
  } else {
    //Say 'Hello' to the Minecraft World
    broadcastToServer("Hello World!");
  }
}
```

Remember that you will also have to modify `plugin.yml` to add the permission node to your plugin.

You can also broadcast a message to only the players who have a specific permission node. The documentation on this can be found at `https://hub.spigotmc.org/javadocs/spigot/org/bukkit/Bukkit.html#broadcast(java.lang.String,%20java.lang.String)`.

Try adding some permission nodes to some other projects that were created in the previous chapters. For example, add the `creeperhiss.scare` permission node to the plugin that has the `/scare <player>` command. As an added challenge, add an option that allows a player to type `/scare all` if they want to scare all the players on the server. In this case, you could check each player for the `creeperhiss.hear` permission node. That way, only those players will hear the sound. This is a good example of a permission node that should be set to `not op` by default.

## Summary

The existing plugins, after being modified, are now more flexible with the aid of a permission plugin. With `CodsPerms` running on your server, you can have multiple groups for players. You can create plugins that give certain players privileged commands, and yet these same players will be prevented from using commands that can be misused. This new knowledge of Bukkit permissions will give you an increased control over both your plugins and server. Now that you know how to program both commands and permissions, you are ready to dive into some of the more challenging and exciting sections of the Bukkit API. In the next chapter, you will learn how to automate your server and customize it by using the Bukkit event system.

# 7
# The Bukkit Event System

At this point, you know how to create a plugin that will run some code when a command is executed. This is very useful in many situations. However, we would rather not be required to type in a command sometimes. We'd prefer it if the code could be automatically triggered to be executed. The trigger could be a specific event that occurs on the server, such as a block being broken, a creeper exploding, or a player sending a message in a chat. The Bukkit event system allows a developer to listen for an event and automatically run a block of code based on that event. By using the Bukkit event system, you can automate your server, which means less work for you to maintain the server in the future. In this chapter, we'll cover the following topics:

- Choosing an event
- Registering an event listener
- Listening for an event
- Canceling an event
- Communicating between events
- Modifying an event as it occurs
- Creating more plugins on your own

## Choosing an event

All the events that Bukkit provides can be found in the API documentation in the `org.bukkit.event` package. Each event is categorized into packages within `org.bukkit.event`, such as `org.bukkit.event.block`, `org.bukkit.event.player`, and `org.bukkit.event.world`. This makes it easy to find the event that you are looking for. A full list of the Bukkit events can be found at `https://hub.spigotmc.org/javadocs/spigot/org/bukkit/event/class-use/Event.html`. I encourage you to take a look at the list to see what type of event you can listen for. Each event has several methods, which give you more information and allow you to modify the event. For example, `BlockBreakEvent` provides methods to get the block that was broken and the player who broke it. Most events can also be canceled if you wish to not allow the events to occur. This is useful in many situations, such as not letting a new player place a TNT block, or preventing a mob from spawning.

As mentioned earlier, listening to events can aid in automating your server and reducing the number of commands being sent. In addition to that, they can simply be a lot of fun to work with. Let's look at a few examples of plugins that can be made using the Bukkit event system. We mentioned that you can listen to the player chat event and modify it as you please. You can use this to monitor messages and censor the offensive words that may be spoken. Placing TNT blocks was also mentioned. You can create a plugin that only lets players place TNT if they have the `build.tnt` permission node. There is also a `WeatherChangeEvent class` that can be canceled. That being said, there are many server administrators who don't like it when it rains on the server. Rain can be loud and annoying. Admins will issue the `/toggledownfall` command to stop the rain every time it starts. In this chapter, we will create a plugin that prevents rain from starting in the first place.

The first thing that we must do is find the appropriate event that we can listen for. To accomplish this, we will have a look at the Bukkit API documentation. Let's say that we are unfamiliar with the API. Therefore, we are unsure about which event we can use. We can look through the list of events until we find the correct one, but you may have better luck if you first find the right package. There are two categories that rain could fall under, namely world events or weather events. It is more likely that rain would be categorized under weather. So, we will look there first. There is no event that includes the word "rain" because rain is categorized with snow. Therefore, the event that we are looking for is the `WeatherChangeEvent class`. If you did not find a correct event to use, look in other packages.

If you are ever unable to find the event that you are looking for, then remember that you can ask for help on the Bukkit/Spigot forums. You can perhaps perform a search on the forums first to check whether anyone else was looking for the same information. It is possible that the event that you are attempting to listen for does not exist. Keep in mind that the Spigot project is not associated with the creators of Minecraft. Therefore, it is impossible to detect or modify some events.

Now that we have found the event, we wish to prevent this event from occurring. Viewing the `WeatherChangeEvent` class reference page, we will see several methods that are offered in this event. We will use the `setCancelled` method to cancel the event and the `toWeatherState` method to ensure that we are only preventing the rain from starting and not stopping.

## Registering an event listener

After deciding which event we will listen for, it is time to start programming. Create a new project, as described in *Chapter 3, Creating Your First Bukkit Plugin*, and call it `NoRain`. Don't forget to create a `plugin.yml` file as well.

In order to listen for an event, your plugin must have a class that is registered as a `Listener class`. We will only have one class, named `NoRain.java`, for this project. Therefore, we will make this a `Listener` class as well. The class declaration will look like the following line of code:

```
public class NoRain extends JavaPlugin implements Listener
```

Alternatively, if this is a large project, you can make a class for the `Listener class`, which is similar to how the `Enchanter` project had `CommandExecutor` as a separate class. Also, like `CommandExecutor`, a `Listener class` will implement an `interface method`. The `interface method` that we wish to implement is `org.bukkit.event.Listener`.

The class is declared as a `Listener` class but it is still not registered with Bukkit. To register all the events within the listener, insert the following line of code in the `onEnable` method:

```
getServer().getPluginManager().registerEvents(this, this);
```

This line retrieves the `PluginManager class` and uses it to register the events. The `PluginManager class` is used for several things, including handling events, enabling/disabling plugins, and handling player permissions. Most of the time, you will use it to register event listeners. It has a `registerEvents` method that takes a `Listener` object and a `JavaPlugin` object as parameters. The only class that exists is both the `Listener` and `JavaPlugin`. So, we will pass the `this` object to both the parameters. If the `Listener` class is separated from the `main` class, then the line will look like the following line of code:

```
getServer().getPluginManager().registerEvents(
  new WeatherListener(), this);
```

This is all that is needed within the `onEnable` method.

## Listening for an event

The next method that we will create is an `EventHandler method`. We will use the `@EventHandler` annotation to tell Bukkit which methods are event listeners. Create a new method that has an event of our choice as the only parameter:

```
public void onWeatherChange(WeatherChangeEvent event)
```

The method must be `public`, and it should not return anything. You can name this method anything you wish, but most programmers will keep the name similar to the name of the event.

Next, we will indicate that this method handles events. Just above the method, add the following annotation:

```
@EventHandler
```

On the same line, we can modify some properties for the `EventHandler method`. A property that you are likely to add to all the `EventHandler` methods is the one that ignores the canceled events. Setting the `ignoreCancelled` property to `true` will result in the method looking like this:

```
@EventHandler (ignoreCancelled = true)
public void onWeatherChange(WeatherChangeEvent event) {
}
```

If the event is already canceled by another plugin, then we don't want to bother listening to it.

The other property is the event priority. By changing the priority of the `EventHandler method`, you can choose to listen for the event before or after other plugins. If the `EventHandler method` has a higher priority than another event, then it is called after the other `EventHandler method` and thus may override anything that the first `EventHandler` method has modified. There are six priority levels, and they are called in the following order:

1. LOWEST
2. LOW
3. NORMAL
4. HIGH
5. HIGHEST
6. MONITOR

Thus, the plugins with the `LOWEST` priority are called first. Imagine that you have a protection plugin. You would not want any other plugin to reverse its decision to cancel an event. Therefore, you would set the priority to `HIGHEST` so that no other plugins would be able to modify the event after yours. By default, each `EventHandler method` has a `NORMAL` priority. If you are not modifying the event, then you will most likely want to listen at the `MONITOR` level. The `MONITOR` priority should not be used when modifying the event, such as canceling it.

We want to cancel this event before plugins that have a `NORMAL` priority even see it. Therefore, let's change the priority of this event to `LOW`. Now, the line that is above the method looks like the following line of code:

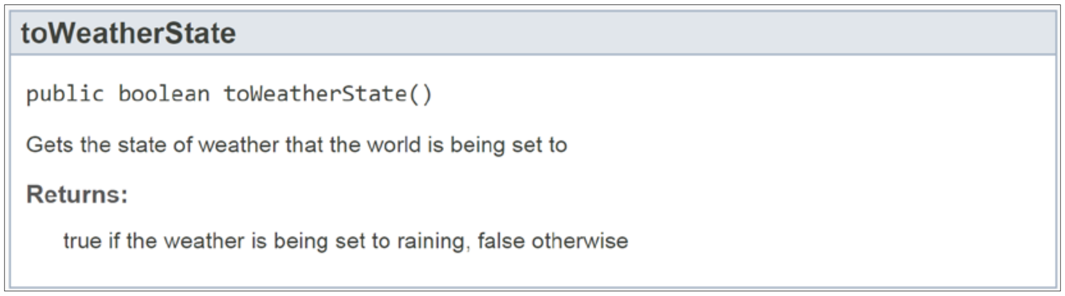
```
@EventHandler (ignoreCancelled = true, priority = EventPriority.LOW)
```

## Canceling an event

Finally, we want to stop the weather from changing. To do so, we will call the `setCancelled` method of the event. The method takes a `Boolean` value as a parameter. We want `canceled` to equal `true`. Therefore, we will use the `setCancelled(true)` code, which is as follows:

```
package com.codisimus.norain;

import org.bukkit.event.EventHandler;
import org.bukkit.event.EventPriority;
import org.bukkit.event.Listener;
import org.bukkit.event.weather.WeatherChangeEvent;
import org.bukkit.plugin.java.JavaPlugin;

public class NoRain extends JavaPlugin implements Listener {
  @Override
  public void onEnable() {
    getServer().getPluginManager().registerEvents(this, this);
  }

  @EventHandler (ignoreCancelled = true, priority =
     EventPriority.LOW)
  public void onWeatherChange(WeatherChangeEvent event) {
    event.setCancelled(true);
  }
}
```

This plugin will work as is. However, there is room for improvement. What if it is already raining in the server world? This plugin would prevent the rain from ever stopping. Let's add an `if` statement so that the `WeatherChangeEvent class` will only be canceled if the weather is starting. The event provides us with a method called `toWeatherState`, which returns a `Boolean` value. This method will return `true` or `false`, informing us about whether the weather is starting or stopping respectively. This is also made clear in the API documentation:



**toWeatherState**

```
public boolean toWeatherState()
```

Gets the state of weather that the world is being set to

**Returns:**

true if the weather is being set to raining, false otherwise

If `toWeatherState` returns `true`, then it is starting to rain. This is the case where we want to cancel the event. Now, let's write the same thing in Java, as follows:

```
if (event.toWeatherState()) {
  event.setCancelled(true);
}
```

After adding this `if` statement, you should test your plugin. Before installing the plugin, log on to your server and use the `/toggledownfall` command to make it rain. Once it is raining, install your newly created plugin and reload the server. At this point, it will still be raining, but you will be able to stop the rain by issuing the `/toggledownfall` command again. If you cannot do so, then the `if` statement that you added may be incorrect; review it to find your mistake and test it again. Once you stop the rain, you can try to use the same command to start the rain again. As long as the code is correct, the rain should not start. If the rain does start, then verify that your event listener is being properly registered within the `onEnable` method. Also, verify that the server is enabling the correct version of the plugin, as explained in *Chapter 4*, *Testing on the Spigot Server*.

## Communicating among events

The plugin works exactly as intended, but what if we have a change of heart and begin to miss the sound of rain? Alternatively, what if our town bursts into flames and it must be extinguished quickly? We do not want to limit our power as an administrator by denying ourselves the use of the `/toggledownfall` command. Next, we will listen for this command to be issued, and when it is issued, we will allow the weather to change. Ultimately, we will still be able to control the weather manually, but the weather will not start on its own.

Let's create another `EventHandler method`. This time, we will listen for a console command being sent so that we can set a Boolean flag, as follows:

```
@EventHandler (ignoreCancelled = true, priority = EventPriority.
MONITOR)
public void onPlayerCommand(PlayerCommandPreprocessEvent event) {
  //Check if the Player is attempting to change the weather
  if (event.getMessage().startsWith("/toggledownfall")) {
    //Verify that the Player has permission to change the weather
    if (event.getPlayer().hasPermission(
      "minecraft.command.toggledownfall")) {
      //Allow the Rain to start for this occasion
      denyRain = false;
    }
  }
}
```

We will not actually be modifying this event at all. Therefore, the event priority will be set to MONITOR. We also want to ignore canceled events. The event that we will listen for is PlayerCommandPreprocessEvent, which will occur every time a player issues a command, whether they are for Minecraft, Bukkit, or another plugin. We only care about one command, namely /toggledownfall. So, the first if statement checks whether the message starts with /toggledownfall. If it is a different command, we will ignore it. As the event name suggests, this event occurs before the command is actually executed. Therefore, we must verify that a player has the permission to run the command. The permission node for the command is minecraft.command.toggledownfall. If these two conditions are met, then we want to allow rain to start on the next WeatherChangeEvent class. The second EventHandler method is completed by using the two if statements and setting a Boolean variable to false.

At this point, a light bulb will appear, informing you that the denyRain symbol cannot be found. When you click on the bulb, you can select Create Field denyRain in com.codisimus.norain.NoRain. This will automatically create a private variable called denyRain inside the class. Note the placement of the new line of code. It is outside the existing method blocks and yet still inside the class. This is important because it defines the variable's scope. The scope of a variable is where it can be accessed. The denyRain variable is private. Therefore, no other class, such as a class from another plugin, can modify it. However, within the NoRain class, all the methods can access it. This is useful because if the variable was declared within the curly braces of the onPlayerCommand method, we would not be able to see it from the onWeatherChange method.

Now that the plugin knows when we wish to allow the rain to start, we must slightly modify the onWeatherChange method to allow for such an exception. Currently, to cancel the event, we will call the setCancelled method with true as the parameter. If we were to pass false as a parameter, then the event would not be cancelled. The denyRain variable is equal to true when we wish to cancel the event. Therefore, rather than passing true or false, we can pass the value of denyRain. So, when denyRain is set to false, we will call setCancelled using the following line of code:

```
event.setCancelled(false);
```

At the end of the onWeatherChange method, we want to reset the value of denyRain to true. In this way, we can ensure that we allow the weather to change only once each time the /toggledownfall command is issued. The final code looks like this:

```
package com.codisimus.norain;

import org.bukkit.event.EventHandler;
import org.bukkit.event.EventPriority;
import org.bukkit.event.Listener;
import org.bukkit.event.player.PlayerCommandPreprocessEvent;
import org.bukkit.event.weather.WeatherChangeEvent;
import org.bukkit.plugin.java.JavaPlugin;

public class NoRain extends JavaPlugin implements Listener {
  //This is a variable that our two methods will use to "communicate"
with each other
  private boolean denyRain = true;

  @Override
  public void onEnable() {
    //Register all of the EventHandlers within this class
    getServer().getPluginManager().registerEvents(this, this);
  }

  @EventHandler (ignoreCancelled = true, priority =
     EventPriority.LOW)
  public void onWeatherChange(WeatherChangeEvent event) {
    if (event.toWeatherState()) { //Rain is trying to turn on
      //Cancel the event if denyRain is set to true
      event.setCancelled(denyRain);
    }
    //Reset the denyRain value until next time a Player issues the /
toggledownfall command
    denyRain = true;
  }

  @EventHandler (ignoreCancelled = true, priority =
     EventPriority.MONITOR)
  public void onPlayerCommand
    (PlayerCommandPreprocessEvent event) {
    //Check if the Player is attempting to change the weather
    if (event.getMessage().startsWith("/toggledownfall")) {
      //Verify that the Player has permission to change the weather
      if (event.getPlayer().hasPermission
        ("minecraft.command.toggledownfall")) {
        //Allow the Rain to start for this occasion
        denyRain = false;
      }
    }
  }
}
```

Note that when we declare the Boolean `denyRain method`, we set its initial value to `true`.

This completes the `NoRain` plugin. Build the JAR file and test it out on your server. With this new version, you will be able to use the `/toggledownfall` command to stop and start rain.

## Modifying an event as it occurs

The Bukkit API allows a programmer to do more than just cancel an event. Depending on the event, you can modify many of its aspects. In this next project, we will modify zombies as they spawn. Every time a zombie spawns, we will give it `40` health rather than the default `20`. This will make zombies more difficult to kill.

Create a new project as you would for any plugin. We will call this plugin `MobEnhancer`. Similar to what we did with the `NoRain` plugin, have the `main` class implement `Listener` and add the following line of code to the `onEnable` method to register the `EventHandlers method`:

```
getServer().getPluginManager().registerEvents(this, this);
```

For this project, we will have an `EventHandler method` that listens for mobs spawning. This will be the `CreatureSpawnEvent class`. This event has many methods that we can call to either modify the event or gain more information about it. We only wish to modify zombies that are spawned. Therefore, the first thing that we will add is an `if` statement, which will check whether the `EntityType method` is `ZOMBIE`. This is done by using the following block of code:

```
if (event.getEntityType() == EntityType.ZOMBIE) {
}
```

Inside the curly braces, we will change the health of the `Entity class` to `40`. We can retrieve the `Entity` class by calling `event.getEntity()`. Once we have the `Entity class`, we have access to many additional methods. You can view all of these methods in the API documentation, which is available at `https://hub.spigotmc.org/javadocs/spigot/org/bukkit/entity/Entity.html`.

One of the methods is `setHealth`. Before we can set the health to `40`, we must set the maximum health, which can have a value of `40`. An `Entity class` cannot have a health of `40` when its maximum health is still `20`. These two lines of code will complete this plugin. The code now looks like this:

```
package com.codisimus.mobenhancer;

import org.bukkit.entity.EntityType;
import org.bukkit.event.EventHandler;
import org.bukkit.event.Listener;
import org.bukkit.event.entity.CreatureSpawnEvent;
import org.bukkit.plugin.java.JavaPlugin;

public class MobEnhancer extends JavaPlugin implements Listener {
  @Override
  public void onEnable() {
    //Register all of the EventHandlers within this class
    getServer().getPluginManager().registerEvents(this, this);
  }

  @EventHandler
  public void onMobSpawn(CreatureSpawnEvent event) {
    if (event.getEntityType() == EntityType.ZOMBIE) {
      int health = 40;
      event.getEntity().setMaxHealth(health);
      event.getEntity().setHealth(health);
    }
  }
}
```

The first version of the `MobEnhancer` plugin is complete with this small class. You can test the plugin by installing it on your server. You will notice that zombies will be much more difficult to kill.

> Note that we declared a local variable named `health` of the `int` type and set its value to `40`. Alternatively, we could simply write `40` in the two lines that follow. However, programming the amount of health this way allows us to easily change it in the future. We only have to change the number in one line of code rather than two or more. Also, you may have noticed that the `setMaxHealth` and `setHealth` methods accept a variable of the `double` type. However, an `int` value may still be passed to the method, as it will be automatically converted to a `double` value with a value of `40.0`.

You can add more code to the plugin in order to modify the health of more types of entities. A list of all the `EntityType methods` can be found in the Bukkit API documentation under the `EntityType` class reference page at `https://hub.spigotmc.org/javadocs/spigot/org/bukkit/entity/EntityType.html`. However, in the next chapter, we will make this plugin configurable in order to change the health of every type of `Entity` that spawns.

## Creating more plugins on your own

Now that you have created these two plugins, you have a hang of how to properly use event listeners. You now have the required knowledge to create hundreds of unique plugins on your own. All that you need to get started is a cool idea. Why don't you try making one of the plugins that was suggested earlier in this chapter? For more ideas, you know where to look. The Bukkit, Spigot, and Minecraft forums or the API documentation are great for inspiration. For example, looking through the list of events, I saw the `ExplosionPrimeEvent` class, which is described as "**Called when an entity has made a decision to explode**". This event is called when a creeper makes that hissing noise that every Minecraft player dreads. When this happens, you can send a message to all the nearby players to make it look like the creeper is talking to them. First, you will create an `EventHandler method` for this event. You will want to return in case the entity is not a creeper. Then, you will want to get the entities that are near the creeper (there is a method for this within the `Entity` class). For each entity that you get, if it is an instance of a player, send them a message, as follows:

```
<Creeper> That sssure isss a nicccee <ItemInHand> you have there. It
would be a ssssshame if anything happened to it.
```

In each message, you will replace `<ItemInHand>` with the type of item that the player is holding. By this time, I am sure that you have some ideas of your own that you are able to implement as well.

Another good thing that you should know about listeners is how to unregister them. You may never need to do this, but if you do ever want to stop modifying or canceling an event, then you can use the following code within the `Listener` class:

```
    HandlerList.unregisterAll(this);
```

This will unregister the entire class. So, if you wish to only unregister specific `EventHandler methods`, then you should split them up into separate classes. Unregistering the listeners will not be the way to go for the `NoRain` plugin, but it may be useful if you add a `/mobenhancer off` command. Then, a `/mobenhancer on` command can register the listeners again, which is similar to how we did this in the `onEnable` method.

## Summary

Both the plugins that we made in this chapter have the entire code within a single class. However, you may choose to separate these into the main plugin class and a listener class. In small plugins like these, it is not necessary. But in larger projects, it will keep your code much cleaner. There will be a few differences, such as having static variables or passing a variable to another class. In the next chapter, we will complete the `MobEnhancer` plugin by adding configuration as well as a `reload` command. We will have the `Listener` and `CommandExecutor` as a part of the `main` class. Once the plugin is complete, we will go over the differences for the same plugin as three individual classes.

# 8
# Making Your Plugin Configurable

A configurable plugin can be very powerful. A single plugin will be able to function in different ways, depending on user preferences. Essentially, your plugin's configuration file will be similar to the `bukkit.yml` file for your server. It will allow you to change settings for the plugin without modifying the Java code. This means that you need not rebuild the plugin JAR file every time you wish to change a small detail. If your plugin is public or used by someone else, adding a `config` file may reduce the time spent on modifying code in the future. The users of your plugin can change the settings that are in the `config` file by themselves and do not require any additional assistance from you as a developer.

To fully understand why we would want a variable to be configurable, let's look at one of the plugins that we previously talked about. In `MobEnhancer`, we set the health of zombies to `40` instead of `20`. Someone else may wish to use your plugin, but they want to set the zombies' health to `60`. You can create two versions of the plugin, which may become very confusing, or you can have one version that is configurable. In the `config` file on your server, you will have the health of zombies set to `40`. But on another server, the health will be set to `60`. Even if your plugin will be used on only one server, configuration will allow for a quick and easy method of changing the amount of health.

There are five steps to making your plugin configurable, which are as follows:

1. Decide exactly which aspects of your plugin will be configurable
2. Create a `config.yml` file that includes each setting and its default value
3. Add code to save the default `config` file as well as load/reload the file
4. Read the configured values and store them in your plugin as class variables
5. Ensure that your code references the class variables that the configuration settings are loaded into

The steps need not be performed in this order, but we will discuss them in the following order in this chapter:

- Configurable data types
- Writing a `config.yml` file
- Saving, loading, and reloading your plugin's configuration
- Reading values from the configuration
- Using the configured settings in your plugin
- Writing an `ItemStack` value in the YAML format
- Understanding the YAML structure and hierarchy
- Storing configuration values locally
- Splitting one class into multiple classes and accessing variables and methods from another class

## Configurable data types

You can easily make most variables in your plugin configurable. The following table comprises various data types and examples of why you may want them to be configurable:

| **Data Type** | **How It Can Be Used** |
|---|---|
| `int` | To define the number of times an event should occur |
| `double` | To set the health of a mob when it spawns |
| `boolean` | To turn a specific feature on or off |
| `String` | To change a message that is sent to a player |
| `ItemStack` | To make a customized item appear |

Adding an `ItemStack` value to a configuration file is complicated, but this will be explained towards the end of this chapter.

We are going to make `MobEnhancer` configurable. We want to give the players a choice of setting the value of the zombies' health. This will simply be one `double` value. Let's expand the plugin to support additional creature types. We will create the `config` file first and then adapt the program to be able to modify different types of mobs. Therefore, we have decided that the `config` file will include a single `double` data type value for each type of mob. This `double` value will be the mob's health.

## Writing a config.yml file

Now, it is time to start writing the `config.yml` file. Create a new `YAML` file in the default package of `MobEnhancer`. The name of this file must be `config.yml` in order for it to be properly loaded by Spigot. The following is an example of how the `config` file for `MobEnhancer` will appear. Note the comments in the example indicated by the `#` character. Remember to always include comments so that users know exactly what each setting is for:

```
#MobEnhancer Config
#Set the health of each Mob below
#1.0 is equal to half a heart so a Player has 20.0 health
#A value of -1.0 will disable modifying the mob's health
#Hostile
ZOMBIE: 20.0
SKELETON: 20.0

#Passive
COW: 10.0
PIG: 10.0
```

> Only a few mobs are included in this `config` file, but the names of all the mob types can be found in the API documentation for the `EntityType` class at `https://hub.spigotmc.org/javadocs/spigot/org/bukkit/entity/EntityType.html`.

This is a simple `YAML` file because it does not contain nested keys. Most of your configurations will be this simple, but we will go over some complicated ones later in this chapter.

## Saving, loading, and reloading the config file

Now that we have the `config.yml` file and it is located in the default package of the plugin, we need to be able to save it to a user's server. Once the file is saved, the user will be able to edit it as they please. Saving the `config` file is as simple as adding the following method call to the `onEnable` method, as follows:

```
saveDefaultConfig();
```

This will copy `config.yml` to `plugins/MobEnhancer/config.yml`. If the file already exists, then this line of code will do nothing.

The loading of the `config` file is done automatically by Spigot, and there is no need for you to do anything in addition to this in your plugin besides using `getConfig` when you actually want to access the configuration file.

Reloading `config.yml` is fairly simple to include; we will add it in the form of a command, as follows:

```
@Override
public boolean onCommand(CommandSender sender, Command command, String
alias, String[] args) {
    reloadConfig();
    sender.sendMessage("MobEnhancer config has been reloaded");
    return true; //The command was executed successfully
}
```

We will put this method inside the `main` class for now. Ensure that the class also implements `CommandExecutor`. Do not forget to register the command with the following line:

```
getCommand("mobenhancerreload").setExecutor(this);
```

The command should also be added to `plugin.yml`, as always. It is a good idea to add a permission node at this point too. The new `plugin.yml file` looks like this:

```
name: MobEnhancer
main: com.codisimus.mobenhancer.MobEnhancer
version: 0.2
description: Modifies Mobs as they spawn
commands:
  mobenhancerreload:
    description: Reloads the config.yml file of the plugin
    aliases: [mereload, merl]
    usage: /<command>
    permission: mobenhancer.rl
    permission-message: You do not have permission to do that
permissions:
  mobenhancer.rl:
    default: op
```

Now, your plugin will have a `reload` command. This means that when you edit `config.yml`, you can reload the plugin rather than restarting the entire server.

## Reading and storing the configured values

Once your configuration file is loaded, you must be able to access the file and read the values that are set. The `JavaPlugin` class, which is extended by the main class, has a `getConfig` method, which returns `FileConfiguration`. This `FileConfiguration` class is what we will use to get the values that we are looking for. You will see that the `FileConfiguration` class has methods such as `getInt`, `getDouble`, `getString`, and `getBoolean`; all of these methods take a string as a parameter. The `string` parameter is the path to the value. To fully understand the path, we need to look at a YAML configuration that contains nested keys. An example of this is the `plugin.yml` file that we were just working with. If we want to get the `MobEnhancer` string from the configuration, then the path will be `name`. If we want to retrieve the description of the `mobenhancerreload` command, then the path will be `commands.mobenhancerreload.description`. Therefore, the Java code needed to retrieve this value will be `getString("commands.mobenhancerreload.description");`. The `config.yml` file for `MobEnhancer` is quite simple. In order to get one of the double values, we can use the `getDouble()` method with the name of the mob as the path. For example, to get the value that is set for the `ZOMBIE Entity`, we will use the following code:

```
double health = this.getConfig().getDouble("ZOMBIE");
```

This will return a `double` value from one of the following three sources:

- The `FileConfiguration` that has been loaded from `plugins/MobEnhance/config.yml`
- The default `FileConfiguration`, which is the `config.yml` file that is located within the default package of the `MobEnhancer` JAR file
- The default value of the data type (`0` for a `double/integer` data type, `false` for a Boolean value, and `null` for a String/`ItemStack`)

The first result that doesn't fail will be returned. A result will fail due to an invalid path or an invalid value. In the previous statement, an invalid path will occur if the `ZOMBIE` path is not within `config.yml`. An invalid value will mean that the value of the given path is not a `double data type`.

Now that we understand how to read the configured data, let's modify the plugin to use these customized values.

## Using configured settings within your plugin

The current `EventHandler method` of the `MobEnhancer` plugin sets the health of zombies to `40`, where the number 40 is **hardcoded**. This means that the value of `40` is a part of the code itself, and this cannot be changed after the code is compiled. We wish to make this value **softcoded**, that is, we want to retrieve the value from an external source, which is `config.yml` in our case:

Currently, the `onMobSpawn` method is as follows:

```
@EventHandler
public void onMobSpawn(CreatureSpawnEvent event) {
    if (event.getEntityType() == EntityType.ZOMBIE) {
        int health = 40;
        event.getEntity().setMaxHealth(health);
        event.getEntity().setHealth(health);
    }
}
```

We will work from this existing code. The `if` statement is no longer needed, because we don't want to limit the plugin to zombies only. As discussed earlier, we also want to replace the hardcoded `40 value` with a `double` value, which will be read from the `config` file. Therefore, `40` should be replaced with `getConfig().getDouble(type)`. You will also have to change the variable type from `int` to `double`. The `Type` in this statement will be a string of the `Entity` type. Some examples of this are `ZOMBIE`, `SKELETON`, or any of the other entity types that are listed in `config.yml`. We already know that we can get the type of the entity that was spawned by using `event.getEntityType()`. However, this gives us `EntityType` in the `enum` form, and we require it in the `string` form. The `EntityType` page of the Bukkit API documentation informs us that we can call the `getname` method to return the string that we are looking for. The new `onMobSpawn` method is as follows:

```
@EventHandler
public void onMobSpawn(CreatureSpawnEvent event) {
    //Find the type of the Entity that spawned
    String type = event.getEntityType().name();

    //Retrieve the custom health amount for the EntityType
    //This will be 0 if the EntityType is not included in the config
    double health = getConfig().getDouble(type);
    event.getEntity().setMaxHealth(health);
    event.getEntity().setHealth(health);
}
```

This `EventHandler method` is nearly complete. We are allowing other people to set the `health` value. We want to ensure that they are entering a valid number. We don't want the plugin to crash because it is being misused. We know that we are receiving a `double` value because even if the user sets a non-numeric value, we will be given the default value of `0` instead. However, not every valid double value will be useable in our situation. For example, we cannot set the health of an entity to a negative value. We also do not want to set the health to `0`, because this will instantly kill the entity. Therefore, we should only modify the health if the new health is set to a positive number. This can be done with a simple `if` statement, as follows:

```
if (health > 0)
```

The `MobEnhancer` plugin is now configurable and supports any type of creature. It is no longer limited to just zombies. The finished code will be similar to the following:

```
package com.codisimus.mobenhancer;

import org.bukkit.command.Command;
import org.bukkit.command.CommandExecutor;
import org.bukkit.command.CommandSender;
import org.bukkit.event.EventHandler;
import org.bukkit.event.Listener;
import org.bukkit.event.entity.CreatureSpawnEvent;
import org.bukkit.plugin.java.JavaPlugin;

public class MobEnhancer extends JavaPlugin implements Listener,
CommandExecutor {
    @Override
    public void onEnable() {
        //Save the default config file if it does not already exist
        saveDefaultConfig();

        //Register all of the EventHandlers within this class
        getServer().getPluginManager().registerEvents(this, this);

        //Register this class as the Executor of the /merl command
        getCommand("mobenhancerreload").setExecutor(this);
    }

    @EventHandler
    public void onMobSpawn(CreatureSpawnEvent event) {
        //Find the type of the Entity that spawned
        String type = event.getEntityType().name();
```

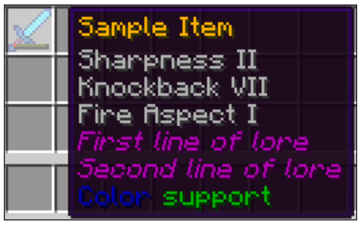
```
            //Retrieve the custom health amount for the EntityType
            //This will be 0 if the EntityType is not in the config
            double health = getConfig().getDouble(type);

            //Mobs cannot have negative health
            if (health > 0) {
                event.getEntity().setMaxHealth(health);
                event.getEntity().setHealth(health);
            }
        }

        @Override
        public boolean onCommand(CommandSender sender, Command command,
        String alias, String[] args) {
            reloadConfig();
            sender.sendMessage("MobEnhancer config has been reloaded");
            return true; //The command was executed successfully
        }
    }
```

## ItemStack within a configuration

Next, we will expand the `MobEnhancer` plugin even further by allowing the option of giving armor and weapons to zombies and skeletons. In order to do this, we must first learn how to add an `ItemStack` object as an option in a configuration file. An `ItemStack` method is more complicated than a simple integer or double. It is an object that has many nested values. It may also include a **meta** value, which will have more nested values. **Meta** contains additional information for the item, such as a custom display name or lines of text that make up the lore of the item. The following is a sample of an `ItemStack` method in a `YAML` file:

```
SampleItem:
  ==: org.bukkit.inventory.ItemStack
  type: DIAMOND_SWORD
  damage: 1500
  amount: 1
  meta:
    ==: ItemMeta
    meta-type: UNSPECIFIC
    display-name: §6Sample Item
    lore:
    - First line of lore
    - Second line of lore
    - §1Color §2support
    enchants:
      DAMAGE_ALL: 2
      KNOCKBACK: 7
      FIRE_ASPECT: 1
```

Once loaded, the item in the results is shown in the following screenshot:



Sample Item
Sharpness II
Knockback VII
Fire Aspect I
First line of lore
Second line of lore
Color support

Only the **type** field is required. You can omit any other segment. The **type** refers to the type of material. These materials can be found in the API documentation under `org.bukkit.Material`, which can be viewed by visiting `https://hub.spigotmc.org/javadocs/spigot/org/bukkit/Material.html`. The **damage** is used to indicate how much damage an item has taken. For items such as `wool`, this will set the color of the wool. The **amount** will set the stack size. For example, I may have one sword or twenty logs. The **meta** includes additional information such as the color and pattern of a banner or the author and number of pages of a book. Given the path, `getConfig().getItemStack("SampleItem");` will retrieve the item.

## YAML configuration hierarchy

Note the hierarchy when working with `ItemStack` in YAML. This is similar to how commands and permissions have nested values in the `plugin.yml` files. We can utilize a hierarchy within the `config` file to make it easier to use and understand.

We want to give items to two types of mobs, namely `Zombie` and `Skeleton`. Each type will have a unique armor and weapon. This means that we will need 10 different `ItemStack` classes. We can name them `ZombieHolding`, `SkeletonHolding`, `ZombieHelmet`, `SkeletonHelmet`, and so on. However, a hierarchy will be much more efficient. We will have a `Zombie` key and a `Skeleton` key. Within each zombie and skeleton, we will have a key for each item. The following is a sample of the hierarchy of the mob armor segment of the `config` file:

```
Zombie:
  holding:
    ==: org.bukkit.inventory.ItemStack
    type: STONE_SWORD
  helmet:
    ==: org.bukkit.inventory.ItemStack
    type: CHAINMAIL_HELMET

Skeleton:
  holding:
    ==: org.bukkit.inventory.ItemStack
    type: BOW
  helmet:
    ==: org.bukkit.inventory.ItemStack
    type: LEATHER_HELMET
```

The rest of the armor pieces can be added in the same way.

If we want to retrieve the `ItemStack` method for the boots of a skeleton, we will use `getConfig().getItemStack("Skeleton.boots");`. Remember that the hierarchy is conveyed using a period. Here is a section that will be appended to `config.yml`, which includes mob armor, as discussed. We also have a `GiveArmorToMobs` Boolean value, which will be included to easily disable the mob armor feature:

```
### MOB ARMOR ###
GiveArmorToMobs: true

Zombie:
  holding:
    ==: org.bukkit.inventory.ItemStack
    type: STONE_SWORD
  helmet:
    ==: org.bukkit.inventory.ItemStack
    type: CHAINMAIL_HELMET

Skeleton:
  holding:
    ==: org.bukkit.inventory.ItemStack
    type: BOW
    meta:
      ==: ItemMeta
      meta-type: UNSPECIFIC
      enchants:
        ARROW_FIRE: 1
  helmet:
    ==: org.bukkit.inventory.ItemStack
    type: LEATHER_HELMET
    color:
      ==: Color
      RED: 102
      BLUE: 51
      GREEN: 127
```

## Storing configuration values as variables

Retrieving a value from your plugin's `config` file requires more time and resources than is required to access a local variable. Therefore, if you will be accessing a specific value very often, it is best to store it as a variable. We will want to do just this with the `GiveArmorToMobs` Boolean value. It is also a good idea to store the `ItemStack` armor locally to prevent creating a new one every time it is used. Let's add the following variables above the methods of the main class:

```
private boolean giveArmorToMobs;
private ItemStack zombieHolding;
private ItemStack skeletonHolding;
```

We will only write the code to set the item that a zombie or skeleton is holding. You can add the rest of the armor yourself, as it will be done the same way.

We want these values to be automatically stored whenever the `config` file is reloaded. Note that when the `config` file is initially loaded, it is actually being reloaded. To ensure that our data is saved every time the `config` file is reloaded, we will add additional code to the `reloadConfig` method of the plugin. This is the method that we call to execute the `/merl` command. The `reloadConfig` method is already included in every Java plugin, but we will modify it by overriding it. This is a lot like how we override the `onEnable` method. Overriding a method will prevent the existing code from being executed. This is not an issue for `onEnable`, because the method has no prior existing code. However, `reloadConfig` has the code that we still wish to execute. Therefore, we will use the following line of code to execute the existing code that we are overriding:

```
super.reloadConfig();
```

This line of code is very important. Once we have it, we can add our own code before or after it. In our case, we want to store the values after the `config` file has been reloaded. Therefore, the additional code should be placed after the preceding line of code. The completed overridden `reloadConfig` method looks like this:

```
/**
 * Reloads the config from the config.yml file
 * Loads values from the newly loaded config
 * This method is automatically called when the plugin is enabled
 */
```

```
@Override
public void reloadConfig() {
    //Reload the config as this method would normally do
    super.reloadConfig();

    //Load values from the config now that it has been reloaded
    giveArmorToMobs = getConfig().getBoolean("GiveArmorToMobs");
    zombieHolding = getConfig().getItemStack("Zombie.holding");
    skeletonHolding = getConfig().getItemStack("Skeleton.holding");
}
```

The last code that we must write is to give armor to specific mobs. We will add this to the end of the `onMobSpawn` method. We only want to do this if `giveArmorToMobs` is set to `true`. Therefore, the block of code will be placed inside an `if` statement, as follows:

```
if (giveArmorToMobs) {

}
```

We can retrieve the entity's armor using the following code:

```
EntityEquipment equipment = event.getEntity().getEquipment();
```

This gives us their equipment slots even though they may not include anything in them at the moment. To know more about this object and what you can do with it, visit its API documentation at `https://hub.spigotmc.org/javadocs/spigot/org/bukkit/inventory/EntityEquipment.html`. Now that we have `EntityEquipment`, setting the pieces of armor is simple.

We have two distinct sets of armor. Therefore, we must first check whether the entity is either a zombie, or a skeleton. We can do this by using an `if/else` statement:

```
if (event.getEntityType() == EntityType.ZOMBIE) {
    //TODO - Give Zombie armor
} else if (event.getEntityType() == EntityType.SKELETON) {
    //TODO – Give Skeleton armor
}
```

However, using a `switch/case` block will be more efficient. Using `switch/case` in this scenario will look like this:

```
switch (event.getEntityType()) {
case ZOMBIE:
    //TODO - Give Zombie armor
    break;
case SKELETON:
    //TODO - Give Skeleton armor
    break;
default: //Any other EntityType
    //Do nothing
    break;
}
```

The `If/else` statements are used to check multiple conditions (*Is the entity a zombie?* or *Is the entity a skeleton?*). A `switch/case statement` saves time by asking a single question (*Which of the following is the type of the entity?*). The code within the correct `case` condition will then be executed. When a `break` condition is fulfilled, the `switch` statement is exited. If you do not end the case with `break`, then you will fall through to the next case and begin executing that code. In some circumstances, that is a good thing, but we do not want that to happen here. The default case, that is, if none of the other cases match, does not need to be included because there is no code in it. However, it does make the statement more conclusive, and the Java coding standards released by Oracle state that the default case should always be included.

Within each of these cases, we will want to equip the correct set of armor.

We should check each piece of armor to ensure that it is not `null` before applying it using the following code. This will prevent the plugin from crashing due to an invalid configuration:

```
if (zombieHolding != null) {
    equipment.setItemInHand(zombieHolding.clone());
}
```

We used the `clone` method here on the `ItemStack`. We don't want to hand out a single `ItemStack class` to every mob. Instead, we will create its clones so that each mob can have its own copy.

Equipping the remaining armor and equipping an armor to a skeleton is very similar. The overall block of code will look like this:

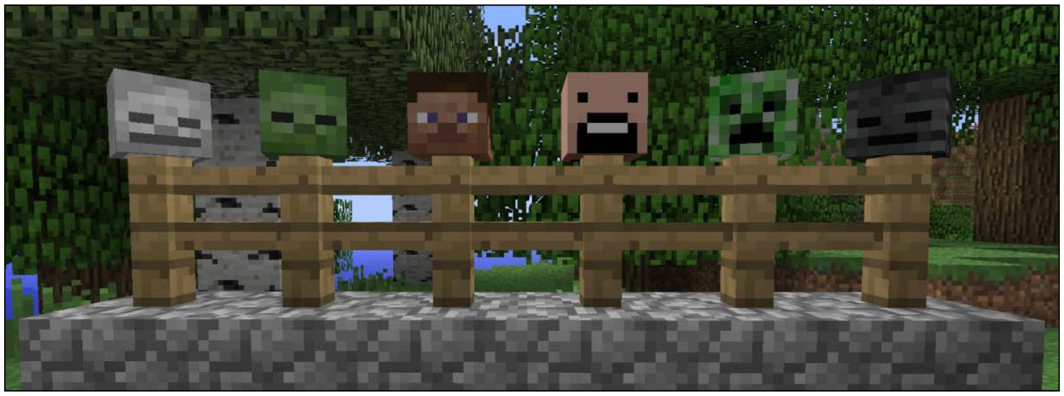
```
if (giveArmorToMobs) {
    //Retrieve the equipment object of the Entity
    EntityEquipment equipment = event.getEntity().getEquipment();

    switch (event.getEntityType()) {
    case ZOMBIE:
        //Set each piece of equipment if they are not null
```

```
            if (zombieHolding != null) {
                equipment.setItemInHand(zombieHolding.clone());
            }
            //TODO - Add rest of armor
            break;

        case SKELETON:
            //Set each piece of equipment if they are not null
            if (skeletonHolding != null) {
                equipment.setItemInHand(skeletonHolding.clone());
            }
            //TODO - Add rest of armor
            break;

        default: //Any other EntityType
            //Do nothing
            break;
        }
    }
```

The `clone` method should be called on each `ItemStack` class so that the original items remain undamaged.

With this, the `MobEnhancer` plugin now supports giving armor to mobs. Try it out on your server to see how it works. We only discussed giving armor to zombies and skeletons because most mobs, including creepers, spiders, and cows, cannot wear armor. If you want, try adding armor and items to other mobs to see what happens. Also, try giving mobs unique items. For example, skeletons can be given a sword or zombies can be given a bow. There is also a skull item that comes in different looks; you can make a mob wear it as a mask.

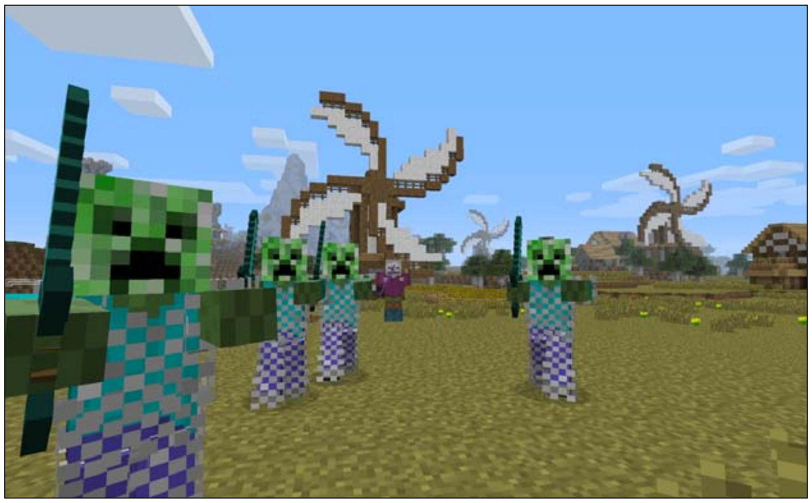
You can even create skulls that represent a specific player, such as Notch, as shown in the following screenshot:



The meta for the `NotchSkull` item is as follows:

```
NotchSkull:
  ==: org.bukkit.inventory.ItemStack
  type: SKULL_ITEM
  damage: 3
  meta:
    ==: ItemMeta
    meta-type: SKULL
    skull-owner: Notch
```

Play around with your new plugin to see what crazy items you can give to zombies and other mobs. The following screenshot illustrates an example of what you can accomplish by modifying the configuration:



## Accessing variables from another class

The `MobEnhancer` class is growing in size. There is no need to place all the code within a single class. This class is currently extending the `JavaPlugin` class as well as implementing both the `Listener` and `CommandExecutor` interfaces. The program will be easier to understand if we split these into three unique classes. This process is known as **refactoring**. Throughout the process of developing software, you will come across code that may be outdated or inefficient and needs to be updated. Changing the code in this way is referred to as **refactoring**. Don't be discouraged if you need to refactor your code in the future; it is a common occurrence in software development, and there are many reasons for it to happen.

- You learned how to write more efficient code
- API changes or new features require/allow code changes
- The existing code is difficult to read or debug
- A method/class has grown too large to manage
- The purpose of the code has changed and it should now do something it was not originally intended to do

We will refactor `MobEnhancer` to split the code into three more manageable classes.

Create two new classes named `MobSpawnListener` and `MobEnhancerReloadCommand`. `MobEnhancer` will still be your main class. Therefore, it will still extend `JavaPlugin`. However, the two new classes will implement `Listener` and `CommandExecutor` respectively. Move the appropriate methods to their new classes, that is, `onMobSpawn` is an event handler and hence it belongs to the `Listener` class, and `onCommand` belongs to the `CommandExecutor` class. When moving the methods, you will see that several errors are introduced. This is because your methods no longer have access to the necessary methods and variables. Let's first address the `MobEnhancerReloadCommand` class, as it has only one error. This error occurs at the following line:

```
reloadConfig();
```

The `reloadConfig` method is in the `JavaPlugin` class, which is no longer merged with the `CommandExector` class. We need to access the `JavaPlugin` object from this separate class. The easiest way to do this is by using a static variable. If a variable or method is static, then it does not change across different instances of the class. This allows us to refer to the variable from a static context. You have done this before when using the `Bukkit` class. The methods that you called were static. Therefore, you could access them using the `Bukkit` class and not a unique `Bukkit` object.

To explain this better, let's imagine that you have a plugin that gives Minecraft players bank accounts. Therefore, you will have a class to represent a player's bank account. This class can be called `PlayerAccount`. You will have numerous `PlayerAccount` objects, one for each player on the server. Within this class, you may have a variable that defines a limit of how much money the account can hold. Let's name this variable `accountLimit`. If we want each account to have a maximum amount of money of `1000`, then the `accountLimit` should be static. If we wish to increase the limit to `2000`, then we set `accountLimit` to `2000` by using `PlayerAccount.accountLimit = 2000;`. Then, all the players now have an account limit of `2000`. If we want some players to have a limit of `1000` and others to have a limit of `2000`, then we should not use a static variable. Without `accountLimit` being static, if we set `accountLimit` to `2000` for instance A of `PlayerAccount`, it would still be `1000` for instance B of `PlayerAccount`.

Storing the plugin as a static variable within the main class will benefit us. Above your current variables, add a `static JavaPlugin` variable named `plugin`, as follows:

```
public class MobEnhancer extends JavaPlugin {
    //Static plugin reference to allow access from other classes.
    static JavaPlugin plugin;
```

We must also instantiate this variable within the `onEnable` method. This can simply be done using `plugin = this;`. Now, we can access the plugin instance by using `MobEnhancer.plugin`. Therefore, where we previously had `reloadConfig();`, we will now have `MobEnhancer.plugin.reloadConfig()`. This will fix the errors in `MobEnhancerReloadCommand`:

```
package com.codisimus.mobenhancer;

import org.bukkit.command.Command;
import org.bukkit.command.CommandExecutor;
import org.bukkit.command.CommandSender;

public class MobEnhancerReloadCommand implements CommandExecutor {
    @Override
    public boolean onCommand(CommandSender sender, Command command,
    String alias, String[] args) {
        MobEnhancer.plugin.reloadConfig();
        sender.sendMessage("MobEnhancer config has been reloaded");
        return true; //The command executed successfully
    }
}
```

MobSpawnListener requires a similar modification, as the plugin object is needed to call the getConfig method.

You will continue seeing errors in MobSpawnListener. It is attempting to access variables that are still in the main class. Let's move the mob armor variables to the Listener class, as follows:

```
public class MobSpawnListener implements Listener {
    private boolean giveArmorToMobs;
    private ItemStack zombieHolding;
    private ItemStack skeletonHolding;
```

We must also modify the reload method within MobEnhancer.java to match the new location of the variables. For example, instead of giveArmorToMobs, we should now have MobSpawnListener.giveArmorToMobs:

```
public void reloadConfig() {
    //Reload the config as this method would normally do
    super.reloadConfig();

    //Load values from the config now that it has been reloaded
    MobSpawnListener.giveArmorToMobs = getConfig().
    getBoolean("GiveArmorToMobs");
    MobSpawnListener.zombieHolding = getConfig().getItemStack("Zombie.
    holding");
    MobSpawnListener.skeletonHolding = getConfig().
    getItemStack("Skeleton.holding");
}
```

Even with this change, we will still be given an error, which reads giveArmorToMobs **has private access in** MobSpawnListener. Each variable is private, which means that they cannot be accessed from another class. We wish to be able to access them from the other classes. Hence, we will remove the private modifier. After doing so, we will be given yet another error. This new error reads **non-static variable** giveArmorToMobs **cannot be referenced from a static context**. This happens because the variables are not defined as static variables. Before you simply change these variables so that they can be static, ensure that it makes sense for them to be static. Refer to the earlier discussion about when static variables should be used. In this situation, we will only have one value of each of these variables. Hence, we do want to make them static, as shown in the following code:

```
public class MobSpawnListener implements Listener {
    static boolean giveArmorToMobs;
    static ItemStack zombieHolding;
    static ItemStack skeletonHolding;
```

There are only two lines remaining in the code that require our attention. These two lines are used to register the event listener and command executor. When calling the `registerEvents` method, two parameters are required. The first parameter is `Listener`, and the second one is `Plugin`. The `this` keyword references the plugin. Therefore, it is fine as the second parameter. However, for the first parameter, you must pass an instance of the `Listener` class. We have done this in *Chapter 7*, *The Bukkit Event System*, when creating the `NoRain` plugin. The same applies to the command executor. We must pass an instance of the `MobEnhancerReloadCommand` class:

```
//Register all of the EventHandlers
getServer().getPluginManager().registerEvents(new MobSpawnListener(),
this);

//Register the Executor of the /mobenhancerreload command
getCommand("mobenhancerreload").setExecutor(new
MobEnhancerReloadCommand());
```

This gets rid of all the errors that resulted from splitting the project into multiple classes.

## Summary

You are now familiar with using a `YAML` configuration file. You can load custom values from a `config.yml` file and use them within the plugin. Doing so will greatly expand your ability to create unique projects that will be beneficial to multiple server administrators. Try adding configurable options to some of your previous projects. For instance, if you created the plugin that sends a message when a creeper is about to explode, add a configuration file to set the area within which the players must be in order to see the message. Now that you are introduced to `FileConfiguration` which can be used with the Bukkit API, in the next chapter, we will save the plugin's data using the same `FileConfiguration` method so that we can load it the next time the plugin is enabled.

# 9
# Saving Your Data

There are many types of Bukkit plugins. Some of them require you to save data. By saving data, I am referring to saving information to the system's hard drive. This is needed if the information must stay intact, even after the server restarts. At this point, none of the plugins that we created have this requirement. Examples of plugins that will save data are as follows:

- Economy plugins must save information about how much money each player has
- Land protection plugins must save information about which plots of land are claimed and who their owner is
- Questing plugins must store all the information for each quest, such as who has completed it

There are countless uses for saving data when a server is shut down. In this chapter, we will create a teleportation plugin that saves various warp locations to a file. Again, we will save these locations to a file so that we do not need to create them again after the server shuts down. You are already familiar with the YAML file format. Therefore, we will utilize the YAML configuration to save and load data. In this chapter, we will cover the following topics:

- The type of data that you can save
- The data in a plugin that is worth saving and the frequency of saving it
- Expanding a prewritten teleportation plugin
- Creating and using a `ConfigurationSerializable` object
- Saving data in a YAML configuration
- Loading the saved data from the YAML configuration

## Types of data that can be saved

You may recall, as discussed in the previous chapter, that only certain data types can be stored in a YAML file. These include primitive types, such as `int` and `boolean`, strings, lists, and types that implement `ConfigurationSerializable`, such as `ItemStack`.

For this reason, we can only save these specific types of data.

You may find yourself wanting to save other types of data such as a `Player` object, or in the case of the teleportation plugin, a `Location` object. These may not be stored directly, but they can usually be broken down in order to save the important values that are needed to load it later. As an example, you cannot save a `Player` object, but you can save the players' **UUID** (**Universal Unique Identifier**), which can be converted into a string. Each `Player` has one UUID. Hence, it is the only information that we need to be able to refer to that specific player later.

Storing a Players name is not an adequate solution, since the name provided in the Minecraft account can be changed.

A `Location` object also cannot be stored directly, but it can be broken down to its world, `x`, `y`, and `z` coordinates, `yaw`, and `pitch`. The `x, y, z, yaw`, and `pitch` values are simply numbers that can be stored. As for the world, it also has a UUID that will never change. Therefore, a location is broken down into one string (`world uuid`), three doubles (`x`, `y`, `z`), and two floats (`yaw` and `pitch`).

As you create your own plugins, you may have classes that you wish to store in a file, such as a `BankAccount` object. As mentioned earlier, we can do this with any class that implements `ConfigurationSerializable`. `ConfigurationSerializable` means that the object can be translated to a form that can be stored within a configuration. This configuration can then be written to a file. In the teleportation plugin, we will create a `location` object that does exactly this.

## Which data to save and when

We know what can be saved to a file, but what should we save? Writing data to a file uses disk space. Therefore, we want to save only what we need to. It is better to think, "Which information do I want to keep after the server shuts down?" For example, a banking plugin will want to keep the balance of each account. As another example, a **PvP** arena plugin will not care about having information about an arena match. It is quite likely that the match will simply be canceled as the server is shutting down. When considering the teleportation plugin, we will want to still have the locations of each warp after the server is shut down.

Our next concern is when to save this information. Writing data to files has the potential to **lag** a server if it is a large amount of data. If you are unfamiliar with the term "lag", it is a phrase that is used to indicate that the server is running slowly. You know when this happens because the game becomes very choppy and players and mobs seem to move around sporadically. This is an unpleasant experience for everyone. Hence, you want to save your data only when you have to. There are three typical options for how often you want to save your data:

- Every time the data is modified
- Periodically, such as every hour
- When the server/plugin is shut down

These options are ordered by how safe they are. For instance, if your data is saved only when the server is shut down, then you run the risk of losing unsaved data in case the server crashes. If data is saved every hour, then in the worst case, you will lose only one hour's worth of data. For this reason, the first option should always be used when possible. The second and third options should be considered only if the plugin handles a large amount of data and/or the data is modified very often, such as several times every minute. The data of the teleportation plugin will only be modified when someone creates/deletes a warp or sets their home warp location. Therefore, we will invoke the `save` method every time the data is modified.

## A sample teleportation plugin

For this project, you will be given an incomplete teleportation plugin. You already know how to program most of this project. Therefore, we will discuss only the following three topics:

- Creating a class that implements `ConfigurationSerializable`
- The `save` method
- The `load` method

The rest of the plugin is provided and can be downloaded from `www.packtpub.com`, as mentioned in the preface. The code that you will be working on is Version 0.1 of the plugin Warper. Go through the plugin and read the comments to try to understand everything that it does. Both `Maps` and `try/catch` blocks are used in this project. If you do not know what either of those are, that is okay. They will be explained when it is time to use them yourself. Note that the `SerializableLocation` class is the location class, which implements `ConfigurationSerializable`; we will discuss this next.

## Writing a ConfigurationSerializable class

**Serialization** is the process of translating data or objects into a form that can be written to a file. In the plugin Warper, we will need to save the Bukkit locations. Locations cannot be serialized themselves. Therefore, we will make our own class that holds the Bukkit `Location` object data and is able to convert it to and from a map. If you are new to maps, they are a very useful type of collection that we will use throughout this project. Maps have keys and values. Each key points to a specific value. The `Warper` plugin is a good example of how maps can be used. When teleporting, a player will choose a specific location to warp to by name. If all the warp locations were in a list, we would have to iterate through the list until the warp location with the correct name was found. With a map, we can pass the key (the name of the warp) to the map, and it will return the value (the warp location).

Create a new class called `SerializableLocation`, which contains a private variable that holds the Bukkit `Location object`. The first constructor will require a `Location` object. We will also include a `getLocation` method. The following code is how the beginning of the new class looks:

```
package com.codisimus.warper;

import org.bukkit.Location;

/**
 * A SerializableLocation represents a Bukkit Location object
 * This class is configuration serializable so that it may be
 * stored using Bukkit's configuration API
 */
public class SerializableLocation {
    private Location loc;

    public SerializableLocation(Location loc) {
        this.loc = loc;
    }
```

```
        /**
         * Returns the Location object in its full form
         *
         * @return The location of this object
         */
        public Location getLocation() {
            return loc;
        }
    }
```

Once you add `implements ConfigurationSerializable`, your IDE should warn you about implementing all the abstract methods. The method that you must override is `serialize`. This will return a map representation of your object. We already mentioned each piece of data that we will need. Now, we just have to assign each piece of data a name and put it in a map. To add data to a map, you can call the `put` method. This method requires two parameters, namely a key and a value of the key. A key is simply a name for the piece of data that allows us to reference it later. The value is the serializable data. To find out more about maps, you can read the **Javadoc** at `https://docs.oracle.com/javase/8/docs/api/java/util/Map.html`. For the `serialize` method, we will need to get all the data that we mentioned earlier and put it in a map, as follows:

```
    /**
     * Returns a map representation of this object for use of
    serialization
     *
     * @return This location as a map of Strings to Objects
     */
    @Override
    public Map<String, Object> serialize() {
        Map map = new TreeMap();
        map.put("world", loc.getWorld().getUID().toString());
        map.put("x", loc.getX());
        map.put("y", loc.getY());
        map.put("z", loc.getZ());
        map.put("yaw", loc.getYaw());
        map.put("pitch", loc.getPitch());
        return map;
    }
```

This handles the saving portion, but we still have to handle the loading. The simplest way to do this is by adding a constructor that takes the map as a parameter, as follows:

```
/**
 * This constructor is used by Bukkit to create this object
 *
 * @param map The map which matches the return value of the
serialize() method
 */
public SerializableLocation(Map<String, Object> map) {
    //Check if the world for this location is loaded
    UUID uuid = UUID.fromString((String) map.get("world"));
    World world = Bukkit.getWorld(uuid);
    if (world != null) {
        //Each coordinate we cast to it's original type
        double x = (double) map.get("x");
        double y = (double) map.get("y");
        double z = (double) map.get("z");

        //Both yaw and pitch are loaded as type Double and then
converted to float
        float yaw = ((Double) map.get("yaw")).floatValue();
        float pitch = ((Double) map.get("pitch")).floatValue();

        loc = new Location(world, x, y, z, yaw, pitch);
    } else {
        Warper.plugin.getLogger().severe("Invalid location, most
likely due to missing world");
    }
}
```

Loading is essentially the opposite of saving. We pull each value from the map and then use it to create the Bukkit `Location` object. As a safeguard, we will first verify that the world is actually loaded. If the world is not loaded, the location will not exist. We do not want the plugin to crash because of this. There is also no reason why you need to try loading the location of a nonexistent world, because no one will be able to teleport to it anyway.

Each object that you get from the map will have to be cast to its original type, which was done in the previous code. The `float` values are an exceptional case. Each of the `float` values will be read as a `double` value. The `double value` is similar to `float`, but it is more precise. Therefore, loading the `float` values as the `double` values and then converting them will not cause data loss.

Both of these methods will be used by Bukkit. As a programmer, you will only have to store this object in the YAML configuration. This can be done by simply using the following line of code:

```
config.set("location", serializableLoc);
```

Then, you can retrieve the data later by using the following code:

```
SerializableLocation loc = (SerializableLocation)config.
get("location");
```

Bukkit uses the `serialize` method and the constructor to handle the rest.

The class name and path are used to reference this class. To see an example of this, take a look at the `ItemStack` object in the `config.yml` file for the `MobEnhancer` plugin. An example of this class has also been provided:

```
==: com.codisimus.warper.SerializableLocation
```

The path will of course have your own namespace, not `com.codisimus`.

This works fine, but it may cause confusion, especially with long pathnames. However, there is a way to ask Bukkit to reference this class by using an alias. Perform the following two steps to complete this:

1. Add the `@SerializableAs` annotation just above the class, as follows:

   ```
   @SerializableAs("WarperLocation")
   public class SerializableLocation implements
   ConfigurationSerializable {
   ```

2. Register your class within the `ConfigurationSerialization class`:

   ```
   ConfigurationSerialization.registerClass(SerializableLocation.
   class, "WarperLocation");
   ```

This can be done in the `onEnable` method. Just ensure that it is executed before you attempt loading the data.

The serializable name must be unique. Therefore, it is better to include your plugin name rather than simply `Location`. That way, you can have a serializable location for another plugin without them conflicting.

## Saving data to a YAML configuration

Now, we are ready to complete the `save` method. We want to save data to a YAML file, much like how we did in `config.yml`. However, we do not want to save it to `config.yml`, because that serves a different purpose. The first thing that we will need to do is create a new YAML configuration, as follows:

```
YamlConfiguration config = new YamlConfiguration();
```

Next, we will store all the information that we wish to save. This is done by setting objects to specific paths, as follows:

```
config.set(String path, Object value);
```

The acceptable types for `value` were mentioned earlier in this chapter. In the teleportation plugin, we have maps, which contain the `SerializableLocation` method. Maps can be added to a YAML configuration as long as they are a map of strings to an object that is `ConfigurationSerializable`. **Hashmaps** are added to a configuration in a different manner. You must create a configuration section using the map.

The following code shows how we will add the teleportation data to the configuration:

```
config.createSection("homes", homes);
config.createSection("warps", warps);
```

Once all the data is stored, all that is left to do is write the configuration to the `save` file. This is done by invoking the `save` method on `config` and passing the file that we wish to use. Calling the `getDataFolder` method of the plugin will give us the directory in which we should store all the plugin data. This is also where `config.yml` is located. We can use this directory to reference the file in which we will save the data, as follows:

```
File file = new File(plugin.getDataFolder(), "warps.yml");
config.save(file);
```

We will put all of these lines of code inside a `try` block to catch an exception that may occur. If you don't already know about exceptions, they are thrown when there is some sort of error or when something unexpected occurs. A `try/catch` block can be used to prevent the error from causing your plugin to crash. In this case, an exception is thrown if the specified file cannot be written for some reason. This reason may be that the user has insufficient privileges or the file location cannot be found. Therefore, the `save` method with the `try` block is as follows:

```
/**
 * Saves our HashMaps of warp locations so that they may be loaded
 */
private static void save() {
    try {
        //Create a new YAML configuration
        YamlConfiguration config = new YamlConfiguration();

        //Add each of our hashmaps to the config by creating sections
        config.createSection("homes", homes);
        config.createSection("warps", warps);

        //Write the configuration to our save file
        config.save(new File(plugin.getDataFolder(), "warps.yml"));
    } catch (Exception saveFailed) {
        plugin.getLogger().log(Level.SEVERE, "Save Failed!",
saveFailed);
    }
}
```

The following is a sample `warps.yml` file that will be created using the Warper plugin:

```
homes:
  18d6a045-cd24-451b-8e2e-b3fe09df46d3:
    ==: WarperLocation
    pitch: 6.1500483
    world: 89fd34ff-2c01-4d47-91c4-fa5d1e9fdb81
    x: -446.45572804715306
    y: 64.0
    yaw: 273.74963
    z: 224.9827566893271
warps:
  spawn:
    ==: WarperLocation
    pitch: 9.450012
```

```
      world: 89fd34ff-2c01-4d47-91c4-fa5d1e9fdb81
      x: -162.47507312961542
      y: 69.0
      yaw: -1.8000238
      z: 259.70096111857805
    Jungle:
      ==: WarperLocation
      pitch: 7.500037
      world: 35dafe89-3451-4c27-a626-3464e3856428
      x: -223.87850735096316
      y: 74.0
      yaw: 87.60001
      z: 382.482006630207
    frozen_lake:
      ==: WarperLocation
      pitch: 16.200054
      world: 53e7fab9-5f95-4e25-99d1-adce40d5447c
      x: -339.3448071127722
      y: 63.0
      yaw: 332.84973
      z: 257.9509874720554
```

## Loading data from a YAML configuration

Now that the `save` method is complete, we are ready to write the `load` method. You are already familiar with loading data using the Bukkit configuration API. What we'll do now is similar to retrieving values from `config.yml`, as discussed in the previous chapter. However, we must first manually load the configuration using the following code, which will be different. We should only do this if the file actually exists. The file will not exist the first time the plugin is used. Therefore, we do not want an error to occur in that situation:

```
File file = new File(plugin.getDataFolder(), "warps.yml");
if (file.exists()) {
    YamlConfiguration config = new YamlConfiguration();
    config.load(file);
```

Now that we have the YAML configuration loaded, we can get values from it. The data has been placed into two unique configuration sections. We will loop through each key of both the sections in order to load all the locations. To get a specific object from a section, all that we need to do is call the `get` method and cast it to the correct object. You can see how this is done in the completed `load` method:

```
/**
 * Loads warp names/locations from warps.yml
 * 'warp' refers to both homes and public warps
 */
```

```
private static void load() {
    try {
        //Ensure that the file exists before attempting to load it
        File file = new File(plugin.getDataFolder(), "warps.yml");
        if (file.exists()) {
            //Load the file as a YAML Configuration
            YamlConfiguration config = new YamlConfiguration();
            config.load(file);

            //Get the homes section which is our saved hash map
            //Each key is the uuid of the Player
            //Each value is the location of their home
            ConfigurationSection section = config.getConfigurationSec
            tion("homes");
            for (String key: section.getKeys(false)) {
                //Get the location for each key
                SerializableLocation loc = (SerializableLocation)
                section.get(key);
                //Only add the warp location if it is valid
                if (loc.getLocation() != null) {
                    homes.put(key, loc);
                }
            }

            //Get the warps section which is our saved hash map
            //Each key is the name of the warp
            //Each value is the warp location
            section = config.getConfigurationSection("warps");
            for (String key: section.getKeys(false)) {
                //Get the location for each key
                SerializableLocation loc = (SerializableLocation)
                section.get(key);
                //Only add the warp location if it is valid
                if (loc.getLocation() != null) {
                    warps.put(key, loc);
                }
            }
        }
    } catch (Exception loadFailed) {
        plugin.getLogger().log(Level.SEVERE, "Load Failed!",
        loadFailed);
    }
}
```

Now that the plugin is complete, you can test it on your server. Set a home location as well as some warp locations and then view the save file. Stop and then start the server again to verify that the plugin does indeed load the correct data.

## Summary

The plugin that we created in this chapter invokes the `save` method whenever the data is modified. In the next chapter, you will learn how to periodically save data. If you wish to save data when the server is shut down, simply call the `save` method from the `onDisable` method of the plugin's `main` class. You can use your programming skills to expand this plugin. You can add permission nodes, which is done by simply adding them to `plugin.yml`. You can also add a `config.yml` file to modify messages or perhaps the amount of time that needs to be set for the upcoming warp delay. If you wish to incorporate a listener, you can listen for a `PlayerRespawnEvent`. Then, you can set a player's respawn location to their home. There are countless more ways to customize this plugin to your liking. Many teleportation plugins use a warp delay to prevent players from teleporting away from a fight. In the next chapter, we will expand this project by adding a warp delay using the Bukkit scheduler.

# 10
# The Bukkit Scheduler

The Bukkit scheduler is a very powerful tool, and it is easy to learn how to use it. It allows you to create repetitive tasks such as saving data. It also allows you to delay how long until a block of code is executed. The Bukkit scheduler can also be used to asynchronously compute lengthy tasks. This means that a task such as writing data to a file or downloading a file to the server can be scheduled to run on a separate thread to prevent the main thread, and thus the game, from lagging. In this chapter, you will learn how to do each of these by continuing to work on the `Warper` teleportation plugin, as well as creating a new plugin called `AlwaysDay`. This new plugin will ensure that it is always daytime on the server by repeatedly setting the time to noon. This chapter will cover the following topics:

- Creating a `BukkitRunnable` class
- Understanding synchronous and asynchronous tasks and when they should be used
- Running a task from a `BukkitRunnable` class
- Scheduling a delayed task from a `BukkitRunnable` class
- Scheduling a repeating task from a `BukkitRunnable` class
- Writing a plugin called `AlwaysDay` that uses a repeating task
- Adding a delayed task to the `Warper` plugin
- Asynchronously executing code

## Creating a BukkitRunnable class

We will start by creating the `AlwaysDay` plugin. The code that we will write for this plugin will be put inside the `onEnable` method. The first step to creating a scheduled task is to create a `BukkitRunnable` class. This class will comprise very few lines of code. Therefore, it is not necessary to create a whole new Java file for it. For this reason, we will create a class within the `onEnable` method. This can be done using the following line of code:

```
BukkitRunnable runnable = new BukkitRunnable();
```

Normally, this code would be valid since you are constructing a new instance of a class. However, `BukkitRunnable` is an **abstract class**, which means that it cannot be instantiated. The purpose of an **abstract class** is to provide some base code that other classes can **extend** and build on top of. An example of this is the `JavaPlugin` class. For each plugin that you created, you started with a class that extends `JavaPlugin`. This allows you to override methods, such as `onEnable`, while keeping the current code of other methods, such as `getConfig`. This is similar to implementing an Interface, such as Listener. The difference between an abstract class and an interface is its purpose. As mentioned earlier, an abstract class is a base for other classes to extend. An interface is more of an outline within which classes can implement. Interfaces do not include code within any of their methods and therefore, all methods within an interface must have an implementation. For an abstract class, only the methods defined as `abstract` must be overridden because they do not include code within them. Therefore, because `BukkitRunnable` is an abstract class, you will be given a warning that will ask you to implement all the abstract methods. NetBeans can automatically add the needed methods for you. The new method that is added for you is `run`. This method will be called when the scheduler runs your task. For the new `AlwaysDay` plugin, we want the task to set each world's time to noon, as follows:

```
BukkitRunnable runnable = new BukkitRunnable() {
  @Override
  public void run() {
    for (World world : Bukkit.getWorlds()) {
      //Set the time to noon
      world.setTime(6000);
    }
  }
};
```

Remember that time on a Minecraft server is measured in ticks. 20 ticks is equivalent to 1 second. The measurement of ticks is as follows:

0 ticks: Dawn

6,000 ticks: Noon

12,000 ticks: Dusk

18,000 ticks: Midnight

Take a look at the API documentation for the `BukkitRunnable` class at `https://hub.spigotmc.org/javadocs/spigot/org/bukkit/scheduler/BukkitRunnable.html`. Note that there are six ways to run this task, which are as follows:

- runTask
- runTaskAsynchronously
- runTaskLater
- runTaskLaterAsynchronously
- runTaskTimer
- runTaskTimerAsynchronously

## Synchronous versus asynchronous tasks

A task can be run either synchronously or asynchronously. Simply put, when a synchronous task is executed, it must be completed before the server can continue running normally. An asynchronous task can run in the background while the server continues to function. If a task accesses the Bukkit API in any way, then it should be run synchronously. For this reason, you will rarely run a task asynchronously. An advantage that an asynchronous task gives is that it can be completed without causing your server to lag. For example, writing data to a save file can be done asynchronously. Later in this chapter, we will modify the `Warper` plugin to save its data asynchronously. As for the `AlwaysDay` plugin, we must run the task synchronously because it accesses the Minecraft server.

## Running a task from a BukkitRunnable class

Calling `runTask` or `runTaskAsynchronously` on a `BukkitRunnable` class will cause the task to run immediately. The only time you are likely to use this is to run a synchronous task from an asynchronous context or vice versa.

## Running a task later from a BukkitRunnable class

Calling `runTaskLater` or `runTaskLaterAsynchronously` on a `BukkitRunnable` class will delay the task from being executed for a specific amount of time. The amount of time is measured in ticks. Remember that there are 20 ticks in every second. In the `Warper` plugin, we will add a warp delay so that the player is teleported 5 seconds after running the warp command. We will accomplish this by running the task later.

## Running a task timer from a BukkitRunnable class

Calling `runTaskTimer` or `runTaskTimerAsynchronously` on a `BukkitRunnable` class will repeat the task every specified number of ticks. The task will repeat until it is canceled. Task timers can also be delayed to offset the initial run of the task. A task timer can be used to periodically save data, but for now, we will use this type of repeating task to complete the `AlwaysDay` plugin.

# Writing a repeating task for a plugin

We already have a `BukkitRunnable` class. Therefore, in order to run a task timer, we just need to determine the delay and the period by which the task is delayed. We want the delay to be 0. That way, if it is night when the plugin is enabled, the time will be set to noon right away. As for the period, we can repeat the task every second if we want to keep the sun always directly above. The task only contains one simple line of code. Repeating it often will not cause the server to lag. However, repeating the task every minute will still prevent the world from ever growing dark and will be less of a strain on the computer. Therefore, we will delay the task by 0 ticks and repeat it every 1,200 ticks. This results in the following line of code:

```
runnable.runTaskTimer(this, 0, 1200);
```

With this, we started a repeating task. It is good practice to cancel repeating tasks when the plugin is disabled. To accomplish this, we will store the `BukkitTask` as a class variable so that we can access it later to disable it. Once you have canceled the task within the `onDisable` method, the entire `AlwaysDay` plugin is given in the following code:

```
package com.codisimus.alwaysday;

import org.bukkit.Bukkit;
import org.bukkit.World;
import org.bukkit.plugin.java.JavaPlugin;
import org.bukkit.scheduler.BukkitRunnable;
import org.bukkit.scheduler.BukkitTask;

public class AlwaysDay extends JavaPlugin {
    BukkitTask dayLightTask;

    @Override
    public void onDisable() {

        dayLightTask.cancel();

    }

  @Override
  public void onEnable() {
    BukkitRunnable bRunnable = new BukkitRunnable() {
      @Override
      public void run() {
        for (World world : Bukkit.getWorlds()) {
          //Set the time to noon
          world.setTime(6000);
        }
      }
    };

    //Repeat task every 1200 ticks (1 minute)
    dayLightTask = bRunnable.runTaskTimer(this, 0, 1200);
  }
}
```

## Adding a delayed task to a plugin

We will now add a warp delay to the `Warper` plugin. This will require players to stand still after running the warp or home commands. If they move too much, the warp task will be canceled and they will not be teleported. This will prevent players from teleporting when someone is attacking them or they are falling to their death.

If you haven't already, add a variable of `warpDelay` in the `main` class. This is given in the following line of code:

```
static int warpDelay = 5; //in seconds
```

This time will be in seconds. We will multiply it by 20 later in the code to calculate the number of ticks by which we wish to delay the task.

We will also need to keep track of the player who is in the process of warping so that we can check whether they are moving. Add another variable of the current `warpers`. This will be a `HashMap` so that we can keep track of which players are warping and the tasks that will be run to teleport them. That way, if a specific player moves, we can get their task and cancel it. This is shown in the following code:

```
private static HashMap<String, BukkitTask>
   warpers = new HashMap<>();
   //Player UUID -> Warp Task
```

The code contains three new methods, which must be added to the `main` class in order to schedule the warp task, check whether a player has a warp task, and cancel a player's warp task. The code is as follows:

```
/**
 * Schedules a Player to be teleported after the delay time
 *
 * @param player The Player being teleported
 * @param loc The location of the destination
 */
public static void scheduleWarp
  (final Player player, final Location loc) {
  //Inform the player that they will be teleported
  player.sendMessage("You will be teleported in "
     + warpDelay + " seconds");

  //Create a task to teleport the player
  BukkitRunnable bRunnable = new BukkitRunnable() {
    @Override
    public void run() {
      player.teleport(loc);

      //Remove the player as a warper because they have already been
      teleported
      warpers.remove(player.getName());
    }
  };

  //Schedule the task to run later
  BukkitTask task = bRunnable.runTaskLater
    (plugin, 20L * warpDelay);
    //Keep track of the player and their warp task
    warpers.put(player.getUniqueId().toString(), task);
  }

  /**
   * Returns true if the player is waiting to be teleported
   *
   * @param player The UUID of the Player in question
   * @return true if the player is waiting to be warped
   */
  public static boolean isWarping(String player) {
    return warpers.containsKey(player);
  }

  /**
   * Cancels the warp task for the given player
   *
   * @param player The UUID of the Player whose warp task will be
  canceled
   */
  public static void cancelWarp(String player) {
    //Check if the player is warping
    if (isWarping(player)) {
      //Remove the player as a warper
      //Cancel the task so that the player is not teleported
      warpers.remove(player).cancel();
    }
  }
```

In the `scheduleTeleportation` method, note that both the `player` and `loc` variables are `final`. This is required to use the variables within the `BukkitRunnable` class. This must be done to ensure that the values will not change before the task is run. Also, note that the `runTaskLater` method call returns a `BukkitTask`, which is what we save inside the `HashMap`. You can see why it is saved by looking at the `cancelWarp` method, which removes the `BukkitTask` of the given player and then invokes the `cancel` method on it before it is executed.

In both the `WarpCommand` and `HomeCommand` classes, we teleport a player. We want to remove that line and replace it with a method call to `scheduleTeleportation`. The feature addition is nearing completion. All that is left to do is calling the `cancelWarp` method when a `warper` moves. For this, add an event listener to listen for the `player move` event. This can be seen in the following code:

```
package com.codisimus.warper;

import org.bukkit.block.Block;
import org.bukkit.entity.Player;
import org.bukkit.event.EventHandler;
import org.bukkit.event.EventPriority;
import org.bukkit.event.Listener;
import org.bukkit.event.player.PlayerMoveEvent;

public class WarperPlayerListener implements Listener {
  @EventHandler (priority = EventPriority.MONITOR)
  public void onPlayerMove(PlayerMoveEvent event) {
    Player player = event.getPlayer();
    String playerUuid = player.getUniqueId().toString();

    //We only care about this event if the player is flagged as
    warping
    if (Warper.isWarping(playerUuid)) {
      //Compare the block locations rather than the player locations
      //This allows a player to move their head without canceling the
      warp
      Block blockFrom = event.getFrom().getBlock();
      Block blockTo = event.getTo().getBlock();

      //Cancel the warp if the player moves to a different block
      if (!blockFrom.equals(blockTo)) {
        Warper.cancelWarp(playerUuid);
        player.sendMessage("Warping canceled because you moved!");
      }
    }
  }
}
```

Do not forget to register the event within the `onEnable` method.

## Executing code asynchronously

We can improve the `Warper` plugin by writing its data to a file asynchronously. This will help keep the main thread of the server running smoothly with no lag.

Take a look at the current `save` method. We will add the data to a `YamlConfiguration` file and then write the configuration to the file. This entire method cannot be run asynchronously. Adding the data to the configuration must be done synchronously to ensure that it is not modified while it is being added. However, the `save` method call on the configuration can be called asynchronously. We will place the entire `try/catch` block within a new `BukkitRunnable class`. We will then run it asynchronously as a task. This task will be stored as a static variable in the `Warper` class. This is shown in the following code:

```
BukkitRunnable saveRunnable = new BukkitRunnable() {
  @Override
  public void run() {
    try {
      //Write the configuration to our save file
      config.save(new File(plugin.getDataFolder(), "warps.yml"));
    } catch (Exception saveFailed) {
      plugin.getLogger().log
        (Level.SEVERE, "Save Failed!", saveFailed);
    }
  }
};

saveTask = saveRunnable.runTaskAsynchronously(plugin);
```

Now, the rest of the server can continue running while the data is being saved.

However, what if we try to save the file again when the previous write is not yet finished? In this case, we do not care about the previous task, because it is now saving outdated data. We will first cancel the task before starting a new one. This will be done using the following code before creating the `BukkitRunnable` class:

```
if (saveTask != null) {
  saveTask.cancel();
}
```

This completes this version of `Warper`. As mentioned in *Chapter 9*, *Saving Your Data*, this plugin has a lot of potential for feature additions. You now have the required knowledge to add these additions on your own.

## Summary

You are now familiar with most of the complicated aspects of the Bukkit API. With this knowledge, you can program almost any type of Bukkit plugin. Try putting this knowledge to use by creating a new plugin. You can perhaps try to write an announcement plugin that will rotate through a list of messages that need to be broadcast on a server. Think about all the Bukkit API concepts and how you can use them to add new features to the plugin. For example, with an announcement plugin, you can do the following:

- Add commands to allow an administrator to add messages that need to be announced
- Add permissions to control who can add messages and even who can see the messages that are announced
- Add an `EventHandler` method to listen for when players log in so that a message can be sent to them
- Add a `config.yml` file to set how often messages should be announced
- Add a save file to save and load all the messages that will be announced
- Use the Bukkit scheduler to repeatedly broadcast the messages while the server is running

For any plugin that you make, think of each segment of the Bukkit API to figure out some way to improve the plugin by adding more features. This will surely make your plugin and server stand out.

There are some topics that were not discussed in this book, but they are simple enough; you can learn how to use them on your own by reading the API documentation. Some interesting features that can spruce up a Bukkit plugin are the `playSound` and `playEffect` methods, which can be found inside the `World` and `Player` classes. I encourage you to read about them and try to use them yourself.

You know how to program plugin commands, player permissions, event listeners, configuration files, the saving and loading of data, and scheduled tasks. All that remains is imagining how to use these newfound skills to create a great and unique plugin for the Bukkit server.

# Index

[PACKT] PUBLISHING

Thank you for buying

**Building Minecraft Server Modifications**

*Second Edition*

## About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

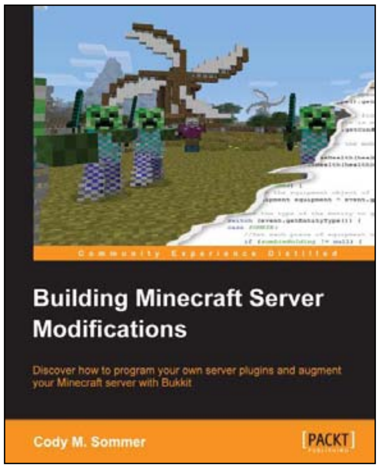
Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

## Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT]
PUBLISHING



## Building Minecraft Server Modifications

ISBN: 978-1-84969-600-5 Paperback: 142 pages

Discover how to program your own server plugins and augment your Minecraft server with Bukkit

1. Create your own Minecraft server mods.
2. Set up a Bukkit server that all your Minecraft friends can play on.
3. Step by step instructions guide you through the creation of several unique mods.

## Instant Minecraft Designs How-to

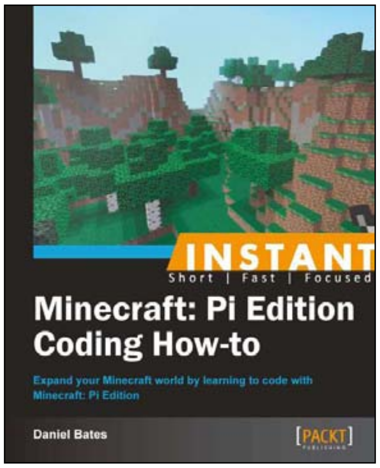
ISBN: 978-1-84969-598-5 Paperback: 76 pages

Build amazing structures using the very popular and most advanced of free mods for Minecraft – WorldEdit CUI and VoxelSniper GUI

1. Learn something new in an Instant! A short, fast, and focused guide delivering immediate results.
2. Build structures quickly and efficiently using WorldEdit CUI.
3. Learn the most useful functions of VoxelSniper GUI to build complex and aesthetically pleasing architecture.

Please check **www.PacktPub.com** for information on our titles

[PACKT]
PUBLISHING

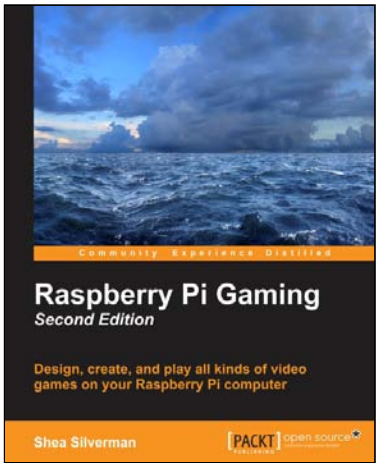


**Instant Minecraft: Pi Edition Coding How-to**

ISBN: 978-1-78328-063-6 Paperback: 50 pages

Expand your Minecraft world by learning to code with Minecraft: Pi Edition

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Enhance your Minecraft building techniques using computer code.
3. Get started with the Linux operating system on the Raspberry Pi.


Community Experience Distilled
Raspberry Pi Gaming
Second Edition
Design, create, and play all kinds of video games on your Raspberry Pi computer
Shea Silverman
[PACKT] open source

**Raspberry Pi Gaming**
***Second Edition***

ISBN: 978-1-78439-933-7 Paperback: 140 pages

Design, create, and play all kinds of video games on your Raspberry Pi computer

1. Program your very own video game on the Raspberry Pi using the Scratch programming language.
2. Install and manage your Raspberry Pi.
3. Set up your Raspberry Pi to play hundreds of retro and classic games.

Please check **www.PacktPub.com** for information on our titles