Building a Command Robot in Java

Keith Richardson, Mentor
Honeoye Robotics – Team 3951
keith@richardsonfam.com

Contents

Robot Overview 3

Installing Visual Studio Code 4

Robot Electronics Wiring and Sensor Map 4

Building your Robot 4

- Lab 1 - Building the Shell of your Program 4
- Lab 2 – The Arm Subsystem 7
- Lab 3 – Creating Arm Commands 14
- Lab 4 – Binding a Command to the Controller 17
- Lab 5 – Deploying the Robot 19
- Lab 6 – Adding the Arm Wheels to the Arm Subsystem 22
- Lab 7 - Stopping a Command with a Sensor 27
- Lab 8 – Building the Drive Subsystem 29
- Lab 9 – Making the robot drive with the controller 32
- Lab 10 – Making the robot turn with the triggers 34
- Lab 11 – Making the robot arms and wheels move with an axis 35

Robot Overview

Our 2018 robot contained the following features

- Arcade drive, with rotate being the triggers
- Arms that rotate up and down, and pull cubes in and out
- Limit switch that prevents the arms from going too far up
- Tower that extends up to hook onto the bar, then releases a latch and winches up
- 3 Autonomous modes that change based on game data

It is a simple robot, and we will build it in stages. We will focus on the teleop functions to get a feel of how the Java Command Based Robot is build.

A command robot splits up programming between subsystems and commands. You execute a command by pressing a button. The command takes over a subsystem to do a task.

Lets use an example:

Arms Subsystem contains a function to run the wheel motors in. The function could be called RunArmMotors and ask for a speed as a parameter. You can send a 1 to run them both full speed forward, and -1 to run them full speed in reverse. The subsystem function will take care of setting both motor controllers, as well as reversing the direction of the opposite motor controller so they both run in the same direction.

You will then build two commands, EatCube and SpitCube. EatCube will take over the Arms subsystem and run the arm motors in at a +1. When the command is terminated, it will stop motors.

SpitCube will be another command that will also take over the subsystem. If you are running Eat and then Spit while eat is running, Eat will be interrupted for Spit to run. Spit will be the reverse of Eat, just running the motors backwards.

We can then bind the commands to a button. For this example, we will use A to Eat and B to Spit.

The A button on the proper joystick will be bound using the WhileHeld function to the EatCube command.

The B button on the proper joystick will be bound using the WhileHeld function to the SpitCube command.

This allows a simple, readable command system for binding functions to buttons, and following the code. Say your arms are going too fast, and you never want them to go faster than +0.75 or -0.75. You can put the function in the Subsystem to override any values that the commands set, allowing you to change it in one place.

If you want to then include it in autonomous mode, you can make a command group, to chain commands together. You could add a timeout to the Spit and Chew command, that is set to a high value when you push the buttons but specify the value during autonomous mode. This allows us to use the same commands multiple times, while having the command code be limited to one spot. If we can’t or don’t want to reuse code, we can always make multiple commands to control the same subsystems. For example, SpitForTime and Spit could be different commands. You might want the teleop to be one power, but autonomous a different power, so making another command might make sense.

Installing Visual Studio Code

The current documentation on what you must do to get Visual Studio code up and running is located at

https://wpilib.screenstepslive.com/s/currentCS/m/79833/l/932382-installing-vs-code

It will have you download Visual Studio Code, the WPILib Extension, and other updates inside Visual Studio code to get up and running.

Robot Electronics Wiring and Sensor Map

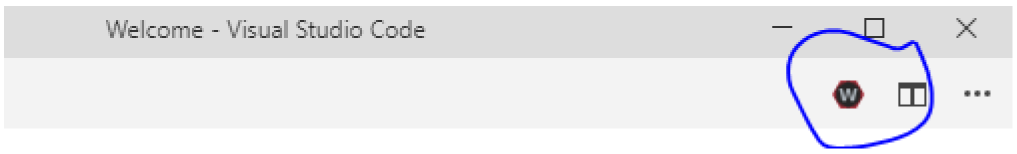
Component Name	Component Type	Bus/Port Type	Channel / ID
Left Drive Motor 1	Talon SRX w/ Encoder	CAN	1
Left Drive Motor 2	Victor SPX	CAN	2
Right Drive Motor 1	Talon SRX w/ Encoder	CAN	3
Right Drive Motor 2	Victor SPX	CAN	4
Arm Rotate Motor (Elbow)	Victor SPX	CAN	7
Arm Wheel Encoder	Encoder	Digital IO	A Channel: 0 B Channel: 1
Arm Wheel Left Motor	Victor SPX	CAN	9
Arm Wheel Right Motor	Victor SPX	CAN	10
Arm Top limit switch	Digital Input	Digital IO	2
Tower lift motor	Victor SPX	CAN	11
Tower lift limit switch	Digital Input	Digital IO	3
Tower latch release	Spark	PWM	0
Winch Climb Motor 1	Victor SPX	CAN	5
Winch Climb Motor 2	Victor SPX	CAN	6
Gyro	ADXRS450_Gyro	SPI	0
Camera	USB Camera	USB	0

Building your Robot

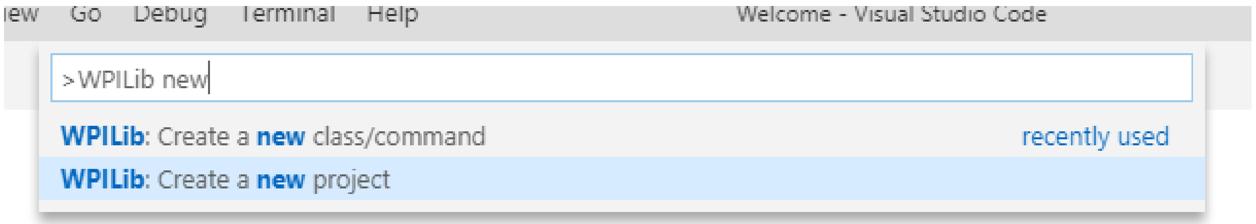
I am going to break up the program into different labs. The goal is each lab would teach you a new concept and allow you quick reference back when you are programming your robot.

Lab 1 - Building the Shell of your Program

Launch Visual Studio Code in a new window. Click on the WPI icon to bring up the shortcuts



Type in the word new to search for Create a new Project and select the option.



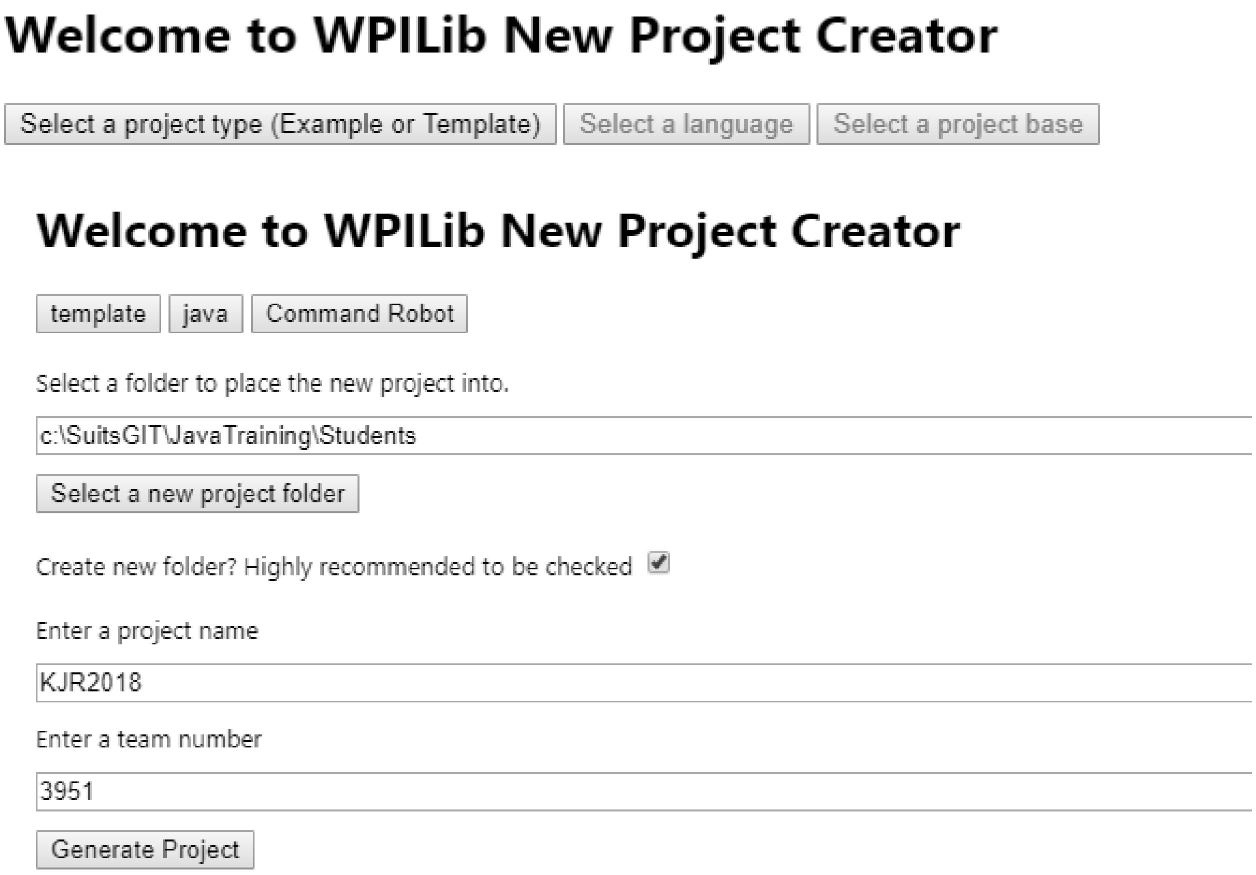
>WPILib new

WPILib: Create a new class/command recently used

WPILib: Create a new project

The Wizard pops up to create a new project.

First you will pick the project type, language, and base class. Ensure you pick template / java / Command Robot for this example.



Welcome to WPILib New Project Creator

Welcome to WPILib New Project Creator

Select a folder to place the new project into.

c:\SuitsGIT\JavaTraining\Students

Create new folder? Highly recommended to be checked ☑

Enter a project name

KJR2018

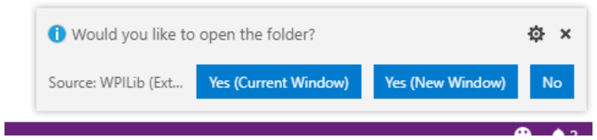
Enter a team number

3951

Next pick the base folder. This will create the new project under the Students folder under our Java Training GitHub project.

Then enter your initials and 2018 for your project name, our team number, and generate project.

Hit Generate Project and it will prompt you how to open the folder. Select Yes (Current Window)



Would you like to open the folder?

Source: WPILib (Ext...

You will notice your robot code will be created.

Expand the SRC folder and you will see the base files for your robot that are created.

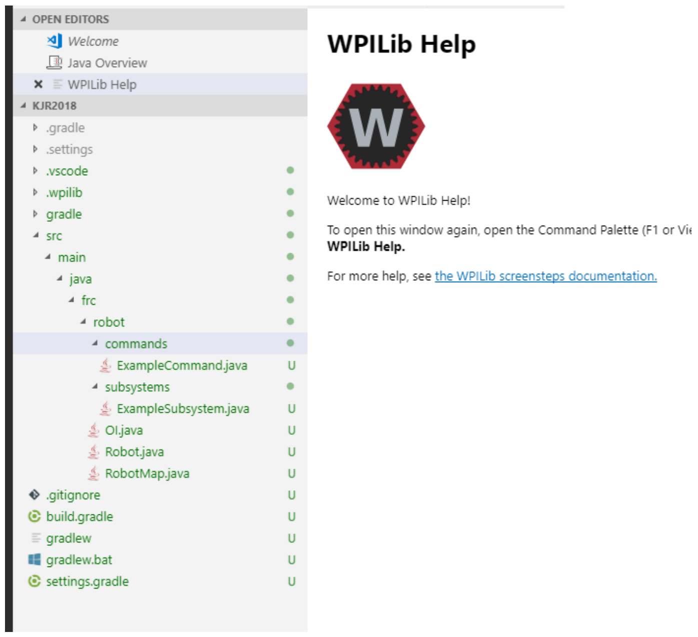
Robot.java is the main robot loop that runs your robot.

RobotMap.java sets the mapping for your motors and other constants that you want to use throughout the robot.

OI.java is the operator interface for the robot. This is where you setup your controllers, button definitions, and bind actions to buttons.

The subsystems folder will contain all of your subsystems. For example, drivetrain, arm, tower, etc.

Your commands folder contains commands that bind to your subsystem.



WPILib Help

Welcome to WPILib Help!

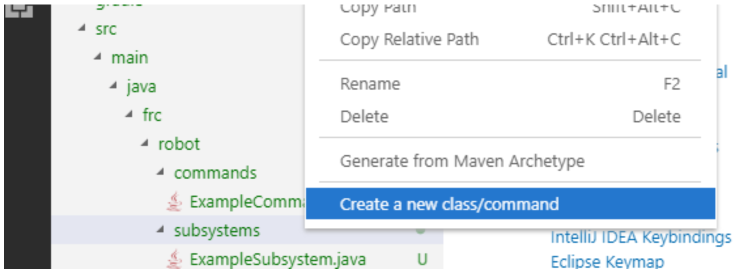
To open this window again, open the Command Palette (F1 or Vi
WPILib Help.

For more help, see the WPILib screensteps documentation.

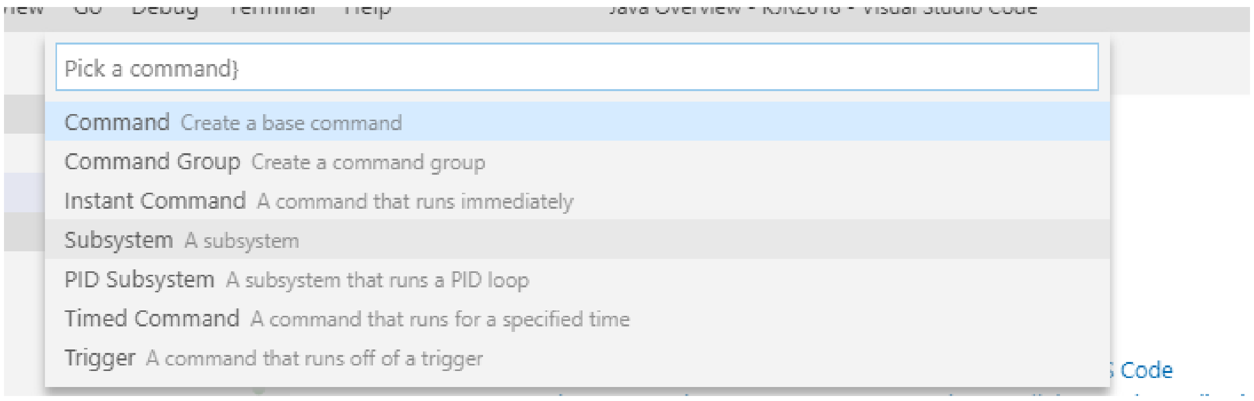
Lab 2 – The Arm Subsystem

The arm subsystem first? Yes, because it is easier than the drive. You might want to build a subsystem to move a few motors quick without needing a complicated drive system.

So first, we will create a new Subsystem. Right click on your Subsystems folder, and select create a new class/command.



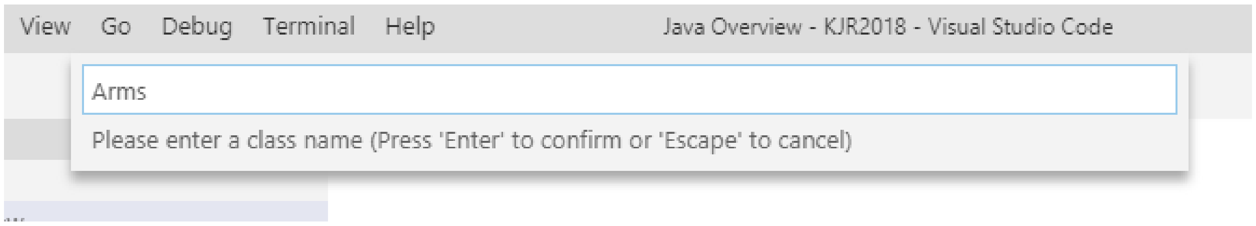
Pick the option for Subsystem to create a subsystem



Pick a command}

Command Create a base command
Command Group Create a command group
Instant Command A command that runs immediately
Subsystem A subsystem
PID Subsystem A subsystem that runs a PID loop
Timed Command A command that runs for a specified time
Trigger A command that runs off of a trigger

Enter Arms for our subsystem and hit enter to create.



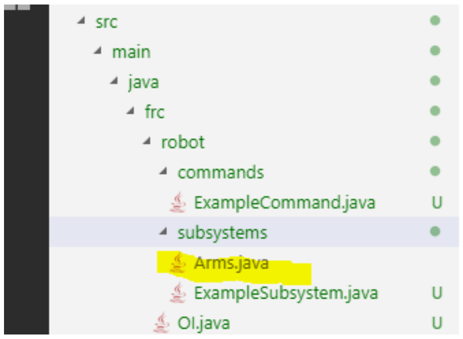
Arms

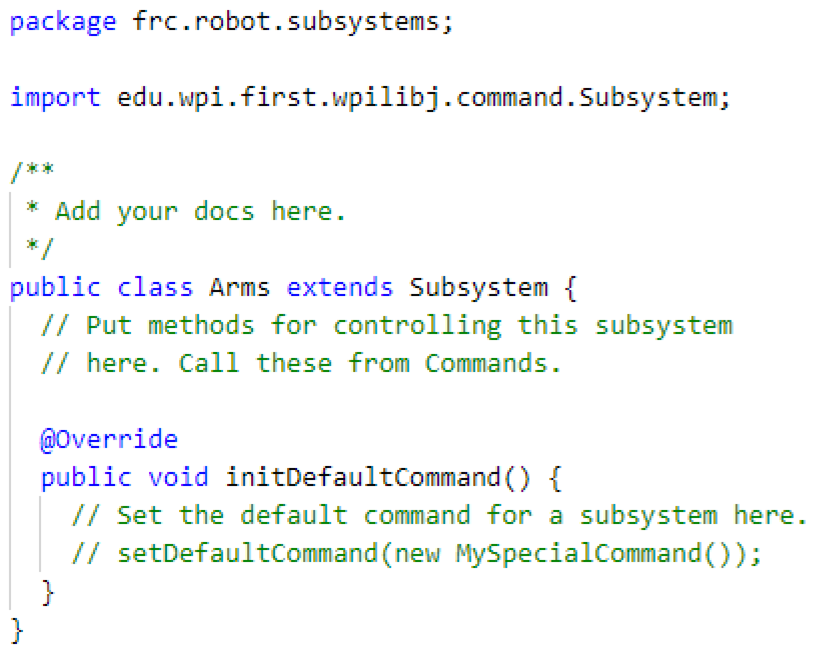
Please enter a class name (Press 'Enter' to confirm or 'Escape' to cancel)

You will now see the arms subsystem in the list:

Double click on the file to open it.

You will see your default subsystem when you load the file.



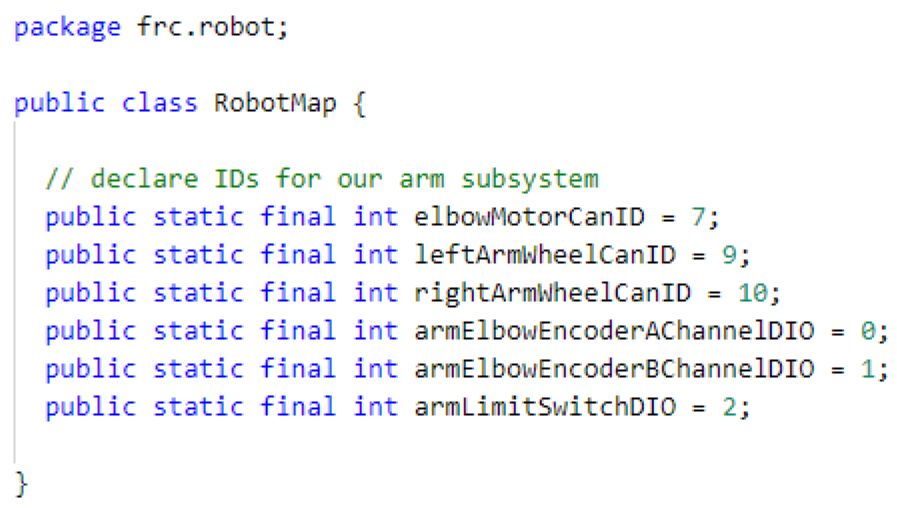


```java
package frc.robot.subsystems;

import edu.wpi.first.wpilibj.command.Subsystem;

/**
 * Add your docs here.
 */
public class Arms extends Subsystem {
  // Put methods for controlling this subsystem
  // here. Call these from Commands.

  @Override
  public void initDefaultCommand() {
    // Set the default command for a subsystem here.
    // setDefaultCommand(new MySpecialCommand());
  }
}
```

The default command runs whenever the subsystem is not doing anything else. By default, it does nothing.

First we will build our motors that our subsystem needs. According to the list of sensors and motors, we will use 2 Victor SPX motors for our arm wheel motors, and 1 for our arm elbow rotate motor. We will also have a limit switch that presents a digital input. Our motor controller for the elbow has the encoder wired into digital inputs as well. In order to build our motors, we need to define the Bus IDs in our RobotMap.java.

Open the RobotMap.java. It will be pretty empty, but with a lot of comments in there showing examples of what we need. It is best for us to use comments to help label what we do, as well as use descriptive variable names. Let’s create variables for the IDs of the controllers and inputs. Your file should look like this when it is done:



```java
package frc.robot;

public class RobotMap {

  // declare IDs for our arm subsystem
  public static final int elbowMotorCanID = 7;
  public static final int leftArmWheelCanID = 9;
  public static final int rightArmWheelCanID = 10;
  public static final int armElbowEncoderAChannelDIO = 0;
  public static final int armElbowEncoderBChannelDIO = 1;
  public static final int armLimitSwitchDIO = 2;

}
```

Notice that the example did not use the word final. Final is a way to say that the variable is read only, so there is no chance of it changing while your program is running.

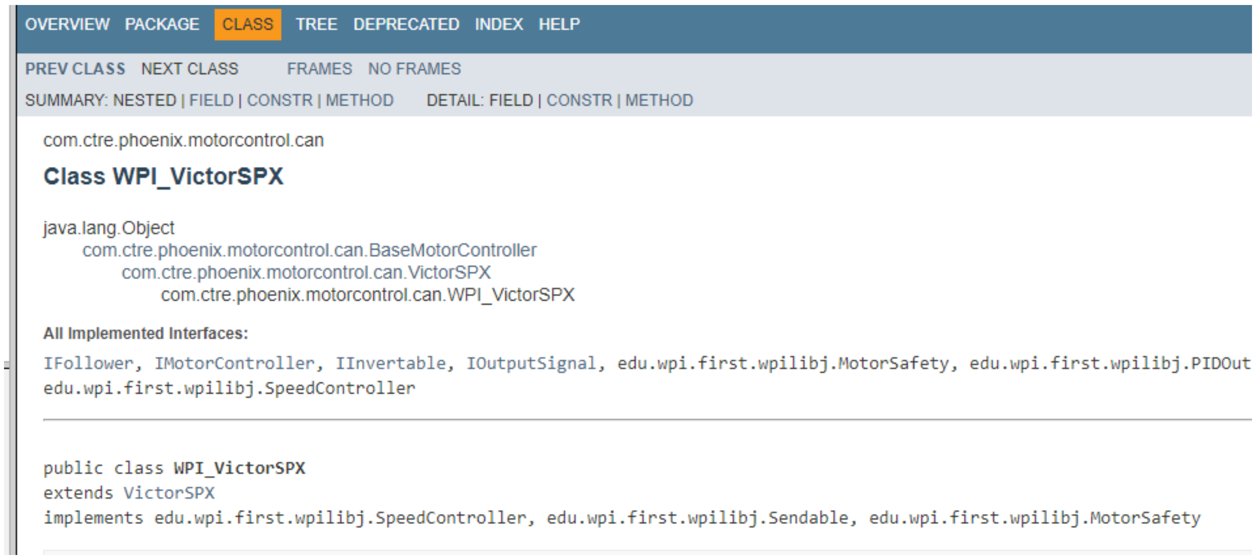
Now that we defined our mappings, we can head back to the Arms.java subsystem file.

Let’s define our first motor controller. We will first create a variable for the Elbow Motor It is using the VictorSPX class. If you want to integrate the VictorSPX into a speed controller group, we need to find the VictorSPX motor controller class on the FRC documentation.

First, look on the FRC WPILib java documentation site at
http://first.wpi.edu/FRC/roborio/release/docs/java/

I could not find the VictorSPX, so I went to the CTR Electronics Java help https://www.ctr-electronics.com/downloads/api/java/html/index.html?index-all.html

And found it. The WPI_ version works well with other WPILib groups and motor controllers, so I decided to use that class as a standard for all, even if I did not specifically need it.


OVERVIEW PACKAGE CLASS TREE DEPRECATED INDEX HELP
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

com.ctre.phoenix.motorcontrol.can

Class WPI_VictorSPX

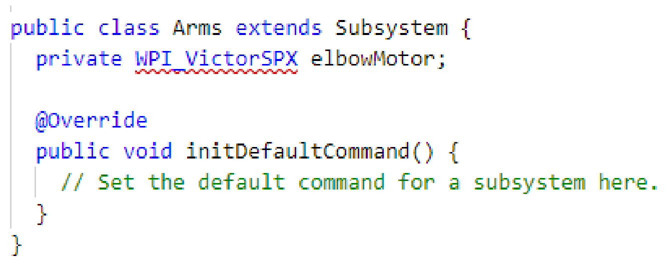
```
java.lang.Object
    com.ctre.phoenix.motorcontrol.can.BaseMotorController
        com.ctre.phoenix.motorcontrol.can.VictorSPX
            com.ctre.phoenix.motorcontrol.can.WPI_VictorSPX
```

All Implemented Interfaces:

```
IFollower, IMotorController, IInvertable, IOutputSignal, edu.wpi.first.wpilibj.MotorSafety, edu.wpi.first.wpilibj.PIDOut
edu.wpi.first.wpilibj.SpeedController
```

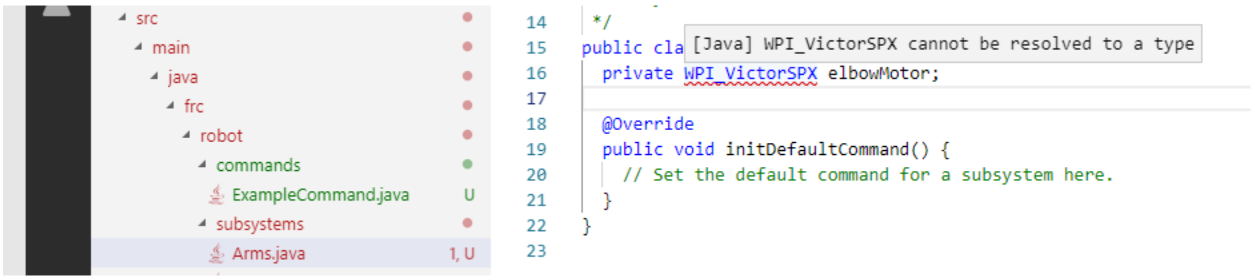
```
public class WPI_VictorSPX
extends VictorSPX
implements edu.wpi.first.wpilibj.SpeedController, edu.wpi.first.wpilibj.Sendable, edu.wpi.first.wpilibj.MotorSafety
```

Now your code will be:



```
public class Arms extends Subsystem {
  private WPI_VictorSPX elbowMotor;

  @Override
  public void initDefaultCommand() {
    // Set the default command for a subsystem here.
  }
}
```

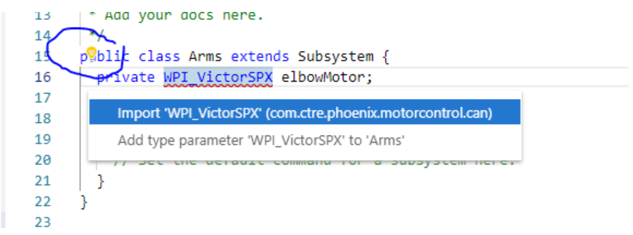
Notice the WPI_VictorSPX has a red underline, and the Arms.java is red. This is a code error. It is because it does not know how to find the class WPI_VictorSPX. Hover your mouse over to see the error message:



```
 */
public cla [Java] WPI_VictorSPX cannot be resolved to a type
  private WPI_VictorSPX elbowMotor;

  @Override
  public void initDefaultCommand() {
    // Set the default command for a subsystem here.
  }
}
```

So we need to fix the resolved type. Above on the documentation it shows it is under com.ctre.phoenix.motorcontrol.can namespace. We will either add an import command, or let the code fix it for us.

Click on the WPI_VictorSPX, then click on the yellow lightbulb that appears. We will choose import 'WPI_VictorSPX' (com.ctre.phoenix.otorcontrol.can)

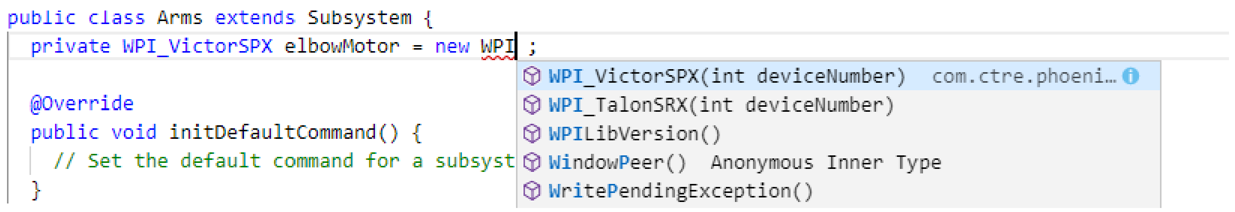

Import 'WPI_VictorSPX' (com.ctre.phoenix.motorcontrol.can)
Add type parameter 'WPI_VictorSPX' to 'Arms'

```
 * Add your docs here.
 */
public class Arms extends Subsystem {
  private WPI_VictorSPX elbowMotor;
```

It added the proper import line. This is a shortcut we will use a LOT.

Now we need to initialize the motor controller. We can do it in an initialize function, or do a quick shortcut and initialize it on the same line. We will add the following code to initialize it.

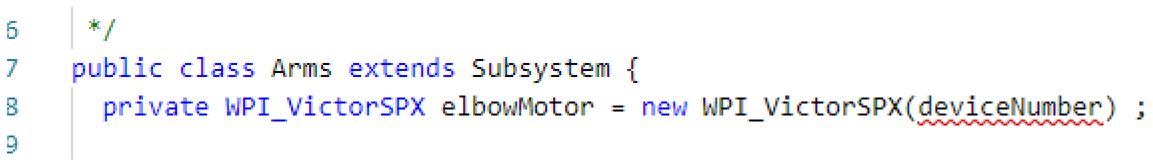
Start typing = new WPI and it will start autocompleting the class, with the different parameters.



```
public class Arms extends Subsystem {
  private WPI_VictorSPX elbowMotor = new WPI ;

  @Override
  public void initDefaultCommand() {
    // Set the default command for a subsyst
  }
```

WPI_VictorSPX(int deviceNumber)   com.ctre.phoeni…
WPI_TalonSRX(int deviceNumber)
WPILibVersion()
WindowPeer() Anonymous Inner Type
WritePendingException()

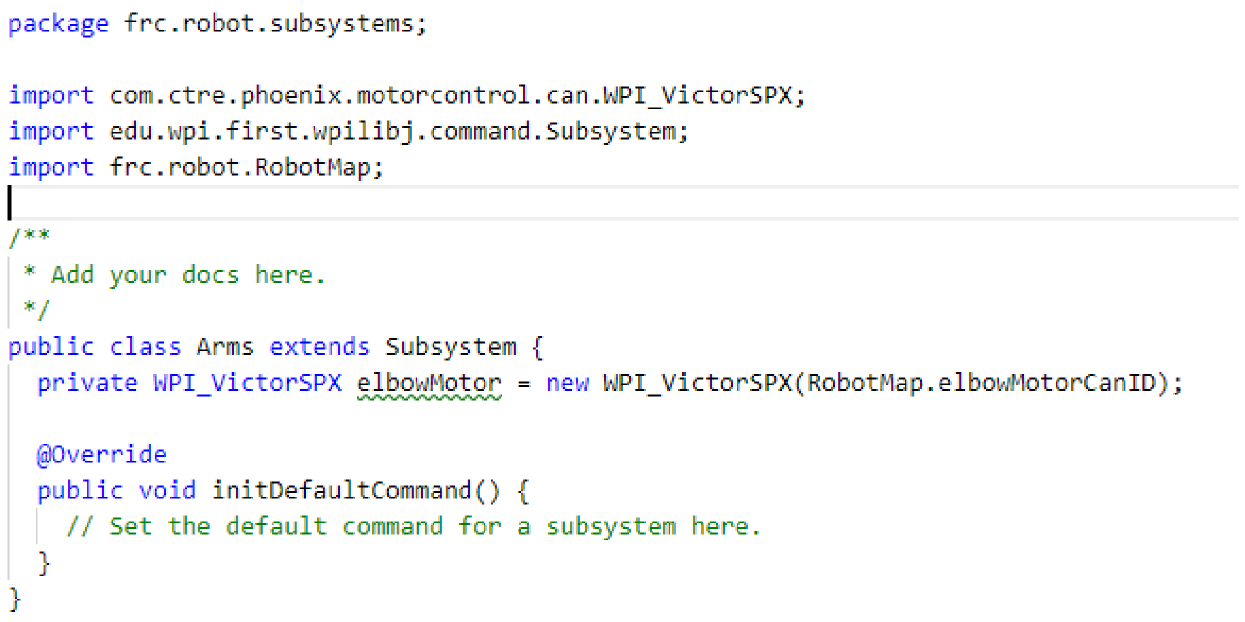
If you see the one in the list, you can now use your arrow keys or continue typing to filter down the list. If it is selected, hit Enter and it will autocomplete. Your code will look like this:



```
 */
public class Arms extends Subsystem {
  private WPI_VictorSPX elbowMotor = new WPI_VictorSPX(deviceNumber) ;

```

Again there is an error, but this time it is because it does not know of a variable called deviceNumber. We will replace it with the variable from the robot map. We can use our shortcuts to select RobotMap.elbowMotorCanID. When we select it, it automatically adds the import for us!

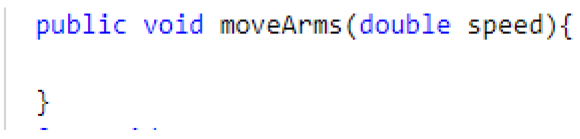
Your code should now look like this:



```
package frc.robot.subsystems;

import com.ctre.phoenix.motorcontrol.can.WPI_VictorSPX;
import edu.wpi.first.wpilibj.command.Subsystem;
import frc.robot.RobotMap;

/**
 * Add your docs here.
 */
public class Arms extends Subsystem {
  private WPI_VictorSPX elbowMotor = new WPI_VictorSPX(RobotMap.elbowMotorCanID);

  @Override
  public void initDefaultCommand() {
    // Set the default command for a subsystem here.
  }
}
```

Now we need to do something to control it. Let's make a function called MoveArms.

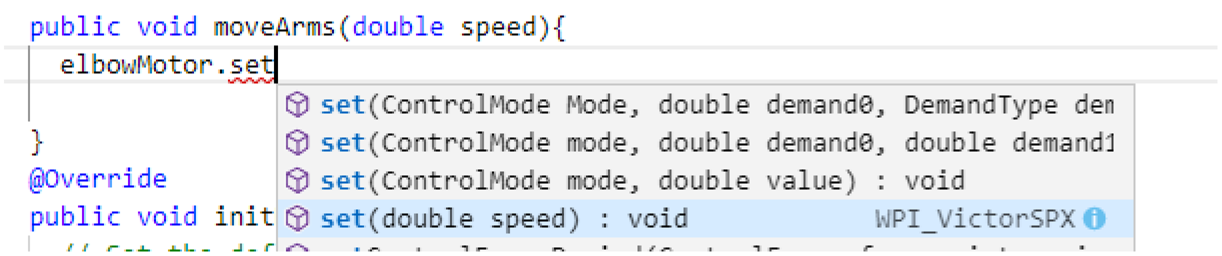
We will give it a parameter called Speed. It will need to be of type double, which is a number with decimals.



```
public void moveArms(double speed){

}
```

We can call the function, but it will do nothing. Now let's make it move the motor.

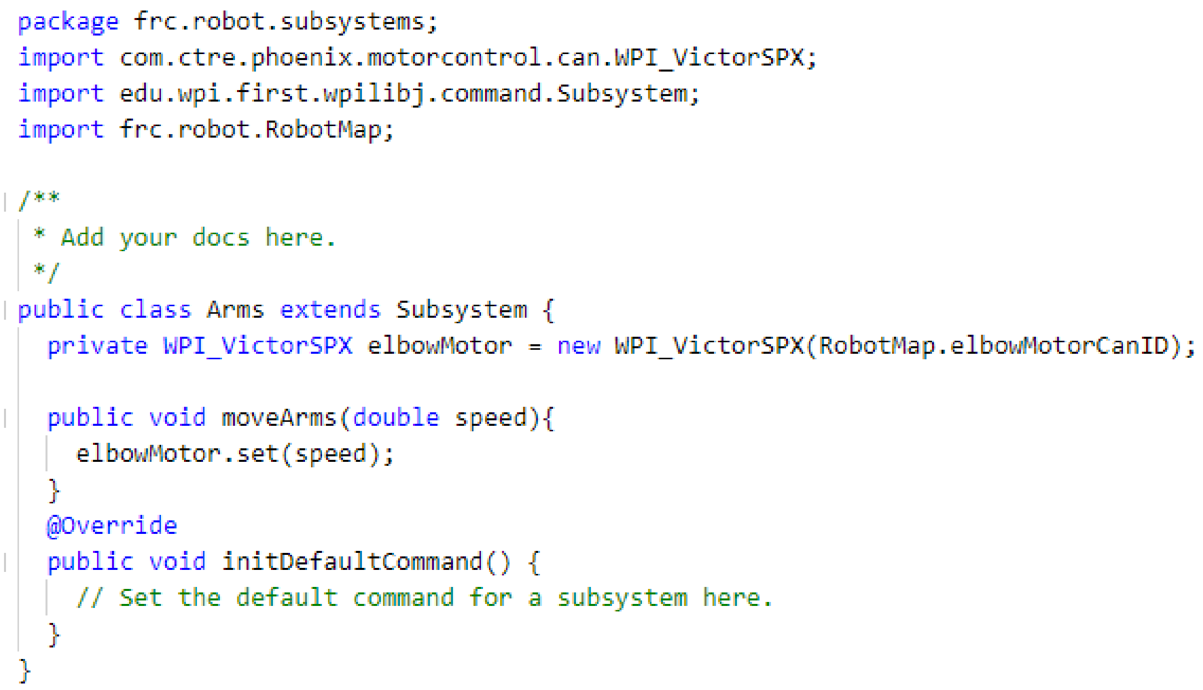
You can type in elbowMotor. and it will show all functions available. You can also use tab with the autocomplete if we are not done, so you can type in elbow then hit tab and it will fill in elbowMotor. Then hit the period and it will bring up the functions available. We will use the set function to set the speed of the motor.



```
public void moveArms(double speed){
  elbowMotor.set
                 set(ControlMode Mode, double demand0, DemandType dem
}                set(ControlMode mode, double demand0, double demand1
@Override        set(ControlMode mode, double value) : void
public void init set(double speed) : void                 WPI_VictorSPX
```

Hit enter or tab to set. Notice that our function name is the same as the default parameter, so it just maps it. If we called our parameter speed something else, we would have to replace it.

Now we have a subsystem that controls a motor!



```
package frc.robot.subsystems;
import com.ctre.phoenix.motorcontrol.can.WPI_VictorSPX;
import edu.wpi.first.wpilibj.command.Subsystem;
import frc.robot.RobotMap;

/**
 * Add your docs here.
 */
public class Arms extends Subsystem {
  private WPI_VictorSPX elbowMotor = new WPI_VictorSPX(RobotMap.elbowMotorCanID);

  public void moveArms(double speed){
    elbowMotor.set(speed);
  }
  @Override
  public void initDefaultCommand() {
    // Set the default command for a subsystem here.
  }
}
```

We have more motors to add to the subsystem, but we will finish creating this one motor to the end before we move on.

But before we can do any commands, we must initialize the subsystem on the robot!

Open your Robot.java file and you will find the initialization for the example subsystem. This should be at the top of the class, where other variables are being declared. Below it, add a line to initialize the arms subsystem. Make sure to use the tab auto complete so we can automatically add the Imports commands!

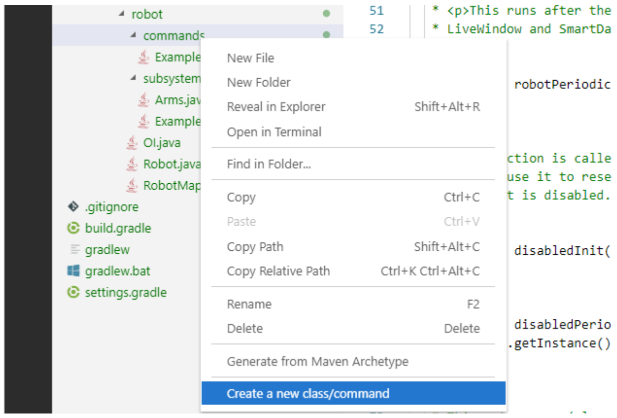
```
public static Arms arms = new Arms();
```

Now we can move onto the commands…

Lab 3 – Creating Arm Commands

We will create 2 commands. First command will be the RaiseArms, the other will be LowerArms.

Right click on the commands folder and select Create a new class/command.



```
* <p>This runs after the
* LiveWindow and SmartDa
robotPeriodic
ction is calle
use it to rese
t is disabled.
disabledInit(
disabledPerio
.getInstance()
```

Pick Command from the list and enter RaiseArms as the name.

Open RaiseArms.java when it is complete.

You will see many default functions.

RaiseArms() is the constructor, which is run when the object is created

initialize() is called just before the command runs for the first time

execute() is called when it is scheduled to run (running)

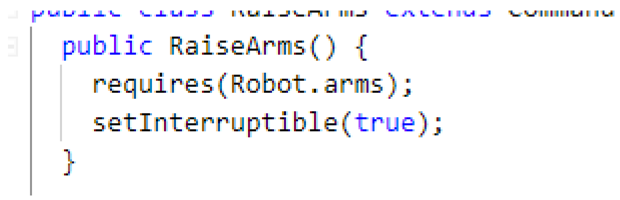
isFinished() checks to see if the command is finished. By default, it returns false every time, which means it is never finished. We could put logic in there to make other conditions to stop the command from running. For example, a limit switch, or checking an encoder value

end() is called after isFinished is hit. This means the command ran successfully. You might want to reset encoder values, stop motors, or other things when the command is done.

Interrupted() is called if the subsystem is interrupted with another command. You might want to mimic the end command or do something different. By default, a command is not interruptible.

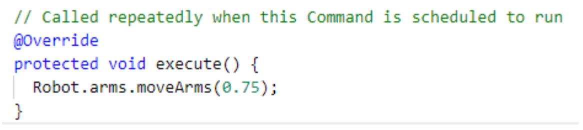
First, we will update the constructor with the subsystem requirements. Enter the following code, which says that it requires the arms subsystem from the robot.

We will also make this command interruptible by using the function setInterruptible. Your code should look like this at the end:



```
  public RaiseArms() {
    requires(Robot.arms);
    setInterruptible(true);
  }
```

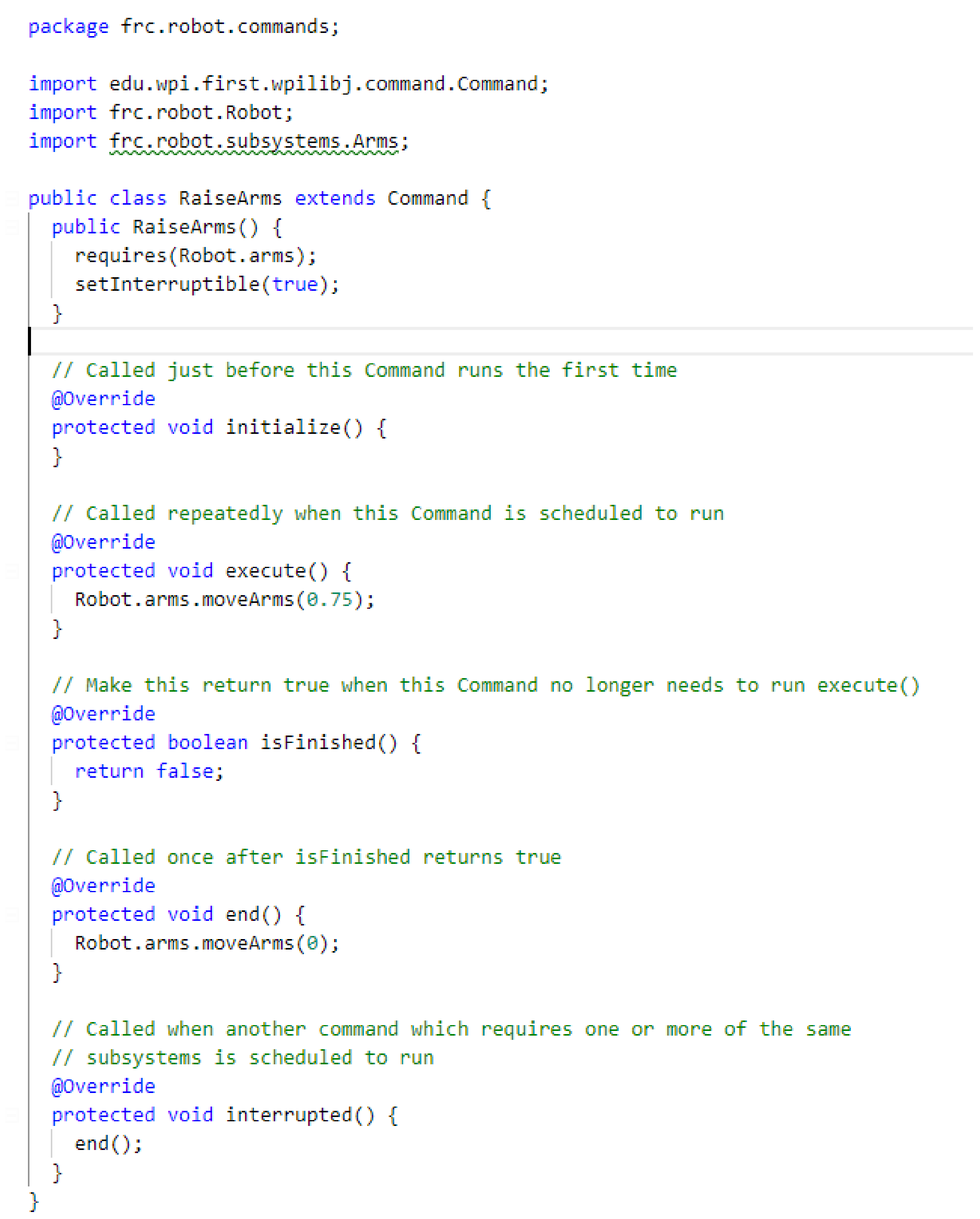
Now that we own the subsystem, we have to do something when it runs. We will run the motors. To do this you will call the function on the subsystem we created earlier, moveArms. We will move them up with a speed of 0.75.



```
// Called repeatedly when this Command is scheduled to run
@Override
protected void execute() {
  Robot.arms.moveArms(0.75);
}
```

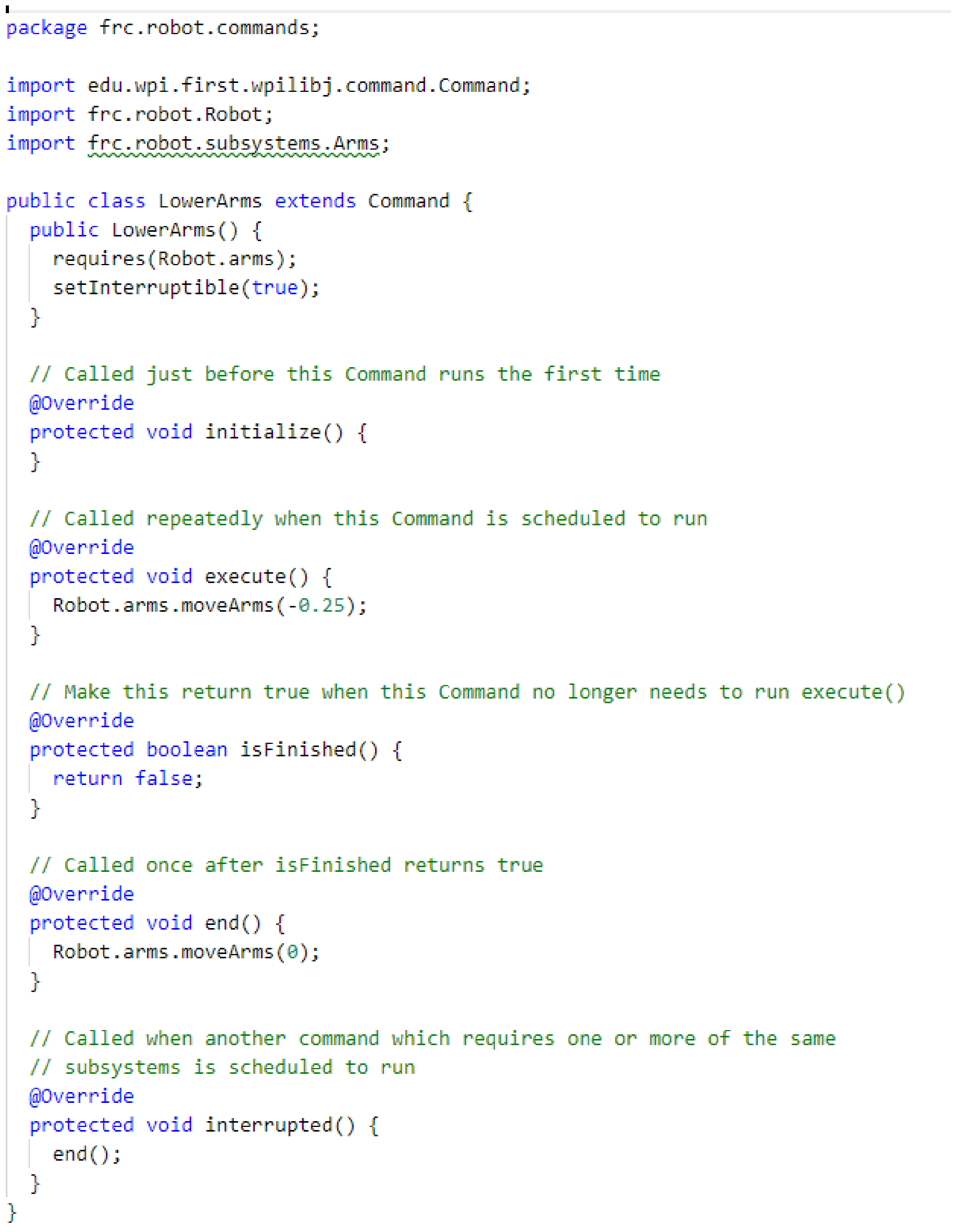
When it is done, we want to stop the motor. We will add the end() function to set the arm speed to 0. We can do the same for the interrupted command, but we will just call the end function from the interrupted function.

We now have a working command, that looks like this:



```java
package frc.robot.commands;

import edu.wpi.first.wpilibj.command.Command;
import frc.robot.Robot;
import frc.robot.subsystems.Arms;

public class RaiseArms extends Command {
  public RaiseArms() {
    requires(Robot.arms);
    setInterruptible(true);
  }

  // Called just before this Command runs the first time
  @Override
  protected void initialize() {
  }

  // Called repeatedly when this Command is scheduled to run
  @Override
  protected void execute() {
    Robot.arms.moveArms(0.75);
  }

  // Make this return true when this Command no longer needs to run execute()
  @Override
  protected boolean isFinished() {
    return false;
  }

  // Called once after isFinished returns true
  @Override
  protected void end() {
    Robot.arms.moveArms(0);
  }

  // Called when another command which requires one or more of the same
  // subsystems is scheduled to run
  @Override
  protected void interrupted() {
    end();
  }
}
```

Now we need to make a similar command for LowerArms. The only difference would be the names, and the motor values. We will have it move down at 0.25 speed, as gravity will help it go down.



```java
package frc.robot.commands;

import edu.wpi.first.wpilibj.command.Command;
import frc.robot.Robot;
import frc.robot.subsystems.Arms;

public class LowerArms extends Command {
  public LowerArms() {
    requires(Robot.arms);
    setInterruptible(true);
  }

  // Called just before this Command runs the first time
  @Override
  protected void initialize() {
  }

  // Called repeatedly when this Command is scheduled to run
  @Override
  protected void execute() {
    Robot.arms.moveArms(-0.25);
  }

  // Make this return true when this Command no longer needs to run execute()
  @Override
  protected boolean isFinished() {
    return false;
  }

  // Called once after isFinished returns true
  @Override
  protected void end() {
    Robot.arms.moveArms(0);
  }

  // Called when another command which requires one or more of the same
  // subsystems is scheduled to run
  @Override
  protected void interrupted() {
    end();
  }
}
```

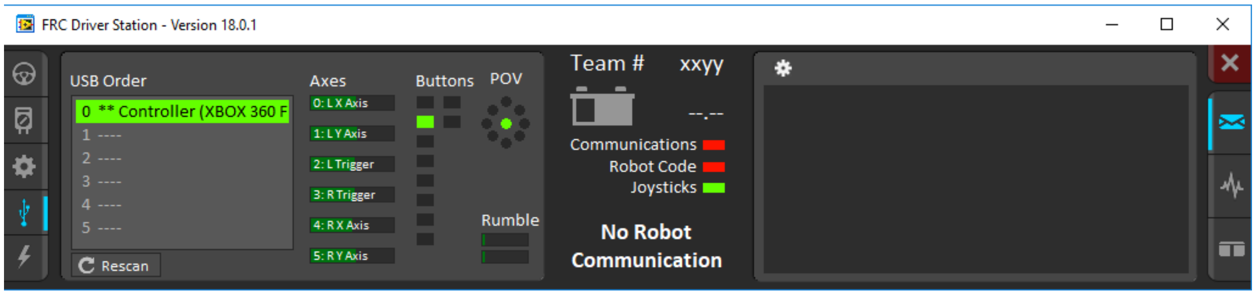
Now we have 2 commands and need to do something to make them fire. We will bind them to joystick buttons.

Lab 4 – Binding a Command to the Controller

We have our subsystem, we have our commands, now let's execute them from a controller. This is all done in the OI.java file, which stands for Operator Interface. All interface commands should be assigned in this file. It makes it easy to track down what does what.

There are many ways to attack this, but the best way is to define a variable for the button/axis like the robot map, and then declare and bind your buttons using those variables. This way you can access the buttons in another file if needed. For example, if you are driving and want to slow down when pressing B, you can check that button from the drive command execute function and slow it down if needed.

If you need to find out what buttons are which IDs, plug in a controller and run the driver station. Click on the USB, select the controller, and hit the button. The below example shows that I am hitting the B button, which is ID 2. A, which is the first button, would be ID 1.



First we need to declare the joystick that the buttons are on.

```java
public class OI {

    public static Joystick driverJoystick = new Joystick(0);
```

Next, we will declare objects for button A and B. They must go under the joystick, as they refer to the joystick.

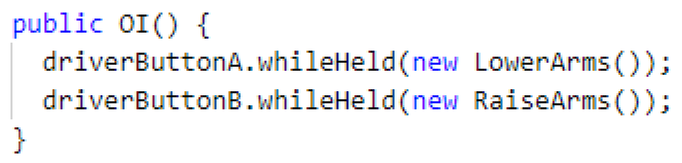
```java
public static Joystick driverJoystick = new Joystick(0);

public static JoystickButton driverButtonA = new JoystickButton(driverJoystick, 1);
public static JoystickButton driverButtonB  = new JoystickButton(driverJoystick, 2);
```

Notice we did not say final on the joystick or buttons, as we will be changing things on the them, and not having it read-only.

Now let's create our constructor and give the buttons something to do. You can run buttons a few different ways, whenPressed, whileHeld, whenReleased. These execute commands when those events happen. We will use the whileHeld function.

Notice when using tab to autofill the functions, you can then tab to the next variable highlighted to replace. This saves time:

Our final constructor will look like this:



```
public OI() {
    driverButtonA.whileHeld(new LowerArms());
    driverButtonB.whileHeld(new RaiseArms());
}
```

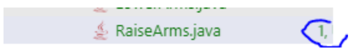
Now you can read what button does what command easily.

WhileHeld will start when the button is pressed and stop when the button is released. Our command says when ending the command to stop the motors, which is what our desired functionality will be.

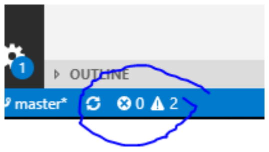
Now we need to test!

Lab 5 – Deploying the Robot

Before we can deploy the robot, we need to build our project, and make sure there are no errors. The three quick ways to see if there are no errors in your project are to first, check the colors of your files in your list. If any are red, they have errors to be looked at. Green means good.



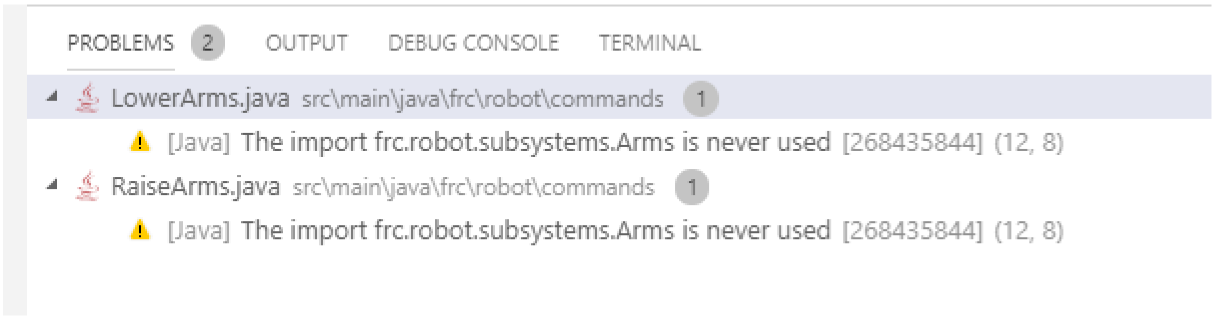
You can see next to the file how many problems, even if just warnings, are with the files.



You can also check the bottom left corner of the Visual Studio Code editor:

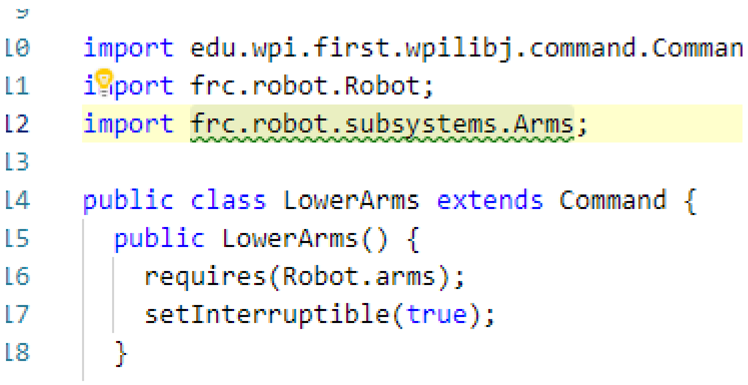
This shows our robot has 0 errors and 2 warnings. You can click on it to see what the issues are. Warnings won't stop a program, but have inefficiencies and should be cleaned up before the final robot deployment.

The warnings in my code are due to adding an Import which was never used:



```
LowerArms.java src\main\java\frc\robot\commands 1
    ⚠ [Java] The import frc.robot.subsystems.Arms is never used [268435844] (12, 8)
RaiseArms.java src\main\java\frc\robot\commands 1
    ⚠ [Java] The import frc.robot.subsystems.Arms is never used [268435844] (12, 8)
```

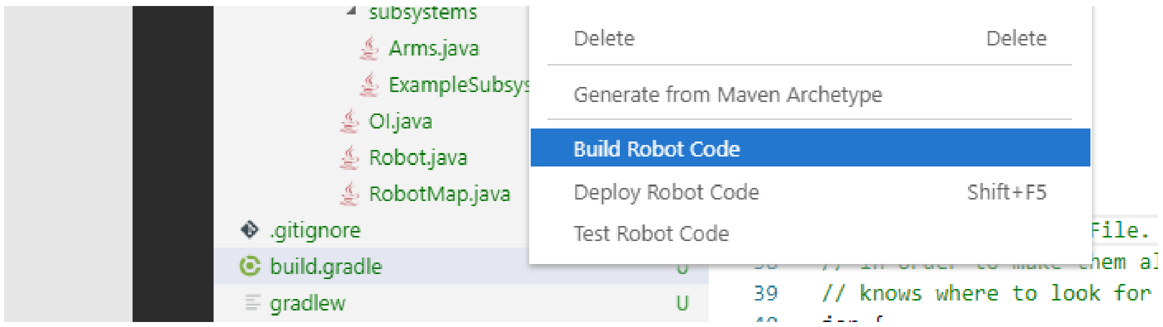
You can double click on it to go to the issue



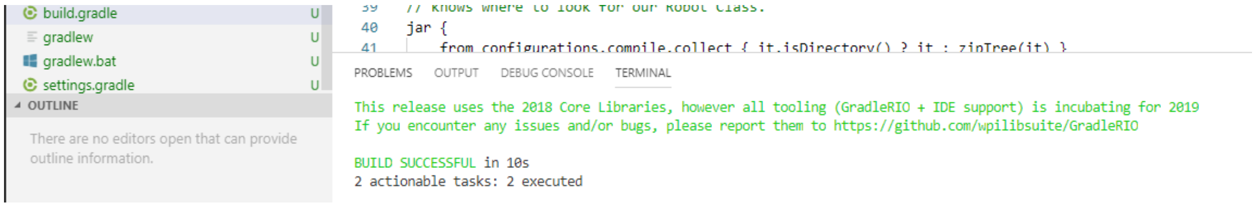
```
import edu.wpi.first.wpilibj.command.Comman
import frc.robot.Robot;
import frc.robot.subsystems.Arms;

public class LowerArms extends Command {
  public LowerArms() {
    requires(Robot.arms);
    setInterruptible(true);
  }
```

Because I am calling the subsystem through the Robot.arms static variable, and not the subsystem directly, we do not need to import the subsystem. I would delete those lines to clear the warnings.

To build our robot, right click on the build.gradle file and select build robot code.



An output window will popup that you can watch and see if the build was completed successfully or not



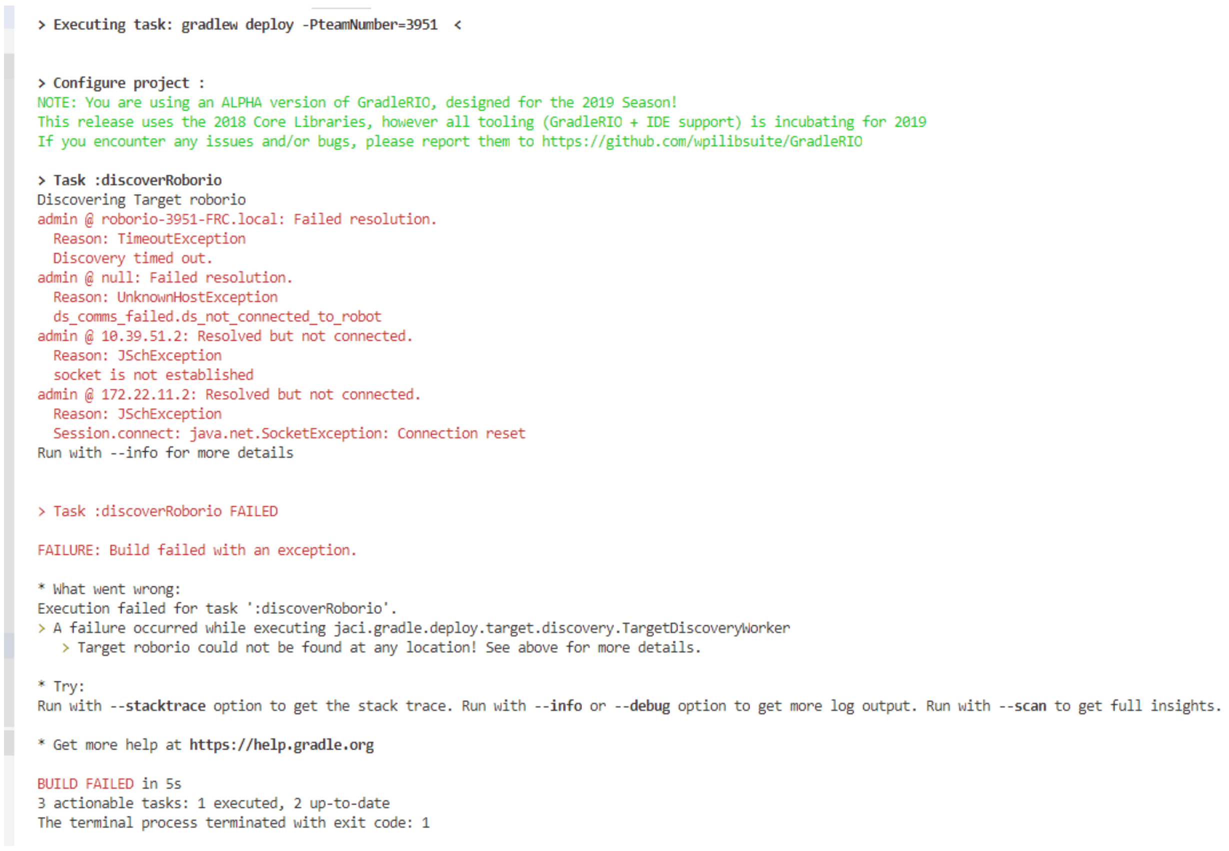
```
jar {
    from configurations.compile.collect { it.isDirectory() ? it : zipTree(it) }
```

```
This release uses the 2018 Core Libraries, however all tooling (GradleRIO + IDE support) is incubating for 2019
If you encounter any issues and/or bugs, please report them to https://github.com/wpilibsuite/GradleRIO

BUILD SUCCESSFUL in 10s
2 actionable tasks: 2 executed
```

The robot completed its build in 10 seconds, and was successful.

The alpha test, which is what this document is built on, has the test disabled.

So now let us deploy the robot by right clicking on the build.gradle and selecting deploy robot. I do not have the robot connected right now, which is causing the robot build to fail. You can see that in the output, and might need to expand the window.



```
> Executing task: gradlew deploy -PteamNumber=3951 <


> Configure project :
NOTE: You are using an ALPHA version of GradleRIO, designed for the 2019 Season!
This release uses the 2018 Core Libraries, however all tooling (GradleRIO + IDE support) is incubating for 2019
If you encounter any issues and/or bugs, please report them to https://github.com/wpilibsuite/GradleRIO

> Task :discoverRoborio
Discovering Target roborio
admin @ roborio-3951-FRC.local: Failed resolution.
  Reason: TimeoutException
  Discovery timed out.
admin @ null: Failed resolution.
  Reason: UnknownHostException
  ds_comms_failed.ds_not_connected_to_robot
admin @ 10.39.51.2: Resolved but not connected.
  Reason: JSchException
  socket is not established
admin @ 172.22.11.2: Resolved but not connected.
  Reason: JSchException
  Session.connect: java.net.SocketException: Connection reset
Run with --info for more details


> Task :discoverRoborio FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':discoverRoborio'.
> A failure occurred while executing jaci.gradle.deploy.target.discovery.TargetDiscoveryWorker
   > Target roborio could not be found at any location! See above for more details.

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output. Run with --scan to get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 5s
3 actionable tasks: 1 executed, 2 up-to-date
The terminal process terminated with exit code: 1
```

As you can see, it tries to connect to the robot from DNS and then IP address, but could not connect and failed deployment.

When the robot is connected, you will be able to deploy your code and then run the robot when done. Test your buttons to make sure things work. If the up and down is in reverse, then you can change the values in the Up and Down commands from positive to negative, rebuild, and redeploy.

Lab 6 – Adding the Arm Wheels to the Arm Subsystem

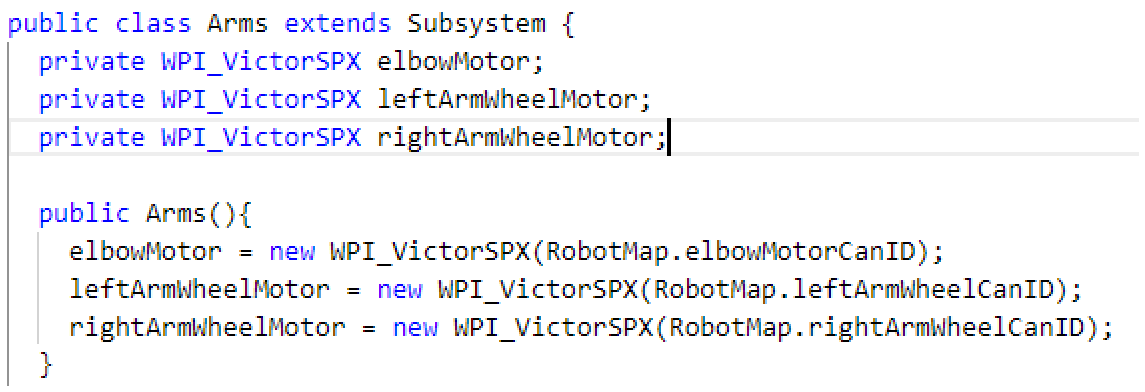
We will speed through this lab, as each of these items are explained above, just more of it.

We already declared the IDs for the subsystem earlier, so let's just move to create the motor controllers.

```
private WPI_VictorSPX leftArmWheelMotor = new WPI_VictorSPX(RobotMap.leftArmWheelCanID);
private WPI_VictorSPX rightArmWheelMotor = new WPI_VictorSPX(RobotMap.rightArmWheelCanID);
```

We want to invert one of the motors. This should be done when the subsystem is constructed. Let's use the constructor to initialize our motors, to make sure things get done in the proper order. You will delete the code after leftArmWheelMotor and rightArmWheelMotor, adding a ; to it, and moving the code into a constructor. From there, we will set one of the motors as inverted.

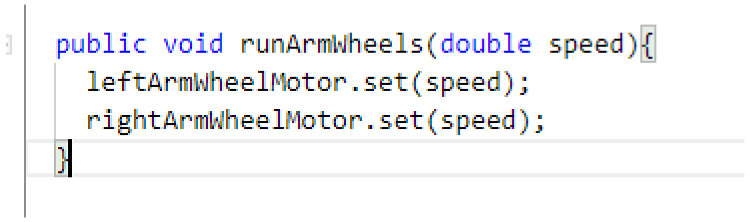
We can also move the elbow motor for consistency sake into the constructor. Your code should look like this:



```
public class Arms extends Subsystem {
  private WPI_VictorSPX elbowMotor;
  private WPI_VictorSPX leftArmWheelMotor;
  private WPI_VictorSPX rightArmWheelMotor;

  public Arms(){
    elbowMotor = new WPI_VictorSPX(RobotMap.elbowMotorCanID);
    leftArmWheelMotor = new WPI_VictorSPX(RobotMap.leftArmWheelCanID);
    rightArmWheelMotor = new WPI_VictorSPX(RobotMap.rightArmWheelCanID);
  }
```

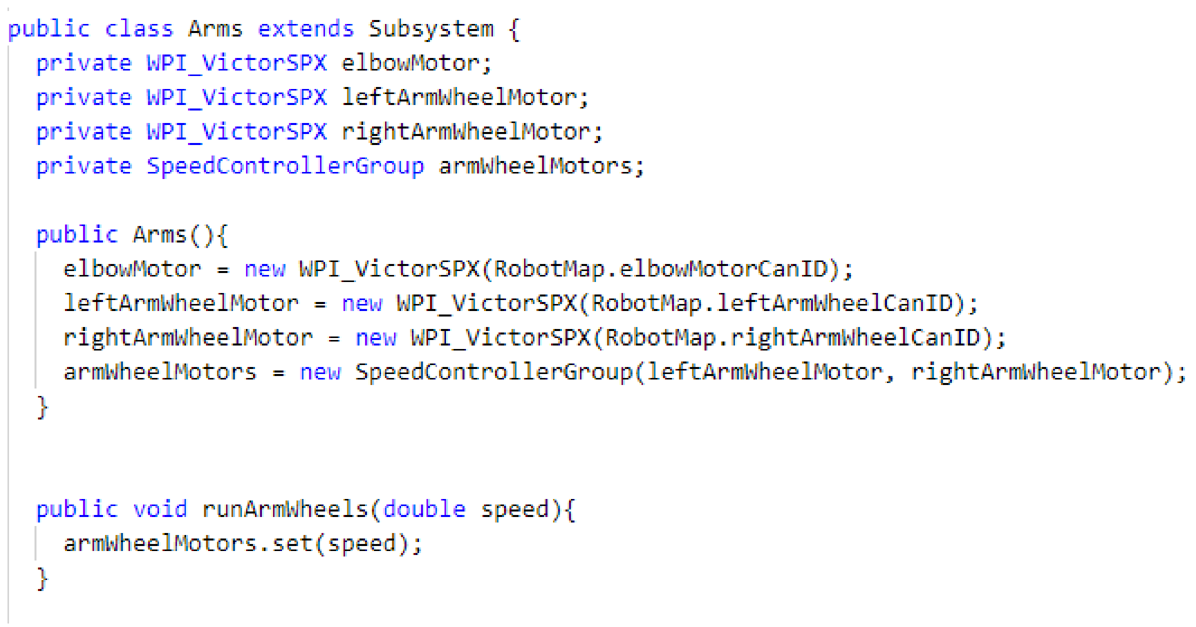
It will declare the variables, but then set them in the constructor. You can set other flags on the motor controllers there as well that cannot be set on the motor controller during declaration.

So we will add a function to run both of our motors with the set speed:



```
  public void runArmWheels(double speed){
    leftArmWheelMotor.set(speed);
    rightArmWheelMotor.set(speed);
  }
```

To prevent someone from using one motor but not the other, we can build a SpeedControllerGroup and control them as one. It will require a little rework:



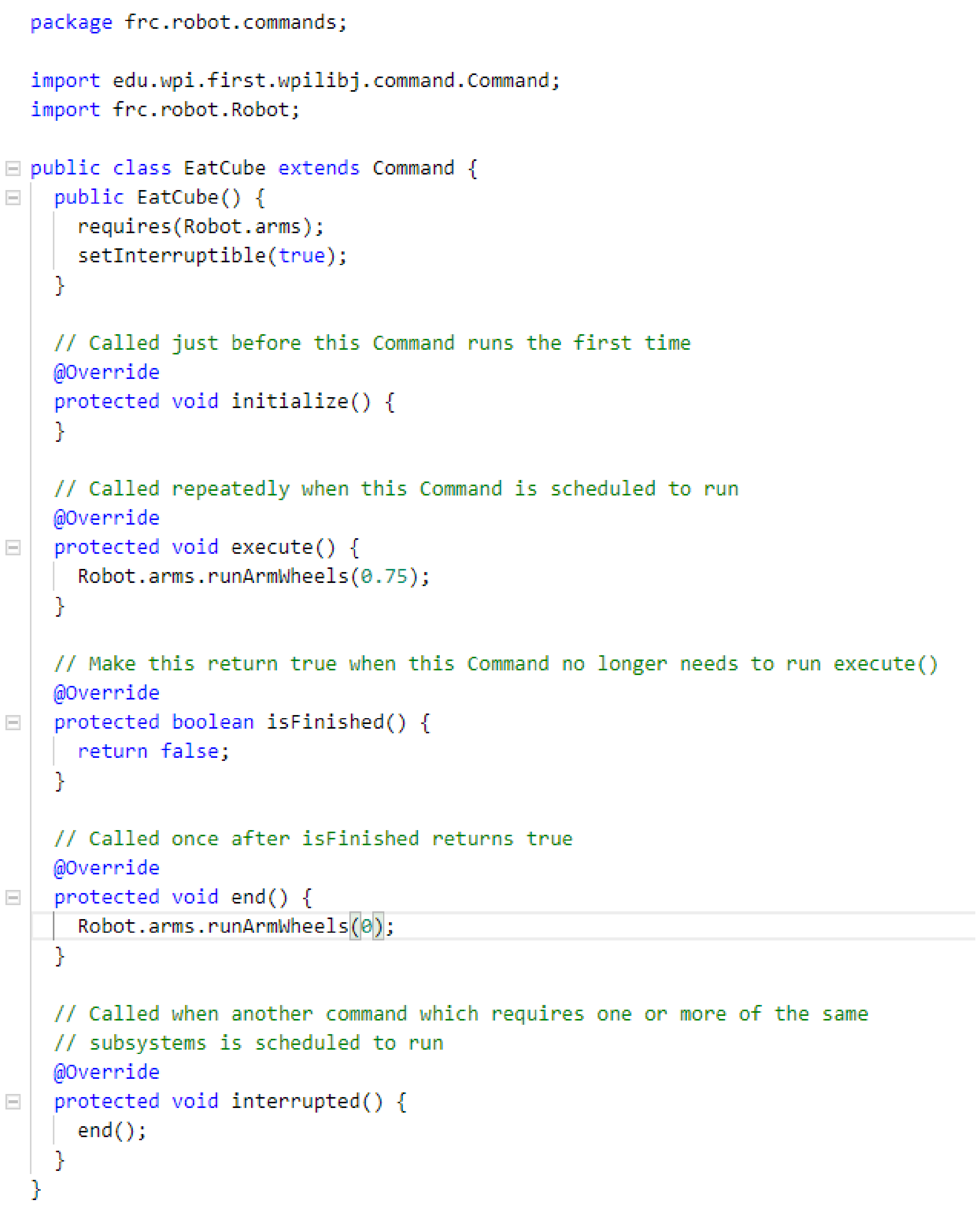
```java
public class Arms extends Subsystem {
  private WPI_VictorSPX elbowMotor;
  private WPI_VictorSPX leftArmWheelMotor;
  private WPI_VictorSPX rightArmWheelMotor;
  private SpeedControllerGroup armWheelMotors;

  public Arms(){
    elbowMotor = new WPI_VictorSPX(RobotMap.elbowMotorCanID);
    leftArmWheelMotor = new WPI_VictorSPX(RobotMap.leftArmWheelCanID);
    rightArmWheelMotor = new WPI_VictorSPX(RobotMap.rightArmWheelCanID);
    armWheelMotors = new SpeedControllerGroup(leftArmWheelMotor, rightArmWheelMotor);
  }


  public void runArmWheels(double speed){
    armWheelMotors.set(speed);
  }
```

We declare the SpeedControllerGroup, and assign the motors to it. Then we can control them as one. Having the code in the constructor ensures the order of initialization is done properly.

Now let’s build our commands, EatCube and SpitCube. Notice the command names are different from the motor classes. This is common, as you want your command to be an action that you are doing, which could be one or many things. Your function inside of the subsystem performs the commands you want on the motors.

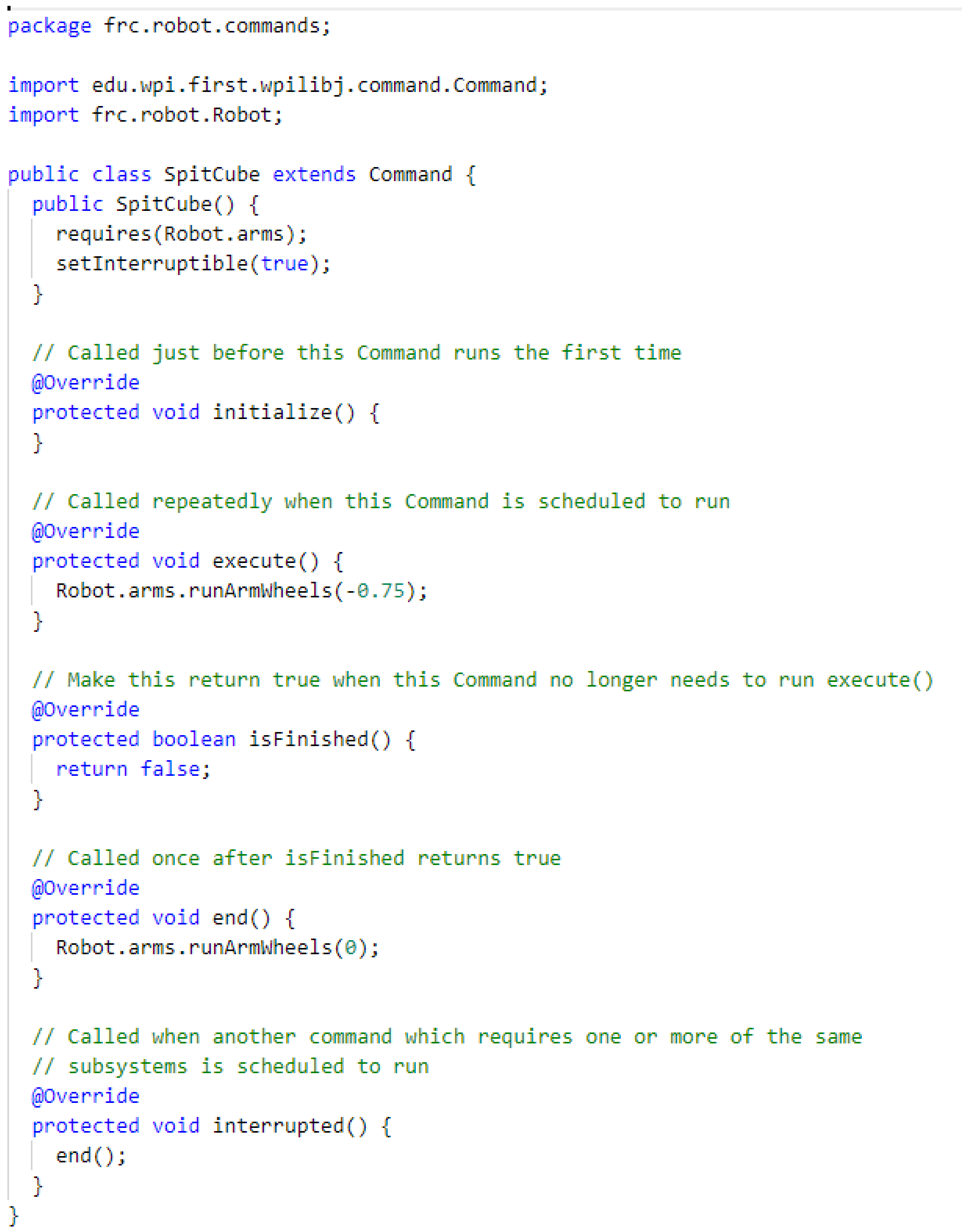
The easiest way to build a command that is like another command, is to right click on the command you want to copy, select copy, then right click on the commands folder and hit paste. It will create a folder with a 1 in the filename. It will throw a bunch of errors because the classes are duplicates. You can change the values and move on. Do it with the RaiseArms command to build our EatCube, then change

- Change class name from RaiseArms to EatCube
- Change constructor from RaiseArms to EatCube
- Change the execute and end functions to run the runArmWheels instead of moveArms functions



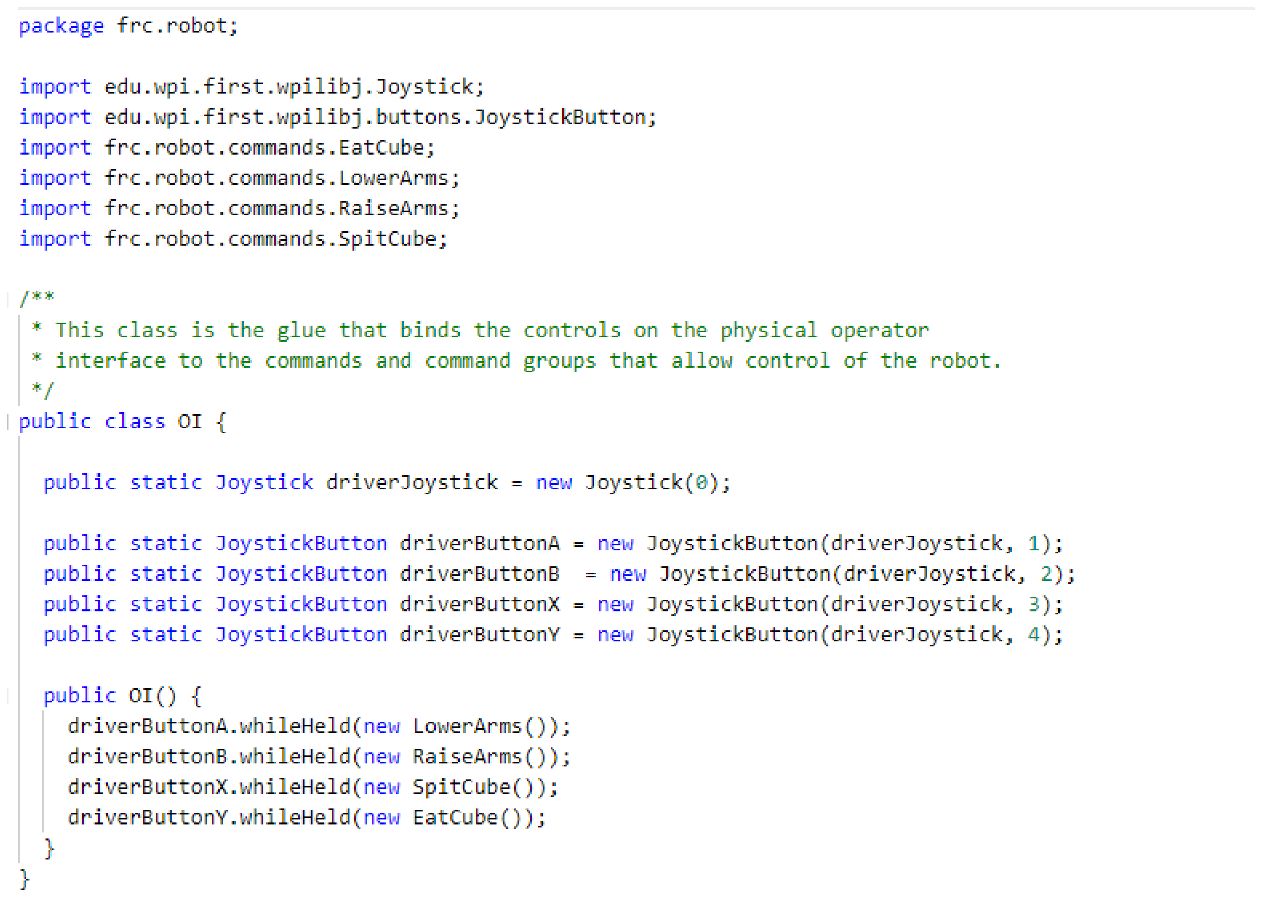
```java
package frc.robot.commands;

import edu.wpi.first.wpilibj.command.Command;
import frc.robot.Robot;

public class EatCube extends Command {
  public EatCube() {
    requires(Robot.arms);
    setInterruptible(true);
  }

  // Called just before this Command runs the first time
  @Override
  protected void initialize() {
  }

  // Called repeatedly when this Command is scheduled to run
  @Override
  protected void execute() {
    Robot.arms.runArmWheels(0.75);
  }

  // Make this return true when this Command no longer needs to run execute()
  @Override
  protected boolean isFinished() {
    return false;
  }

  // Called once after isFinished returns true
  @Override
  protected void end() {
    Robot.arms.runArmWheels(0);
  }

  // Called when another command which requires one or more of the same
  // subsystems is scheduled to run
  @Override
  protected void interrupted() {
    end();
  }
}
```

Now do the same for SpitCube, copying the EatCube. You will have to change the motor value from a positive to negative number.



```java
package frc.robot.commands;

import edu.wpi.first.wpilibj.command.Command;
import frc.robot.Robot;

public class SpitCube extends Command {
  public SpitCube() {
    requires(Robot.arms);
    setInterruptible(true);
  }

  // Called just before this Command runs the first time
  @Override
  protected void initialize() {
  }

  // Called repeatedly when this Command is scheduled to run
  @Override
  protected void execute() {
    Robot.arms.runArmWheels(-0.75);
  }

  // Make this return true when this Command no longer needs to run execute()
  @Override
  protected boolean isFinished() {
    return false;
  }

  // Called once after isFinished returns true
  @Override
  protected void end() {
    Robot.arms.runArmWheels(0);
  }

  // Called when another command which requires one or more of the same
  // subsystems is scheduled to run
  @Override
  protected void interrupted() {
    end();
  }
}
```

Now we will bind the commands to two new buttons, X (3) and Y (4). Our OI.Java file will look like this:



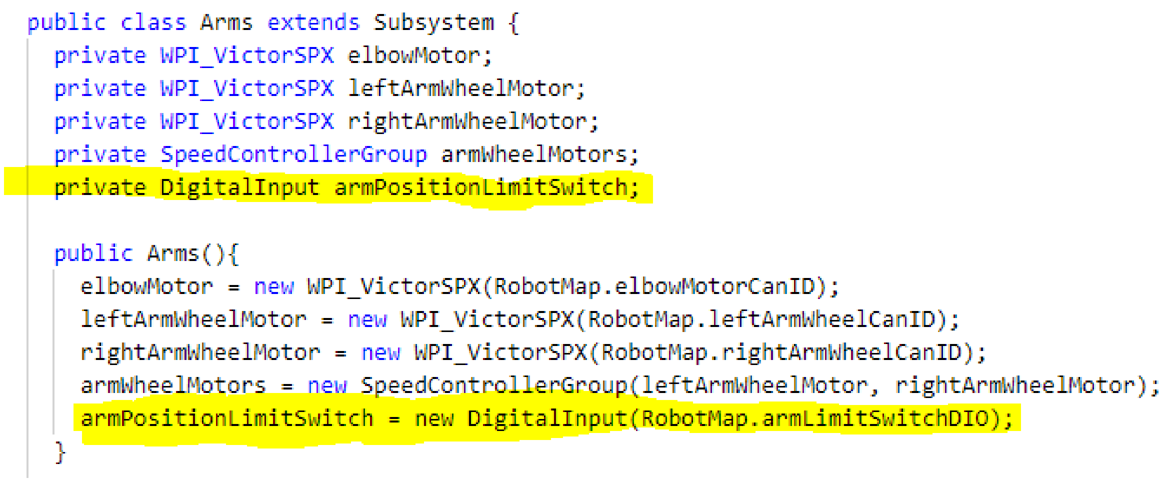
```java
package frc.robot;

import edu.wpi.first.wpilibj.Joystick;
import edu.wpi.first.wpilibj.buttons.JoystickButton;
import frc.robot.commands.EatCube;
import frc.robot.commands.LowerArms;
import frc.robot.commands.RaiseArms;
import frc.robot.commands.SpitCube;

/**
 * This class is the glue that binds the controls on the physical operator
 * interface to the commands and command groups that allow control of the robot.
 */
public class OI {

  public static Joystick driverJoystick = new Joystick(0);

  public static JoystickButton driverButtonA = new JoystickButton(driverJoystick, 1);
  public static JoystickButton driverButtonB  = new JoystickButton(driverJoystick, 2);
  public static JoystickButton driverButtonX = new JoystickButton(driverJoystick, 3);
  public static JoystickButton driverButtonY = new JoystickButton(driverJoystick, 4);

  public OI() {
    driverButtonA.whileHeld(new LowerArms());
    driverButtonB.whileHeld(new RaiseArms());
    driverButtonX.whileHeld(new SpitCube());
    driverButtonY.whileHeld(new EatCube());
  }
}
```

Now we are good to test and see that the buttons are mapped properly. Again, you might have to change the motors to reverse the direction of one motor or the other.

Lab 7 - Stopping a Command with a Sensor

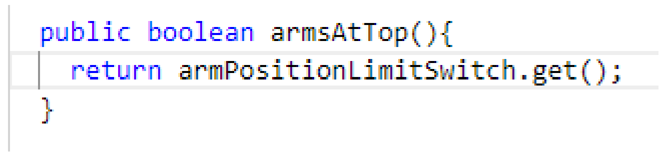
We want to use sensors to do many things on our robot. The most important would be to stop a motor when it hits a specific point. We will modify our Arms to stop when it hits the limit switch when lifting. We also want output to see when the sensor is hit, so we can verify our code without running the switch.

First, we need to declare the Sensor object, and give it a function to call the sensor in the Arms.java file. We will be using the DigitalInput class.



```java
public class Arms extends Subsystem {
  private WPI_VictorSPX elbowMotor;
  private WPI_VictorSPX leftArmWheelMotor;
  private WPI_VictorSPX rightArmWheelMotor;
  private SpeedControllerGroup armWheelMotors;
  private DigitalInput armPositionLimitSwitch;

  public Arms(){
    elbowMotor = new WPI_VictorSPX(RobotMap.elbowMotorCanID);
    leftArmWheelMotor = new WPI_VictorSPX(RobotMap.leftArmWheelCanID);
    rightArmWheelMotor = new WPI_VictorSPX(RobotMap.rightArmWheelCanID);
    armWheelMotors = new SpeedControllerGroup(leftArmWheelMotor, rightArmWheelMotor);
    armPositionLimitSwitch = new DigitalInput(RobotMap.armLimitSwitchDIO);
  }
```

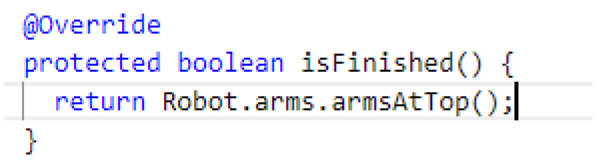
Now we will create a function. We will use a more descriptive function name. It will be a return type of Boolean, which is true/false.



```java
public boolean armsAtTop(){
  return armPositionLimitSwitch.get();
}
```

This function would return true if the switch is hit. So in English: if you ask the subsystem if the arms are at the top? If switch is hit, yes. If not, no.

Now we can implement the sensor in the RaiseArms command. Modify the isFinished() function to check the sensor:



```java
@Override
protected boolean isFinished() {
  return Robot.arms.armsAtTop();
}
```

This would say “I am finished if the arms are at the top”

We might think we are done. The command will stop when the arms are at the top. But what if another command calls the arm wheels and does not make the check? We should add a safety check in the motor file. This is another good reason to split out our moveArms into a raiseArms and lowerArms functions. We can build the checks into that, rather than giving direct control of the motor from a command.

Remove the moveArms function, and add the raiseArms and lowerArms functions. Now we need a stop function. We can stop all functions with this, ensuring everything is not running in the subsystem.

We will also force the value to be a positive value for raise, and negative for lower.

```java
public void raiseArms(double speed){
  //if the arms are at the top, stop the motors.
  if(armsAtTop())
    speed = 0;
  //if speed is negative, make positive as we want to run the motor up.
  if(speed < 0)
    speed = speed * -1;
  elbowMotor.set(speed);
}

public void lowerArms(double speed){
  //if speed is a positive number, make it negative to run the motor down.
  if(speed > 0)
    speed = speed * -1;
  elbowMotor.set(speed);
}

public void stop(){
  elbowMotor.set(0);
  armWheelMotors.set(0);
}
```

Then update the execute function and stop functions of the command RaiseArms and LowerArms. Your stop will look like this:

```java
// Called once after isFinished returns true
@Override
protected void end() {
  Robot.arms.stop();
}
```

And Raise Arms execute function

```java
// Called repeatedly when this Command is scheduled to run
@Override
protected void execute() {
  Robot.arms.raiseArms(0.75);
}
```

And Lower Arms execute function

```java
// Called repeatedly when this Command is scheduled to run
@Override
protected void execute() {
  Robot.arms.lowerArms(0.25);
}
```

We could do the same with the arm wheels for eat and spit. Give it a try, as it will mimic what we just did for the arms, just without a sensor.

Lab 8 – Building the Drive Subsystem

The drive subsystem is a more complicated sub system than our arms. More motors! But it is actually a bit easier.

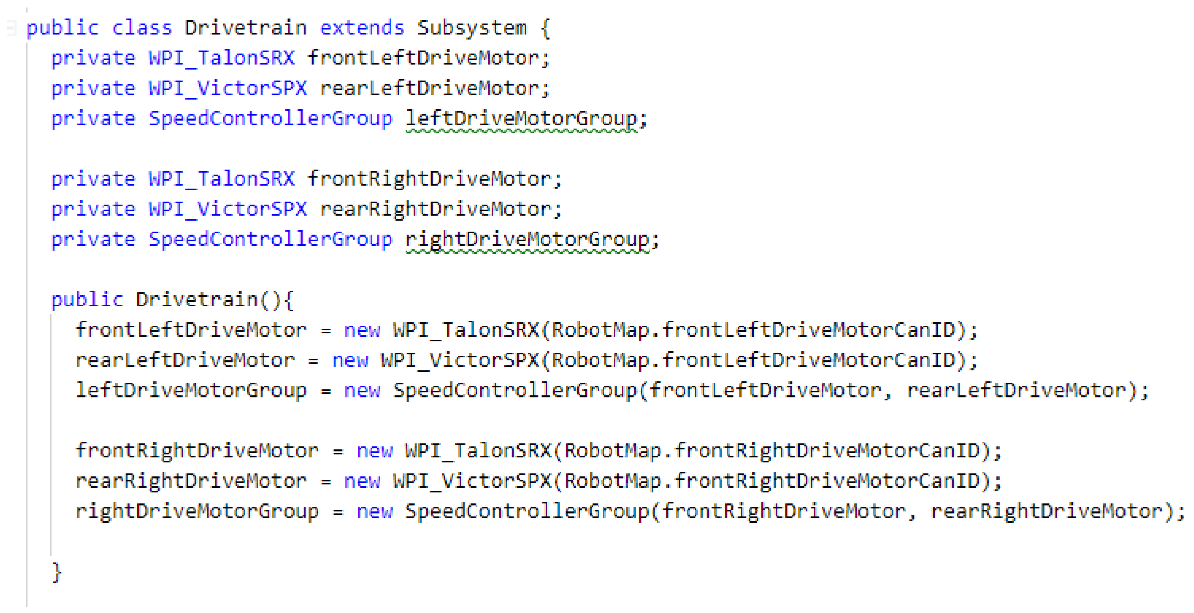
We will be using the standard Arcade Drive, using the left thumb stick to move the robot around.

First we need to declare our Can ID's in the RobotMap for our motors.

```java
//declare Ids for our drivetrain subsystem
public static final int frontLeftDriveMotorCanID = 1;
public static final int rearLeftDriveMotorCanID = 2;
public static final int frontRightDriveMotorCanID = 3;
public static final int rearRightDriveMotorCanID = 4;
```

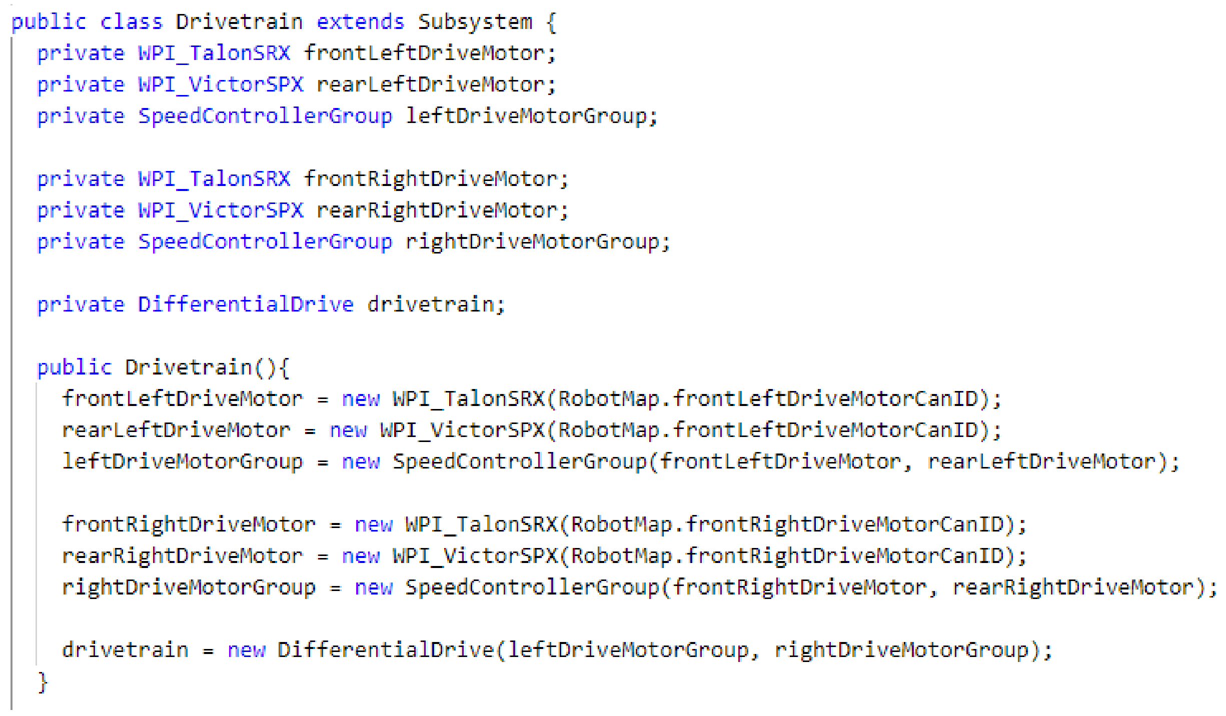
First, we need to build a new subsystem called Drivetrain.

We will declare our 4 motor controllers and 2 SpeedControllerGroups – one for the left side and one for the right.



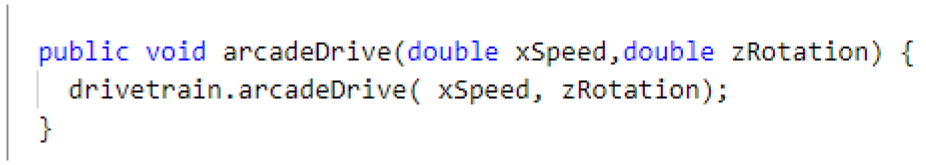
```java
public class Drivetrain extends Subsystem {
  private WPI_TalonSRX frontLeftDriveMotor;
  private WPI_VictorSPX rearLeftDriveMotor;
  private SpeedControllerGroup leftDriveMotorGroup;

  private WPI_TalonSRX frontRightDriveMotor;
  private WPI_VictorSPX rearRightDriveMotor;
  private SpeedControllerGroup rightDriveMotorGroup;

  public Drivetrain(){
    frontLeftDriveMotor = new WPI_TalonSRX(RobotMap.frontLeftDriveMotorCanID);
    rearLeftDriveMotor = new WPI_VictorSPX(RobotMap.frontLeftDriveMotorCanID);
    leftDriveMotorGroup = new SpeedControllerGroup(frontLeftDriveMotor, rearLeftDriveMotor);

    frontRightDriveMotor = new WPI_TalonSRX(RobotMap.frontRightDriveMotorCanID);
    rearRightDriveMotor = new WPI_VictorSPX(RobotMap.frontRightDriveMotorCanID);
    rightDriveMotorGroup = new SpeedControllerGroup(frontRightDriveMotor, rearRightDriveMotor);

  }
```

Now we will use the DifferentialDrive class to put our drivetrain together.



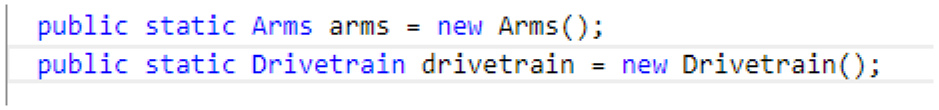
```java
public class Drivetrain extends Subsystem {
  private WPI_TalonSRX frontLeftDriveMotor;
  private WPI_VictorSPX rearLeftDriveMotor;
  private SpeedControllerGroup leftDriveMotorGroup;

  private WPI_TalonSRX frontRightDriveMotor;
  private WPI_VictorSPX rearRightDriveMotor;
  private SpeedControllerGroup rightDriveMotorGroup;

  private DifferentialDrive drivetrain;

  public Drivetrain(){
    frontLeftDriveMotor = new WPI_TalonSRX(RobotMap.frontLeftDriveMotorCanID);
    rearLeftDriveMotor = new WPI_VictorSPX(RobotMap.frontLeftDriveMotorCanID);
    leftDriveMotorGroup = new SpeedControllerGroup(frontLeftDriveMotor, rearLeftDriveMotor);

    frontRightDriveMotor = new WPI_TalonSRX(RobotMap.frontRightDriveMotorCanID);
    rearRightDriveMotor = new WPI_VictorSPX(RobotMap.frontRightDriveMotorCanID);
    rightDriveMotorGroup = new SpeedControllerGroup(frontRightDriveMotor, rearRightDriveMotor);

    drivetrain = new DifferentialDrive(leftDriveMotorGroup, rightDriveMotorGroup);
  }
```

Now, add a function to use the drive type requested. We will use Arcade Drive.



```java
public void arcadeDrive(double xSpeed,double zRotation) {
  drivetrain.arcadeDrive( xSpeed, zRotation);
}
```

Now that we have our subsystem, we have to link it in the Robot.java file.



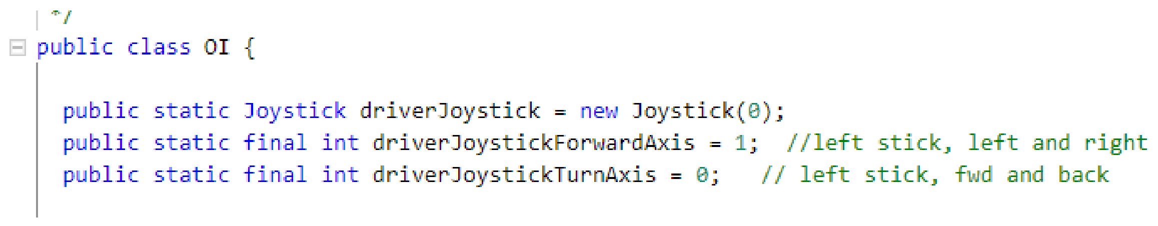
```java
public static Arms arms = new Arms();
public static Drivetrain drivetrain = new Drivetrain();
```

Now we have a subsystem that we can use to drive the robot. Onto the next lesson, making it drive.

Lab 9 – Making the robot drive with the controller

Now that we have our drive subsystem, we need to make it drive. We want to drive all of the time, so we will use the initDefaultCommand() function to set our new default drive command.

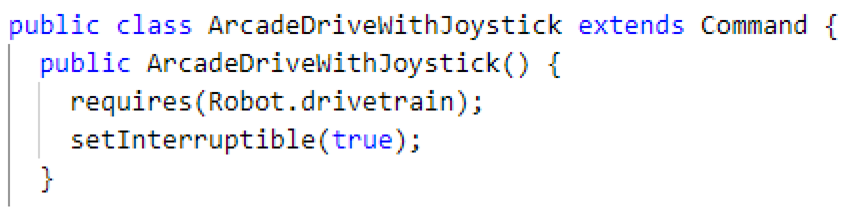
We will create variables in the OI.java file to refer to the axis we want to use to drive:



```
 */
public class OI {

  public static Joystick driverJoystick = new Joystick(0);
  public static final int driverJoystickForwardAxis = 1;  //left stick, left and right
  public static final int driverJoystickTurnAxis = 0;   // left stick, fwd and back
```

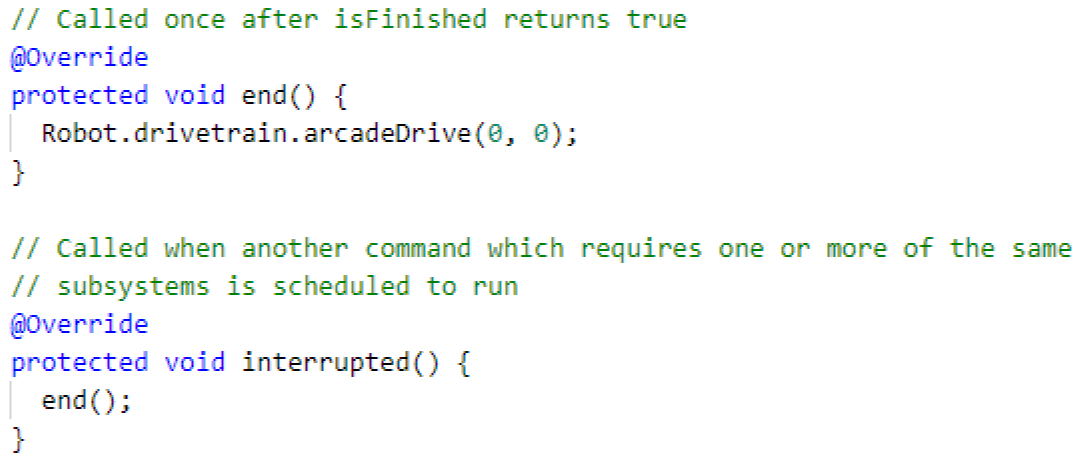
Next, we a new command called ArcadeDriveWithJoystick.

In the constructor, we want to take over the drivetrain, and set it as interruptible.



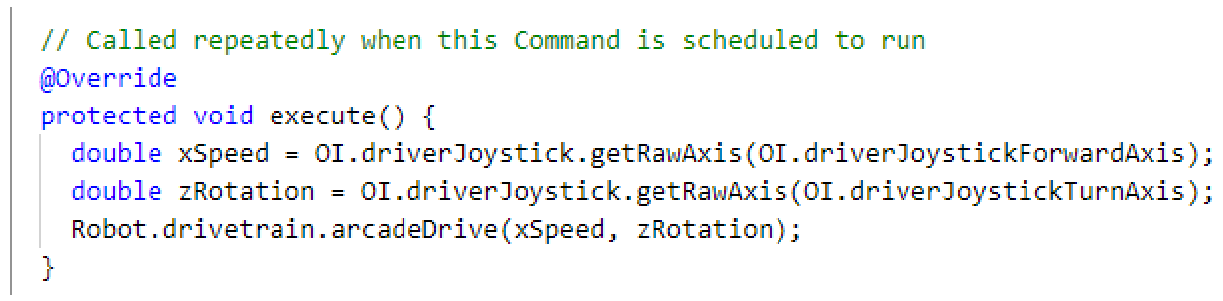
```
public class ArcadeDriveWithJoystick extends Command {
  public ArcadeDriveWithJoystick() {
    requires(Robot.drivetrain);
    setInterruptible(true);
  }
```

Update the end function to stop the drivetrain



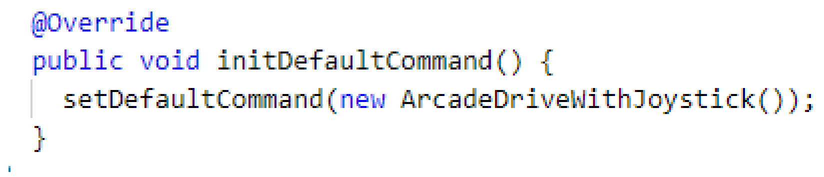
```
// Called once after isFinished returns true
@Override
protected void end() {
  Robot.drivetrain.arcadeDrive(0, 0);
}

// Called when another command which requires one or more of the same
// subsystems is scheduled to run
@Override
protected void interrupted() {
  end();
}
```

For the execute function, we will pull out the speed and rotation into variables, and then set them to the raw axis of the joystick.



```java
// Called repeatedly when this Command is scheduled to run
@Override
protected void execute() {
  double xSpeed = OI.driverJoystick.getRawAxis(OI.driverJoystickForwardAxis);
  double zRotation = OI.driverJoystick.getRawAxis(OI.driverJoystickTurnAxis);
  Robot.drivetrain.arcadeDrive(xSpeed, zRotation);
}
```

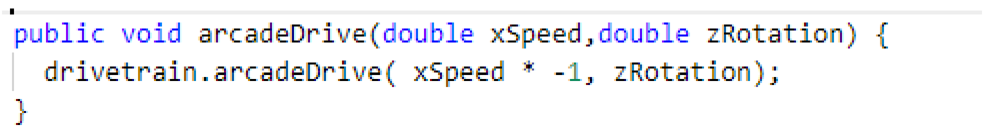
To drive the robot, we have to bind the command. This is not done binding a button, but setting the default command in the drivetrain subsystem.



```java
@Override
public void initDefaultCommand() {
  setDefaultCommand(new ArcadeDriveWithJoystick());
}
```

Deploy your robot, and you will be able to drive and control the arms.

For our robot, it is moving backwards! Rotation works fine, but this is based on how the motor controllers are wired. You can either reverse them to ensure the positive is move forward, or just negate the X axis:

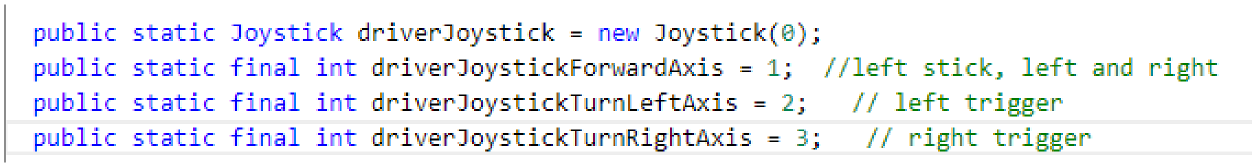


```java
public void arcadeDrive(double xSpeed,double zRotation) {
  drivetrain.arcadeDrive( xSpeed * -1, zRotation);
}
```

This would be enough to get our robot to do everything in teleop (before we climb).

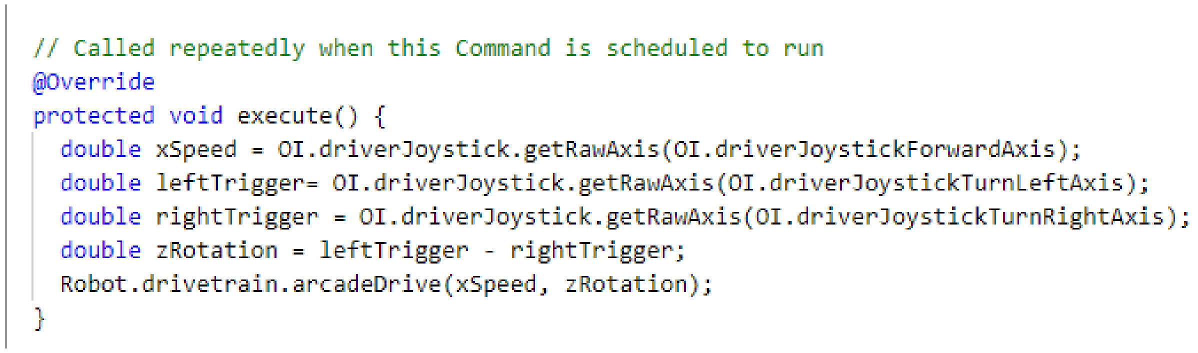
Lab 10 – Making the robot turn with the triggers

One of the common things we do is make the robot turn with triggers, rather than left/right. To do this, we will need to identify the axis we need to check in our OI file. We can get rid of the Turn axis, and end up with the following variables:



```
public static Joystick driverJoystick = new Joystick(0);
public static final int driverJoystickForwardAxis = 1;  //left stick, left and right
public static final int driverJoystickTurnLeftAxis = 2;   // left trigger
public static final int driverJoystickTurnRightAxis = 3;   // right trigger
```

Now we have to calculate the rotation speed by subtracting the axis from each other, and send the result:



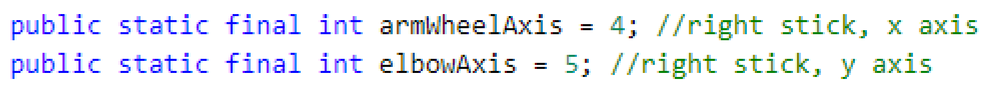
```
// Called repeatedly when this Command is scheduled to run
@Override
protected void execute() {
  double xSpeed = OI.driverJoystick.getRawAxis(OI.driverJoystickForwardAxis);
  double leftTrigger= OI.driverJoystick.getRawAxis(OI.driverJoystickTurnLeftAxis);
  double rightTrigger = OI.driverJoystick.getRawAxis(OI.driverJoystickTurnRightAxis);
  double zRotation = leftTrigger - rightTrigger;
  Robot.drivetrain.arcadeDrive(xSpeed, zRotation);
}
```

Build and test!

Lab 11 – Making the robot arms and wheels move with an axis

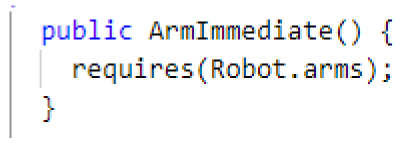
We want to move the arm wheel motors and lift with a d-pad. You cannot bind the command to the D-Pad, so we would have to make a command to control both, and make it the default command of the arms subsystem, just like driving.

We will use the bottom right D-Pad. The Y axis is axis 5, and the X Axis is axis 4.



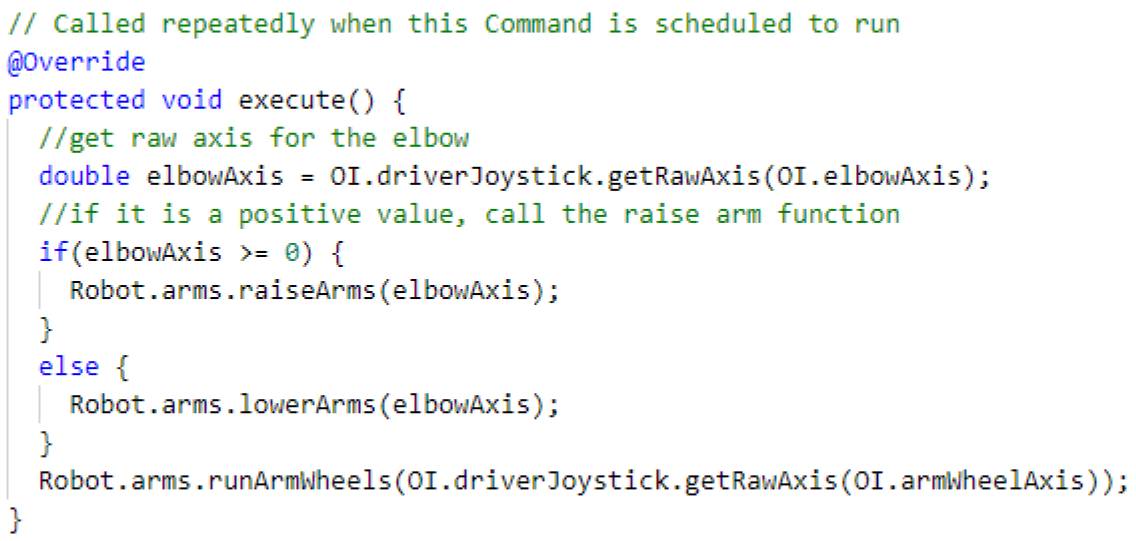
```java
public static final int armWheelAxis = 4; //right stick, x axis
public static final int elbowAxis = 5; //right stick, y axis
```

Then we create a new command, ArmImmediate. For the constructor, we will take over the arm subsystem.



```java
public ArmImmediate() {
  requires(Robot.arms);
}
```

For the execute function, we need to check the axis and execute the functions as needed.

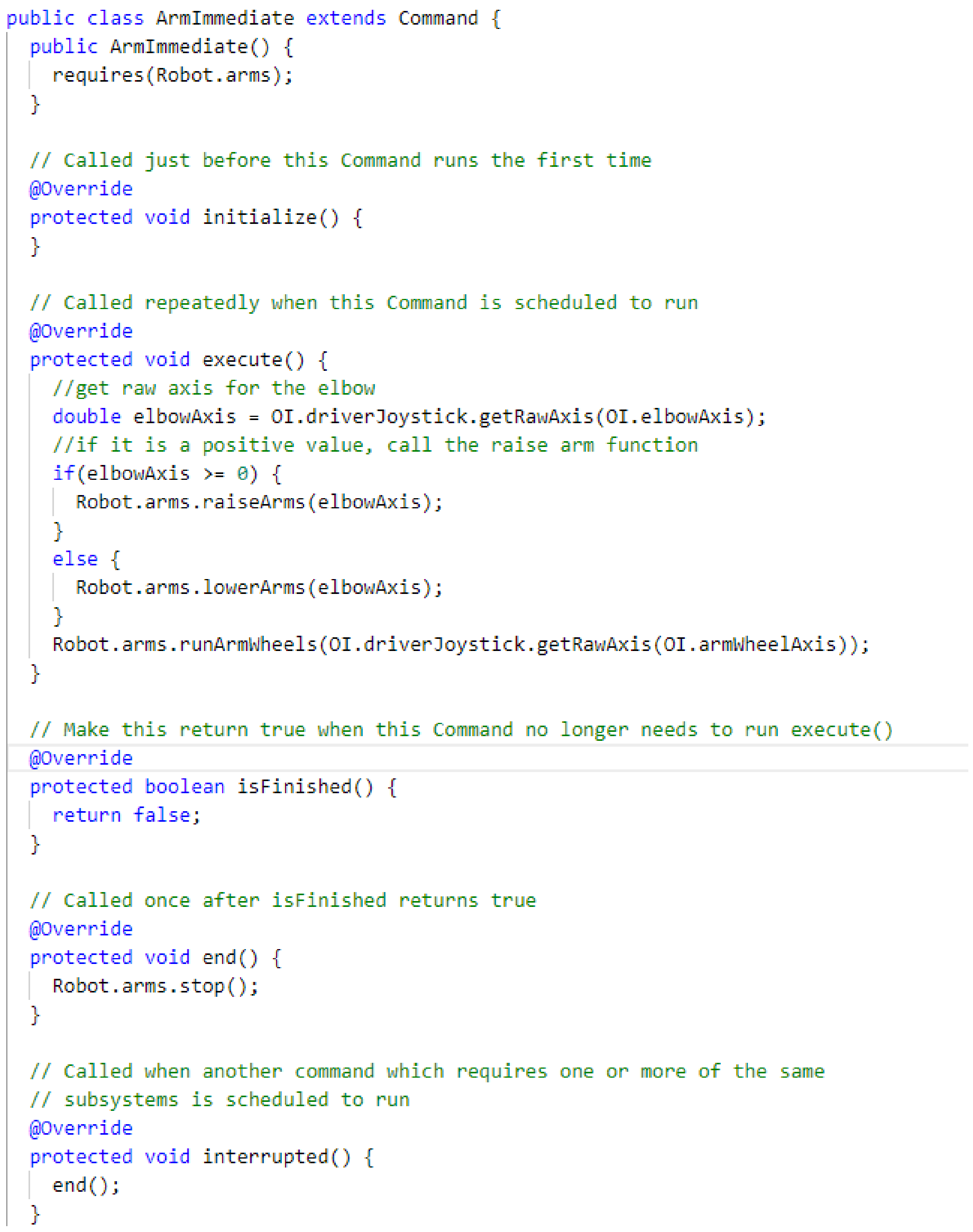


```java
// Called repeatedly when this Command is scheduled to run
@Override
protected void execute() {
  //get raw axis for the elbow
  double elbowAxis = OI.driverJoystick.getRawAxis(OI.elbowAxis);
  //if it is a positive value, call the raise arm function
  if(elbowAxis >= 0) {
    Robot.arms.raiseArms(elbowAxis);
  }
  else {
    Robot.arms.lowerArms(elbowAxis);
  }
  Robot.arms.runArmWheels(OI.driverJoystick.getRawAxis(OI.armWheelAxis));
}
```

Because we are not mapping the raw value to the motor, to contain our other logic, we have to split out the functions. The raise/lower will handle ensuring the values are running the motor in a positive or negative direction.

We can map the arm wheels directly to the Robot.

On the end function, make sure you stop the motors. The final code will be:



```java
public class ArmImmediate extends Command {
  public ArmImmediate() {
    requires(Robot.arms);
  }

  // Called just before this Command runs the first time
  @Override
  protected void initialize() {
  }

  // Called repeatedly when this Command is scheduled to run
  @Override
  protected void execute() {
    //get raw axis for the elbow
    double elbowAxis = OI.driverJoystick.getRawAxis(OI.elbowAxis);
    //if it is a positive value, call the raise arm function
    if(elbowAxis >= 0) {
      Robot.arms.raiseArms(elbowAxis);
    }
    else {
      Robot.arms.lowerArms(elbowAxis);
    }
    Robot.arms.runArmWheels(OI.driverJoystick.getRawAxis(OI.armWheelAxis));
  }

  // Make this return true when this Command no longer needs to run execute()
  @Override
  protected boolean isFinished() {
    return false;
  }

  // Called once after isFinished returns true
  @Override
  protected void end() {
    Robot.arms.stop();
  }

  // Called when another command which requires one or more of the same
  // subsystems is scheduled to run
  @Override
  protected void interrupted() {
    end();
  }
```

Now we need to bind it to the arms subsystem. To do this, set our new command as the default command of the Arms subsystem.

```
@Override
public void initDefaultCommand() {
    setDefaultCommand(new ArmImmediate());
}
```

Build and test. Our test joystick had a problem that when centered, it always had a small value so the wheels always ran. We need to put in a deadband into the command to ensure the play on the joystick does not trigger the motors. We can do this in the subsystem, but we might want to send a small value to the motor and this would prevent it.

```
if((armWheelSpeed > 0 && armWheelSpeed < 0.10) || (armWheelSpeed < 0 && armWheelSpeed > -0.10)){
    armWheelSpeed = 0;
}
```

The && operator equates to the word AND, and the || operator is OR. If either one is true, set the arm wheel speed to 0, stopping the motor.

We can do the same for the elbow. The final code would be:

```
// Called repeatedly when this Command is scheduled to run
@Override
protected void execute() {
    //get raw axis for the elbow
    double elbowAxis = OI.driverJoystick.getRawAxis(OI.elbowAxis);
    if((elbowAxis > 0 && elbowAxis < 0.10) || (elbowAxis < 0 && elbowAxis > -0.10)){
        elbowAxis = 0;
    }
    //if it is a positive value, call the raise arm function
    if(elbowAxis >= 0) {
        Robot.arms.raiseArms(elbowAxis);
    }
    else {
        Robot.arms.lowerArms(elbowAxis);
    }
    double armWheelSpeed = OI.driverJoystick.getRawAxis(OI.armWheelAxis);
    //if arm speed is betweeen 0 and 0.10, or arm speed is between 0 and -0.10, set to 0
    if((armWheelSpeed > 0 && armWheelSpeed < 0.10) || (armWheelSpeed < 0 && armWheelSpeed > -0.10)){
        armWheelSpeed = 0;
    }
    Robot.arms.runArmWheels(armWheelSpeed);
}
```