ROBOTICS PROGRAMMING: ALL YOU NEED TO KNOW

THE ROBOT WON'T CODE ITSELF!

Hello! If you are reading this, it is too late. Turn back while you still can. I am serious. Really? You're staying? Well, don't say I didn't warn you. You are now stuck programming the robot for Robotics. But don't worry! This guide will let you know some of the basics of how to program the robot and help you get started on figuring this stuff out. This guide is written on the assumption that you have some basic knowledge of Java and how it works (if you are taking AP Computer Science this year, you will be fine). If not, this may be a little bit harder for you. Either way, I wish you the best of luck!

TABLE OF CONTENTS

CHAPTER 1: UPDATING THE SOFTWARE

The first part in any coding project is making sure that the software you are using is up to date. There are several pieces of software that we use to build and test the robot. Some main ones include Visual Studio Code (VS Code), the FRC Driver Station, the FRC Radio Configuration, the roboRIO Imaging Tool, Phoenix Tuner, REV Hardware Client, Limelight Finder, and Shuffleboard. Depending on what the challenge is for the given year, you may end up using all of these, some of these, or even some I have not included. I will go over what each of these tools are used for in the table below.

Application	Usage
VS Code	This is the main application you will use to code the robot. This IDE will contain various folder and text colors that I will go over in the next section.
FRC Driver Station	This is the application that you will use to turn the robot on and control its actions. We use gaming controllers to control the robot's movements and bind the robot's actions to the controller buttons. You will use this platform to see the controller as well as enable and disable the autonomous and remote operation (TeleOperated) for the robot.
FRC Radio Configuration	The radio is the medium sized white box thingy that allows the robot to connect to the laptop. The radio needs to be a similar version to the roboRIO. This app allows the radio to be configured (updated and downgraded) through a complicated process that I am unfamiliar with. This also is used during competitions so that the robot can connect wirelessly to the competition officials panel thingy.

roboRIO Imaging Tool	This application is used to configure, connect to, and update the roboRIO. This is the gray flare box with a lot of ports to which the radio and several other components of the robot connect to. It is like the brain of the robot that controls pretty much all of its actions.
Phoenix Tuner	This app is used to see and communicate with the Talon motor controllers as well as the Power Distribution Hub. This is where you can update the above to the correct versions as well as see the ports to use when you implement them in the code.
REV Hardware Client	In a similar way to Phoenix Tuner, this app is used to see and communicate with the SPARK motor controllers as well as the Pneumatics Control Hub (PCH). This is where you can update the above to the correct versions as well as set the CAN ID to use when you implement them in the code. *Make sure the CAN ID matches the port that the item is plugged into!!! (This will save you a lot of trouble later. You're welcome)*
Limelight Finder	This app is used to communicate with the limelight, which acts as a sort camera for the robot. It can also be used to gauge distances from certain reflective surfaces used in the field. This app allows for the connection and preparation of the limelight for later use.
Shuffleboard	This is the later use of the limelight (I know, that was fast). Shuffleboard is the app that allows for the data from the limelight to be interpreted and sent

	to the laptop. This information can then be sent to the code for use however is needed.

Reference the video _____ for more information about each of the above applications.

Arguably the most tedious and annoying part of the entire process is updating all of the software. The first thing that can and should be done ASAP is to go on the FIRST website, as well as other sources as necessary, to look up the updates and new software that will need to be installed for this year's season. While it may seem tempting to just ignore the updates and skip them, it is important to keep the programs up to date so that a few years down the line we are not completely stuck in the past (plus the robot may not work properly if not sooooo :P).

CHAPTER 2: HOW VISUAL STUDIO CODE WORKS

As I said earlier, VS Code is the IDE on which you will be coding the robot. Now at first, it may seem very confusing because there are several steps that need to be taken so that it will work properly. I will detail them out as best as I can.

1. Open the latest version of VS Code that you (hopefully) just installed.
2. In either case, if you click on the fancy little 'W' icon on the top right of the screen, you should be prompted with a text box that appears in the middle towards the top of your screen.
3. Select the option that says 'WPILib: Create a new project'.
4. This will then take you to a screen full of white text boxes where you need to put in a couple pieces of information. In the first box, select 'template'. Then select 'java' and 'Command Robot'. Afterwards, choose a file from the laptop to save the code in and give the code a name. Enter the team number (5016) and press the 'Generate Project' button on the bottom. You can then open the project in a new window if it prompts you to do so.

This is the VS Code IDE. The first thing to note is that there are like a million files on the left side that look very intimidating. We are only going to be

messing with a couple of those files. Click on the tab that says '>src' and open '>java'. This is the only place that we will be coding (as you open certain tabs, note how they pop on the top just like tabs of google and whatnot so that you can reopen them easily later). In this tab, there are a couple of important parts. The 'subsystems' tab, the 'commands' tab, the 'Constants.java' tab, and the 'RobotContainer.java' tab. I will break them down one at a time. However, the most important tool you have is last year's code so that you can reference what they did and copy it for this year's code.

Subsystems

The 'subsystems' section is where we will be putting the majority of the classes, one for each part of the robot (you can be as specific as you want or as general as you want when creating these subsystems but the fewer the make, the more crowded each one will be so I recommend spreading them out into several different smaller section of a manageable size each). Each time you begin a new project, you should always have a tab called 'ExampleSubsystems.java'. This tab is full of sample code complete with a sample constructor and a sample method that both do nothing. The thing that makes it useful is that the imports are correct for a possible subsystem and some of the less intuitive code that is needed to make the code work is already provided. This makes it ideal for copying each time you want to add a new subsystem to the robot. That being said, much of the code will need to be edited as it is not relevant to the code that you will be adding. To copy the tab, right click the file and click 'copy' and then press CTRL and V to paste the code in. When you paste it in, rename the file to whatever subsystem you want. Make sure that the class header and the constructor all match the new file name you created. Pretty much all of the sample code after the constructor can be deleted and replaced with whatever code you need.

Commands

The 'commands' section is where you will create tasks for the robot to execute. This involves importing and using the classes/methods created in the subsystems section to create a series of actions for the robot to follow. In a similar way to the subsystems tab, an ExampleCommand.java tab will be provided when a new project is created. You will want to once again copy this file

and paste a new version in, renaming it as you see fit. This section also comes with an Autos.java tab which I promptly deleted from my files but you can use as a sort of template in the same way as the ExampleCommand.java should you see fit. Just like a subsystems file, a file in the commands tab also needs imports, a class header, and a constructor (make sure to match the file name to the class header and the constructor just as you do in the subsystems section). However, here the prewritten code is a little more important so we will go through each part one by one. Just like in any other code, the instance variables (mostly just variables specific to the imports [subsystems and FRC custom libraries] are needed) will go under the class header and will be specified in the constructor. This is where things get a little different. The ‘initialize’ method that is already provided is where you will put the code for the robot that runs ONLY ONCE as soon as the command is called. The ‘execute’ method that is already provided is where you will put the code for the robot that runs PRETTY MUCH FOREVER (think of it as a while loop with nothing telling it to stop) until the command ends. The ‘end’ method that is already provided is where you will put the code for the robot that runs ONLY ONCE when the command is terminated. The ‘isFinished’ command then turns to true (which you do not need to add anything in this method) and the command ends. Think of it like this, I want to make a command for ordering a coffee. My instance variable would be the size of the drink (that I choose as the parameter in the constructor when I call the method in another file). Then, in the ‘initialize’ method, I would tell the robot to get the cup and the coffee. In the ‘execute’ method, I would tell it to pour continuously until the command ends. Finally, in the ‘end’ method, it will give me the coffee (whereupon the ‘isFinished' command returns true and I leave).

Constants.java

The ‘Constants.java’ section is where you can and should store your constants for the entire robot. Such constants include the port numbers for the motors and the integer that corresponds to certain joystick buttons. Such constants will be found in other applications (such as the CAN ID on PhoenixTuner and the number [in order starting from 1] of each light that corresponds to a certain joystick button in the FRC Driver Station). In other tabs you can access the constants by making sure the ‘Constants.java’ file is imported and typing Constants.*variableName*.

RobotContainer.java

The ‘RobotContainer.java’ section is like the Main file of most other IDEs. This is pretty much the only file that the laptop will actually read (unless you refer to other classes in this file which you most certainly will). All of the code here is pretty much a template for what you will need to code in this file, just simply replacing the examples with actual code you will use. To start, this is where you will need to import all of the files: the commands, the subsystems, and the constants. Below the class header, you will need to create instance variables for all of the classes you will use in the entirety of the robot as well as setting up the joysticks (generally we use one for the driver and one for the operator that controls the moving parts of the robot). Below that in the constructor, you will need to set a (or multiple) default command(s) that will run during the entirety of the teleop period. Under the ‘configureButtonBindings’ method, you will need to configure the joystick buttons to run the proper tasks (such as drive when held or shift gears when pressed). Finally, the ‘getAutonomousCommand’ method is where you will, guess what?, write your autonomous command for the robot. This will be called during the initial autonomous period of competition (generally the first 15 seconds of the match) or when you enable it in the FRC Driver Station during testing.

Overall, if you are unsure as to what certain lines of code do or what certain files are used for, you should refer to the green comments on each file as they often provide very useful information or go online at Chief Delphi or another forum website to see if anyone else is also unsure as well as a possible explanation.

CHAPTER 3: UNDERSTANDING THE CODE

If you are coding the robot, the single most important thing to know is how the code works. As having a foundation in Java is all but necessary for coding the robot, I will not be covering the basics of Java, only going over the specific parts related to VS Code that will make your life easier. Such details include their color coding system, some errors with possible fixes, and some general tips and tricks I learned through my experience programming the robot. But most importantly: *** Be sure to save your code as you progress as it does not save automatically! ***

Color Coding System

VS Code uses their own color coding system, making different pieces of text different colors in order to help you better understand what different pieces of code do in the context of the whole section. It is worth noting that the theme colors can be changed in the settings but for the default color scheme, here is what each color represents:

Yellow: The color yellow is used to denote class names and method names.
Green: The color green is used to denote subsystem names as well as import links and data types.
Light Blue: The color light blue is generally used to denote variables.
Dark Blue: The color dark blue is used to denote several important class creation keywords such as ‘public’, ‘class’, ‘private’, ‘import’, and ‘package’.
Purple: The color purple is used to denote several Java keywords like ‘new’, ‘return’, ‘while’, and ‘if’.
White: The color white is used to denote characters that are not necessarily words but are still important to the code like a comma, a semi colon, a period, and mathematical symbols.

Deploying the Code

Deploying the code is the way you send the code from the laptop to the robot. There are several steps you will need to take to make sure the code actually gets sent to the robot successfully. This process includes several different applications including VS Code, the laptop’s wifi, and the FRC Driver’s Station.

1. Open the laptop’s wifi and connect to the roboRIO of the robot. This can be either hardwired through an Ethernet cable or wirelessly through the laptop’s wifi. If the radio is already configured and updated, the radio wifi might be named (such as the wifi name being ‘5016’). If not, the radio might have a weird wifi name (like ‘qca_5g_test’).
2. Open the FRC Driver Station and take a look at the three colored lights in the middle of the screen (you should by default be on this tab but if not,

click on the top leftmost icon in the Drivers Station). If step 1 was completed correctly, the top light of the three should be green. If you have a joystick plugged into a USB on the laptop, the bottom light should be green. If the code has not been sent to the robot yet, the middle light will be red (the light will be green if you deployed the code to the robot recently in the past but you will need to do it again if you update the code at all).

3. Open the VS Code project you want to deploy and click on the fancy little 'W' icon on the top right of the screen, you should be prompted with a text box that appears in the middle towards the top of your screen.
4. Select the option that says 'WPILib: Deploy Robot Code' (this text box saves your most recent searches so if you recently deployed the code, it should be one of the top few options).
5. This should then prompt the terminal on the bottom to pop up (if it was not up already) whereupon you will see the progress of the deployment. If all goes well and no errors pop up, the terminal on the right side will also appear after a few seconds (if it was not up already) and it will run its own set of diagnostics.
6. Go back to the FRC Driver Station and see if after several seconds (generally around 10-15), the middle light turns green. If so, the code was successfully sent! If not, double check that there were no errors in the code and try again.
7. When all three lights are green, select the mode you want from the list on the left (generally it is either 'TeleOperated' or 'Autonomous') and click 'Enable' (if it wont let you click 'Enable' but all the lights are green, try closing out of the FRC Driver Station and reopening it again).

Errors

Coding the robot is not an easy task, so you are bound to make mistakes throughout your coding adventure. As such, you will get to become pretty familiar with some of the errors you encounter as the code progresses. There are three main types of errors that you will come across: the yellow squiggly lines under the text, the red squiggly lines under the text, and the error messages in the terminal at the bottom and right side of the screen. The yellow squiggles are similar to the blue squiggles on Google Docs. They are warnings that mean that the code will compile but just be aware of what they are trying to tell you. These

will make the tab appear golden with the number of warnings that are present. The red squiggly lines are similar to the red squiggly lines on Google Docs, there is a mistake that needs fixing or else the code will not run. These will make the tab appear red with the number of errors you have. Finally, the error messages in the terminals will appear only after you deploy the code and will let you know that something is not working properly. I will try to go over as many as I can remember and give you some ideas on how to fix them should they come up in your code (Note to readers: feel free to add to this list so that it becomes better and more comprehensive each year).

Errors	Explanation	Fixes
What the error actually says, as close as I can make it.	What the error means and or how it got there.	What you can do to resolve the error.
Yellow Squiggly Lines		
Yellow squiggly lines appear underneath some lines of the code.	These are warnings that happen occasionally. They might affect the performance of the code and will not cause any compile error and you can still deploy the code despite having them. Do not feel the need to remove all of them and oftentimes it is not necessary.	Most of the times I have seen these warnings are because the code is being called but not used. Sometimes, this means you can delete the underlined section of code. However, other times deleting it might just make the code not work so it is better off just leaving it. I would stick to the motto of if it ain't broke, dont fix it.
Red Squiggly Lines		
Red squiggly lines	These are error	If it is under one character, make

appear underneath some lines of the code.	lines that happen if you make a mistake in the code. This can be as small as forgetting to add a semicolon (in which case the squiggle would be pretty localized) or as big as a line of code being syntactically incorrect (in which case the squiggle would cover pretty much the whole line).	sure that a character is not missing (like a parenthesis, semicolon, or period). If it is under a whole word, double check that the color of the word matches what color it should be (for example if you create a variable of the Shifter class called shifter [with a lowercase s], double check that you are referring to the right thing). Other possible fixes include double checking that a variable is created and it is of the right type (private vs private static vs private static final), making sure the class you are referencing from is imported in the file, and double checking for spelling mistakes. If just the first section of the import name has the squiggles, click on the fancy little 'W' in the top right corner and select 'WPILib: Manage Vendor Libraries' and make sure that the correct libraries are installed.
Terminal Errors		
BUILD FAILED in 4s 4 actionable tasks: 4 executed	This error appears in the terminal on the bottom of your screen. What this means is that your radio is not connected to the roboRIO and the code is not being sent to the robot for one reason or another.	Check that the roboRIO is being powered and that the correct lights are on. Check that the radio is being powered and the correct lights are on. Check (on FRC Driver Station) that you are connected to the radio wifi.

Miscellaneous Things		
I clicked a button and like 5 million lines of code appeared in the tab.	This sometimes happens if something has a strikethrough in it and you click the wrong quick fix.	Close VS Code and reopen it.
I accidentally deleted all my code.	If you delete a file by mistake or give the laptop to the wrong person for a bit, sometimes the code can disappear.	Go to the desktop and open the recycling bin. Right click on the deleted file and click restore. It is always a good idea to have a backup of the code regardless just in case you for some reason can not get the code back.

If you come across an error and are still unsure what to do, try clicking on the lightbulb to the left of the line of code and see if it helps. If not, try checking on Chief Delphi or another forum website to see if anyone else had the same issue as you and some possible solutions.

Tips and Tricks

After coding the robot for a season, I have learned a couple tricks that would have been nice to know from the first day. So take this as a sort of insider's scoop of useful information for coding the robot.

1. **Save your code:** the code on VS Code does not automatically save so if you exit the application without saving, your code may be lost. This also applies to having a complete backup of the code in several places in case one file gets corrupted or for some reason something gets lost.
2. **Make sure the ports line up:** when dealing with motor controllers or the PCH, make sure that the ports that you are setting for the devices in the

code and in the applications (such as the CAN ID) matches up with the ports that the items are wired into and or are displayed on the app (for instance if a solenoid ID is plugged into port 7 on the PCH, make sure its ID is set to 7 both on the application and on the Constants.java file of the code).

3. **Double check the instance variable declarations:** when creating instance variables outside of robotics (such as in AP Computer Science), it may seem like you should more often than not use 'private *dataType variableName*'. However, for coding the robot, you are often going to have to use 'private static *dataType variableName*' and 'private static final *dataType variableName*' perhaps more often than you might expect. If these are incorrect, your code might not get errors but it will not work properly if they are wrong.
4. **Storage space:** the final product for the robot (just the code itself) may be about 750 MB. While this may not seem like a tremendous amount, the storage on the laptop gets filled very quickly with all of the applications that need to be downloaded. As such, it is important to make sure that the laptop has enough storage space for the code or else there may be some errors that cause some confusion. If the laptop needs more storage, in years past, we have used an external hard drive to add more space to the laptop.
5. **Following the trail:** when coding the robot, there may come a time when you start to forget what files cause what action, which may make debugging and or tweaking code fairly complicated. The way to make this as easy as possible is to follow the trail of what happens starting from the top. For instance, if I want the forward arrow to make the robot drive forward but it is not working, I will need to find the source of the problem by seeing what files and methods are being utilized to see which ones might be the culprit. So I would start by looking at the line of code with the button declaration and tracing the source back through a command, to another command, to a subsystem, to a method in the subsystem, to a constant in another file. While this may not directly give you an answer, it is a good way of seeing what parts are related to the issue and what components may need to be changed to resolve the issue.

Overall, if you are still confused after reading this, FRC has a lot of great information and sample code that you can use to get started on coding a new

device or using a new application. There are also a lot of videos online from other teams that have gone over some of the more confusing things when coding should you want to use it.

CHAPTER 4: SETTING UP RAMEN

Before coding the real robot, it is good to get some practice to make sure you are ready. Because of this in years past, the team has built a separate robot for the sole purpose of us programmers learning how to code them. As the neatness of the wiring is not really a top priority of this practice machine (looking like a plate of spaghetti), we call the robot 'NoodleBot' (or more affectionately 'Ramen'). Just like any robot, Ramen needs to be configured properly before we can even start to program her. I will go over the basic components of Ramen (because it is important that you are familiar with at least some of them) as well as the basics of how to get her moving.

The Components

Whatever robot you are programming (or really anything you are ever going to program), it is important to at least be somewhat familiar with the different components of the device and how they work. I will go over some of the most essential pieces of the robot and their use in the robot as a whole (in future years if this information becomes outdated, feel free to update it so that later readers have a more accurate description).

1. **The on/off switch:** the most important thing to know about the robot is how to turn it on and off. This is essential to making sure all of the members working on the robot (including us coders) are safe. The on/off switch has two parts, the side switch and the top switch. The side switch is used to turn the robot on. Simply push the side switch in (like a hinge) and when it clicks, the robot should turn on (as long as the battery is connected and charged). The top switch is a red button that turns the robot off.
2. **The battery:** the battery is the part of the robot that supplies power to the whole machine. It is slightly bigger and weighs more than you might first expect and stores around 12 volts on a good day. However the robots often consume a lot of energy so about every day or so you should make sure to switch out the battery for a fully charged one. To change the

battery, simply connect the connector of the battery to the corresponding connector on the robot.

3. **The radio:** the radio is what the laptop uses to connect wirelessly to the robot. It looks like a medium sized white box with sports for wires to plug into on one side. When the code is deployed (see Chapter 3, Deploying the Code for more information), it is through this device that the code actually reaches the robot. It also has several different indicator lights that mean different things depending on what is displayed.
4. **The roboRIO:** the roboRIO is like the brain of the robot. It is like the central center where information is passed into and out of and is where the code is ultimately sent to to make the robot do its thing. It looks like a mostly flat gray square with a bunch of ports sticking out of the top for other components of the robot to be plugged into. It too has several different indicator lights that mean different things depending on what is displayed.
5. **The motors:** the motors are what make the robot actually move. As programmers, we generally do not mess with the motors except in the code itself when it comes to configuring them.
6. **The motor controllers:** the motor controllers are what receive the code and pass the information on to the motors. Just like the motors themselves, we generally do not mess with them except in the code itself when it comes to configuring them. It is also important to know which type of motor controllers are being used as well as which application is used to manage them so that it streamlines the configuration process.
7. **The safety light:** the safety light is a vital part of the robot that lets people around the robot know whether it is on or off (the light is solid when the robot is on but disabled, blinking when the robot is on and enabled, and off when the robot is off). While there is nothing to be coded for it, it is still important that you are aware of what it does and what the lights mean.

Coding Ramen

Now that you know what each part of the robot is and what it does, we can start programming Ramen! When coding a robot, there are three main stages: determining and setting up the constants, creating and implementing the teleop code, and creating and implementing the autonomous code. As this is our first robot, we will focus only on getting the robot to move both with the joystick (in the teleoperated mode) and on its own (in the autonomous mode). I will go through

each one in as much detail as I can to guide you along the process. The green text will represent lines of code while the italics will represent variable names that can be changed to whatever you want them to be.

Part 1: The Setup

The first part of programming the robot is setting up the ‘Constants.java’ file and making sure all of the robot parts are linked up to the correct constants so that they can be accessed in the code. As this is a more simplistic robot, this file will only contain constants from two main components: the joystick and the motor controllers (I'm going to go through them relatively quickly but you can reference the table Chapter 1 and the video below the table for more information). Let's talk about the joystick first.

The joystick is a wireless controller that plugs in via USB to the laptop. The FRC Driver Station is what we will use to connect to the joystick and monitor what buttons are being pressed. This is done through the ‘USB Devices’ tab on the left side of the app. When this tab is open, the first thing to note is the constant you will use for the joystick itself is listed under the ’USB Order’ column on the left side. You will also see the potential inputs from the joystick which will change or turn green as they are activated. The joysticks (the actual slidey knob thingies on the controller, not to be confused with the controller itself which is sometimes called a joystick), are registered under the ‘Axes’ section where the constant value we will use is printed right on the bar and the input is given as a double value between -1 and 1. The ‘Buttons’ tab is used to detect which button is being pressed. They light up green when pressed and are numbered as follows: in the left column, the first green light is number 1, the one directly below it number 2, the one directly below that number 3 and so on, continuing into the next column. These numbers will be the constants we use for each button respectively. Finally, we have the ‘POV’ section. This is used for the arrows on the controller and works with angles rather than small whole numbers. Their classifications are as follows: the forward arrow is considered to be angle 0, the right arrow to be angle 90, the down arrow to be angle 180, and the left arrow to be angle 270 (it is worth noting that there are inputs between each of the arrows at 45 degrees but they are not going to be used in this guide).

Now let's talk about the motor controllers. As of the time this is being written, Ramen is equipped with four Talon SRX motor controllers. To determine these constants, we will need the help of Phoenix Tuner. When connected either

wirelessly or hardwired to the robot, click on the ‘Tuner Setup’ tab and press the green ‘Run Temporary Diagnostic Server’ button. When it is finished loading, you can click on the ‘CAN Devices’ tab to see the connected motor controllers (you may need to refresh to get them to pop up. This can be done by clicking on the ‘Options’ tab on the very top right of the app and setting ‘Auto Refresh Devices’ to ‘Disable’ and then clicking ‘Refresh Devices’). When they pop up, you will be able to see the constants you will need to use under the ‘ID’ column of the table. As we will be using tank drive for the drivetrain of Ramen (meaning all of the left motors spin together and all of the right motors spin together), you will want to figure out which motor controllers are on which side of the robot. This can be done by clicking on one of the devices and pressing the ‘Blink’ button. This will make the lights on the motor controller rapidly flash orange for a brief period whereupon you can write down what side they go to.

Once you know all of the constants you are going to use, you can start adding them to the ‘Constants.java’ tab. The way you are going to do that is first by deleting the ‘OperatorConstants’ class that is given to you by default. What you should be left with is the class header for ‘Constants’ and a single package on the top. Inside the class header for ‘Constants’ is where you will put all of the constants you will need. To set up a constant, you will need to type this code:

```
public static final int variableName = constant#;
```

The variable name should be related to the part that you are setting up a constant for (such as frontLeftTalon). You will need to copy the above line of code each time you want to add a new constant. You should also create a constant for speed (a double out of 1 with 1 being 100% of the motor’s power) to make sure that all of the motors on a given side are running at the same speed (if they are running at different speeds, something will break). You will probably have around 8 or so constants by the time you are done with this file (4 for the motor controllers, 1 for the controller port, and 2 for the joysticks that you will use to move the robot with [the arrow keys come in a different section], and one to to determine the speed). With that, it is time to move on to creating the movement files.

Part 2: Subsystem

For the teleop control to work, we are going to need to create a subsystem for the drivetrain and a command to get the drivetrain of the robot moving. We will first start with the subsystem. As you should always do when creating a new file, copy the example file 'ExampleSubsystem.java' and paste it in under the 'subsystems' folder. You can then rename the file to 'DriveTrain.java' or any other name you want so long as it ends in .java (I would recommend naming it something that makes sense because you will likely end up having over a dozen different files and they start to get confusing even when you do name them properly). When you do this, you will need to update the following lines of code to match your file name:

```
public class fileName extends SystemBase{
    public fileName() {}
```

You can then delete all of the code after these lines (except the necessary brackets) as well as the import with the yellow warning squiggle under it. Then you need to create your instance variables. These will be placed under the class header but above the constructor. The code will be slightly different depending on what motor controller is being used but the idea is the same either way. Add the following instance variables (for a Talon SRX motor controller):

```
private static final TalonSRXControlMode talonSRXControlMode = null;
private static TalonSRXControlMode percentOutput = null;
TalonSRX frontLeftMotor;
TalonSRX frontRightMotor;
TalonSRX backLeftMotor;
TalonSRX backRightMotor;
```

This code sets up variables for each motor controller (each of which are hooked up to a separate motor). However, there will be errors unless you import the proper files. The imports are as follows:

```
import com.ctre.phoenix.motorcontrol.TalonSRXControlMode;
import com.ctre.phoenix.motorcontrol.can.TalonSRX;
```

These imports allow for the use of the TalonSRX and TalonSRXControlMode classes which we are using to create objects of those classes. However, the

imports might not work unless you install the proper libraries on the project. If this happens, click on the fancy little ‘W’ in the top right corner and select ‘WPILib: Manage Vendor Libraries’ and click ‘Install new libraries (offline)’. Then, check all of the boxes next to the libraries to install and click ‘OK’. The next step is to set the instance variables. To do this, type the following code into the constructor:

```
frontLeftMotor = new TalonSRX(Constants.frontLeftTalon);
frontRightMotor = new TalonSRX(Constants.frontRightTalon);
backLeftMotor = new TalonSRX(Constants.backLeftTalon);
backRightMotor = new TalonSRX(Constants.backRightTalon);
```

This code sets up four new variables as instances of the Talson SRX class. This enables us to use methods on them that are custom to the class. The constants in parentheses are a reference to the ‘Costants.java’ tab we created earlier. As such, you will need to import the ‘Costants.java’ tab to this file. That is done with the import:

```
import frc.robot.Constants;
```

Alternatively, you can click on the red squiggly word and then click the lightbulb icon on the left where it will prompt you to, among other things, import the file for you. The next step is to create the method that will be used to drive the robot.

```
public void drive (double leftSpeed, double rightSpeed){
        frontLeftMotor.set(TalonSRXControlMode.percentOutput, leftSpeed);
        frontRightMotor.set(TalonSRXControlMode.percentOutput,
           -rightSpeed);
        backLeftMotor.set(TalonSRXControlMode.percentOutput,
           leftSpeed);
        backRightMotor.set(TalonSRXControlMode.percentOutput,
           -rightSpeed);
}
```

This method utilized a double parameter (between 0 and 1) to set the percent output speed of the motors. The negative signs in front of the *rightSpeed* are because the positive direction happens to spin the motor backwards (this changes depending on the direction the motor is set to and is different on each

motor so be sure to be aware of this and test for it). We now want to make specific methods that utilize this *drive* method to set different directions of robot movement.

```
public void straightForward (double speed){
        drive (speed, speed);
}

public void straightBackward(double speed){
        drive (-speed, -speed);
}

public void turnLeft(double speed){
        drive (speed, -speed);
}

public void turnRight(double speed){
        drive (-speed, speed);
}
```

Since *leftSpeed* is the first parameter in *drive*, the first *speed* in our new methods refers to the left tread of the robot. Since *rightSpeed* is the second parameter in *drive*, the second *speed* in our new methods refers to the right tread of the robot. Also, because we have the negative signs in our *drive* method, a positive speed parameter will always turn the robot wheels (treads) forward. This is why our *straightForward* method takes two positive *speed* values while our *straightBackward* method takes two negative *speed* values. The last thing we want to do for this subsystem is to add a stop method that stops the motors entirely. We can do this with the following method that sets both the *leftSpeed* and *rightSpeed* equal to 0:

```
public void stop(){
        drive (0,0);
}
```

Part 3: Command

The next part of making Ramen move is adding a command that lets the robot know how and when to use the methods. As you should always do when creating a new file, copy the example file 'ExampleCommand.java' and paste it in under the 'commands' folder. You can then rename the file to 'DefaultTeleopDrive.java' or any other name you want. When you do this, you will need to update the following lines of code to match your file name:

```
public class fileName extends SystemBase{
```

And later on:

```
public fileName() {}
```

As described in the Commands section of Chapter 2, a lot of this code is going to be very useful to you and we are only going to modify certain lines rather than delete it and start fresh. We will start with the imports. Since you are going to want to use the code that we created in the subsystems file, you are going to need to import that into our current file. While you are at it, you should also import the library for the controller (called a Joystick). You can do both with the following lines of code:

```
import frc.robot.subsystems.DriveTrain;
import edu.wpi.first.wpilibj.Joystick;
```

Once these are imported, you are going to want to make instance variables of each of these. This can be done by replacing the example instance variable with the following two:

```
private final DriveTrain runCoderun;
private Joystick driver;
```

After this, you will need to make your constructor with parameters that match the instance variables. You can replace the one they have with this:

```
public DefaultTeleopDrive (DriveTrain runCodeRun, Joystick driver){
```

```
this.runCodeRun = runCodeRun;
this.driver = driver;
addRequirements(runCodeRun);
}
```

Now that all of the instance variables are set, we can move on to the last piece of the puzzle for the command: the execute method (see the Commands section of Chapter 2 for more information). It is here that we are going to want to put our command to move because we want it to run whenever we are pressing it so long as the robot is enabled. We can put the following code in the execute method to make this happen:

```
double right = driver.getRawAxis(-Constants.driverRightJoystick);
double left = driver.getRawAxis(-Constants.driverRightJoystick);
runCodeRun.drive(left, right);
```

This code uses the ‘getRawAxis method from the Joystick class to get the double value of the controller’s left and right joystick (as you may have seen when testing, pressing forward on the joystick sends back a negative double when we really want a positive. As such, we need to negate the input which is seen in the negative sign in the above code). However, just as before, we are going to need to import our ‘Constants.java’ file to make this work which you can do either with the lightbulb or with the following line of code:

```
import frc.robot.Constants;
```

And with that, we just have one more step to make Ramen drive! This is configuring the ‘RobotContainer.java’ file. As described in the RobotContainer.java section of Chapter 2, this is like the Main file of most other IDEs. As such, this is where we need to tell the robot what we want it to do when we enable teleop mode. To do so, the first thing we are going to need to do is open the ‘RobotContainer.java’ file. The first thing we want to do is edit the red squiggly import. If you remember back in the beginning of the Ramen configuration we deleted ‘OperatorConstants’. As such, to fix the error, we simply need to remove this portion of the import so it matches the one we have been using like such:

```
import frc.robot.Constants.OperatorConstant;
```

After this, we are going to want to deal with the other error in this file. We are going to replace the two lines of code that set up a 'CommandXboxController' with code that creates an instance of the Joystick class and uses the correct constants that we set up. We also want to set up an instance of our *DriveTrain* subsystem. We can do both with the following two lines of code:

```
private final Joystick driver = new Joystick (Constants.joystickPort);
private final DriveTrain runCodeRun = new DriveTrain();
```

This will cause an error because we do not have the Joystick class nor the *DriveTrain* subsystem imported in this file so add this import with the others:

```
import edu.wpi.first.wpilibj.Joystick;
import frc.robot.subsystems.DriveTrain;
```

Then, to add the command for the robot to drive, we need to add the following line of code in the constructor:

```
runCodeRun.setDefaultCommand(new DefaultTeleopDrive(runCodeRun,
    driver));
```

What this does is tells the robot that, by default, we want the command you just made to be executed constantly. But of course, we have to import the command before we can make Ramen move. That import is:

```
import frc.robot.commands.DefaultTeleopDrive;
```

With that you can delete the one line of code that has an error and Ramen should be up and running! All you need to do is follow the steps outlined in the Deploying the Code section of Chapter 3.

Part 4: Autonomous

Now that Ramen moves when you tell her to, you have completed a basic teleoperated command. The other part of the competition (which actually comes first in the order of events) is the autonomous section. Essentially, it is time to get Ramen to move on her own. This is done through returning a command in the getAutonomousCommand method in 'RobotContainer.java'. I will walk you through making two different commands which we can then combine to make the robot go forwards, turn around, and come back. The first step in making this happen is creating the commands.

As you should always do when creating a new file, copy the example file 'ExampleCommand.java' and paste it in under the 'commands' folder. You can then rename the file to 'Forward.java' or any other name you want. When you do this, you will need to update the following lines of code to match your file name:

```
public class fileName extends SystemBase{
```

And later on:

```
public fileName() {}
```

Afterwards, under the class header, you are going to want to replace the sample instance variable with ones that are more pertinent to the class we are creating. As such, you should add the following lines of code:

```
private final DriveTrain runCodeRun;
Private double speed;
```

Don't forget to import the file for *DriveTrain* with the following code:

```
import frc.robot.subsystems.DriveTrain;
```

Then, in the setup of the constructor, you are going to want to add parameters for your instance variables, which you will set the value of inside the brackets with the following code:

```
public Forward (DriveTrain runCodeRun, double speed){
```

```
        this.runCodeRun = runCodeRun;
        this.speed = speed;
        addRequirements(runCodeRun);
    }
```

The last step in the setup of this file is to add code to tell the robot what to do when the class is executed (in the ‘execute’ method) and when the command ends (in the ‘end’ method). All you need to do is add this code:

```
    @Override
    public void execute() {
        runCodeRun.drive(speed, speed);
    }

    @Override
    public void end(boolean interrupted) {
        runCodeRun.drive(0, 0);
    }
```

What it does is that when this command is enabled, the *DriveTrain*’s *drive* method tells the motors to spin at a certain speed, causing the robot to move. When the command ends, the motor speed is reset to 0 and the robot stops. Now, you just need to delete the import with a warning and one of the two autonomous commands is done! See, it's not that bad. We just need to create the other file. However this time, I will teach you a shortcut. Before creating this file, I know that I want to make it the same as the *Forward* file except this time, I want to change the direction the treads spin in order to make the robot turn right. From before, I also know that this can be done simply by changing the sign of the *speed* variable. As such, instead of copying the ‘ExanpleCommand.java’ file, I am going to copy the *Forward*.java file to use as my template for my new file *DriveRight*.java. For the last time in this manual, you will need to update the following lines of code to match your file name:

```
    public class fileName extends SystemBase{
```

And later on:

```
public fileName() {}
```

This time, the only things you are going to need to change is the actual task to be completed in the ‘execute’ and ‘end’ methods just replace the code you had with the following and you are done:

```
@Override
public void execute() {
        runCodeRun.turnRight(speed);
}

@Override
public void end(boolean interrupted) {
        runCodeRun.turnRight(0);
}
```

In a similar way to the *Forward* file, these methods also tell the robot to move a certain way. However this time, the *turnRight* method of the *DriveTrain* class is being called, which then utilizes the *drive* method to spin the left tread forward and the right tread backwards, causing the robot to turn to the right.XXX Now that those commands are created, all you need to do is piece together the autonomous command to return to the ‘getAutonomousCommand’ method of the ‘RobotContainer.java’ file. Before you do that though, you are going to need to import the command files you just made with the following lines of code:

```
import frc.robot.commands.Forward;
import frc.robot.commands.TurnRight;
```

Now that this is squared away, just add this line of code to the ‘getAutonomousCommand’ method:

```
return (new Forward(runCodeRun,
   Constants.driveSpeed).withTimeout(5).andThen (new
   TurnRight(runCodeRun,
   Constants.driveSpeed).withTimeout(1.5).andThen (new
   Forward(runCodeRun, Constants.driveSpeed).withTimeout(5)));
```

I know this code may look a little crazy but I will break it down piece by piece. Everything after the ‘return’ is a command that is being returned to the computer to tell it what to do during the autonomous period. The first thing that happens is that a *Forward* command is sent (utilizing the proper parameters). This command uses the ‘.withTimeout’ method to tell the computer to run this command for 5 seconds (as that is the number in the parenthesis). The ‘.andThen’ method tells the computer that it is not done yet and still has more commands to send. This is then followed by the command *TurnRight* which runs for 1.5 seconds before calling the *Forward* method again for another 5 seconds (The timings in the ‘.withTimeout’ can and should be altered to fit what you want the robot to do. In my testing, the 1.5 seconds for *TurnRight* was about the right amount to make the robot turn around 180 degrees. The 5 seconds in the second calling of the *Forward* command is just to make the robot come back. The number 5 is arbitrary but it should match the number in the first *Forward* command if you want the robot to return to its original position). All that’s left to do is deploy the robot code and click ‘Enable’ on the ‘Autonomous’ section of the FRC Driver Station to see Ramen drive!

CONCLUSION:

For anyone who reads this, I hope this helped you gain a little more confidence in traversing the code and programming the robot. As a last piece of advice, try to get familiar with Ramen in the preseason so that you are ready to go with the real robot by the time build season comes around. Good luck in coding the robot this year and if you find something in this guide that is incorrect (whether due to a software update or just a plain mistake), feel free to edit this and keep this up to date for future programmers. Have fun and try not to get too stressed out while you're at it!