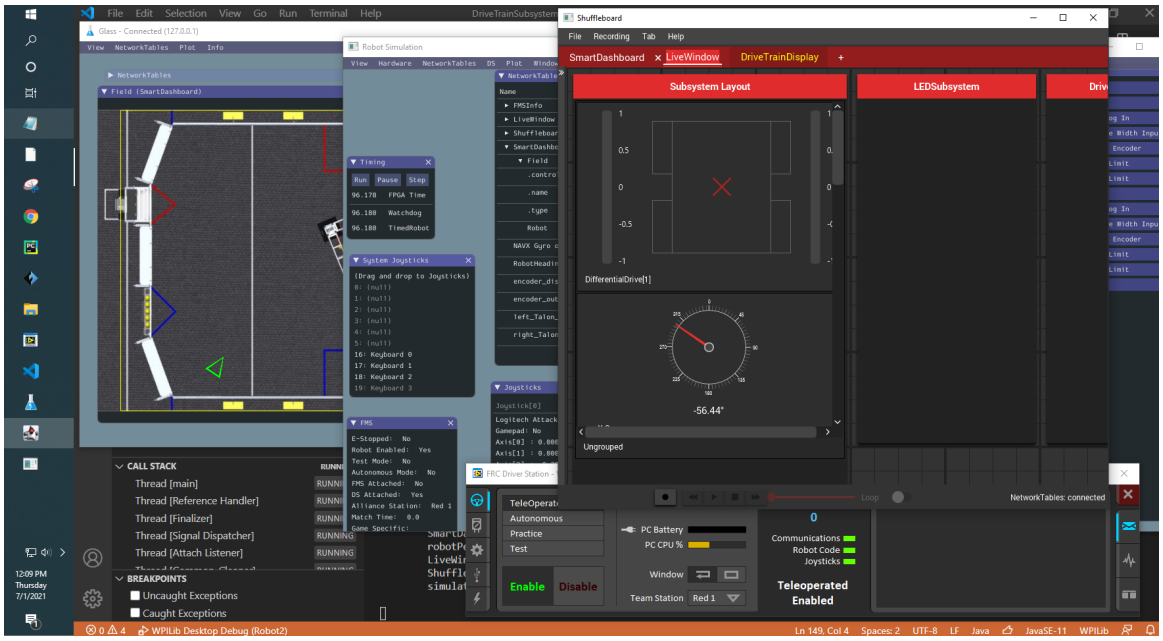
FRC Process to Write JAVA code for a Command Based Robot (July 1, 2021)

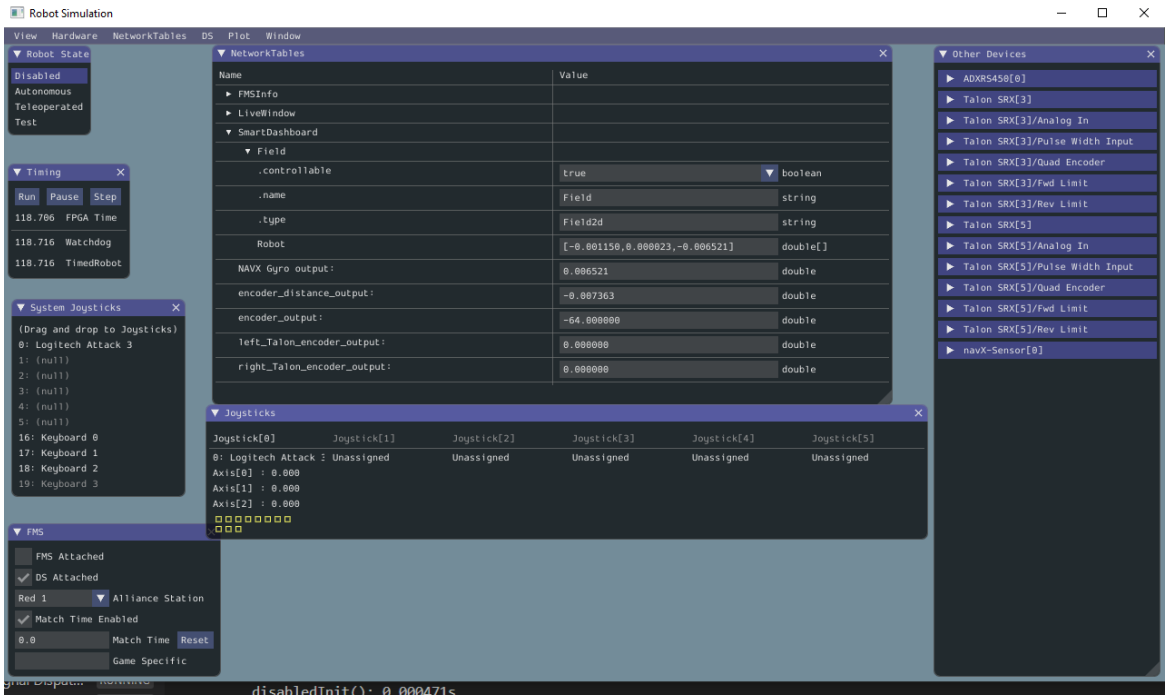
Purpose:

This document walks a new robotics team student thru the process to write JAVA code for a (FRC) FIRST Robotics Competition robot using the WPI 2021 library in VSCode

This document also includes the steps required to run the WPI built-in two-dimensional simulation including the simulation of CTRE (Cross the Road Electronics) Talon SRX and NAVX Gyro.

The WPI built-in integration provides a way to checkout the basic logic of your robot code and presents a graphic of your robot driving on the field. The simulation supports the actual driver station application and Shuffleboard debugging interface. To achieve realistic physical modeling of your robot, a number of tuning parameters must be configured. This procedure is not incorporated in this document.





FRC Process to Write JAVA code for a Command Based Robot (July 1, 2021)

Goal:

The target audience for this document is new Robotic Team students who have little programming experience. At the end of this document, you should have a robot that drives under control of a joystick and can autonomously drive in a square. An LED Strip is incorporated into the project.

The goal of this document is to build code and understand the purpose of most of the code. The full code is provided at the end of the document for support but recommend you walk through the instructions and assemble the code one function at a time.

If you do not have a physical robot available, the simulation capability can be incorporated and used for initial checkout. This document provides an intentionally minimized implementation of the simulation to assist robotics teams in understanding how the simulation operates.

This document has almost 300 steps to demonstrate a broad set of functionalities. You should cut and paste the code blocks into your projects. You should have a working robot after 75 steps. Getting the simulation working will take 127 steps.

Require Resources:

To follow this document, you need the following items:

1) A computer with VSCode enhanced with the FRC 2021 code and the drivers station installed.
2) The laptop must have access to the internet (to download the third-party libraries)
3) The laptop should have the standardized path of **c:_robotics\2021-2022** to hold the new code or what the current path is.
4) A Gamepad game controller
5) [Optional using simulation] A Team 1895 practice robot. This is a simple two motor, drop center 6-wheel drive robot with the following configuration:
    a. The left motor is at CAN Talon address of 5
    b. The right motor CAN Talon address of 3
    c. An analog range finder (Maxbotix MaxSonar type) on port 0
    d. LED Strip on PWM port 0

Status:

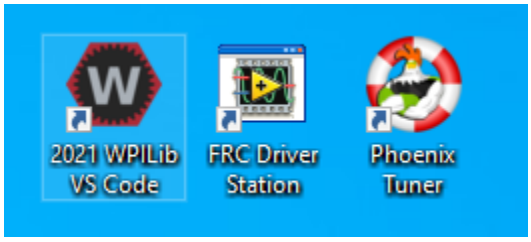
This document is nearly complete. The instructions were tested on a RoboRIO, two talons and a single motor. The instructions need to be verified on an actual robot before they are ready for a new programming student. I'll retest once we're allowed back in the school in Fall 2021.

The WPI built-in simulation is working. Detailed tuning of the physical parameters of our robot has not been performed.

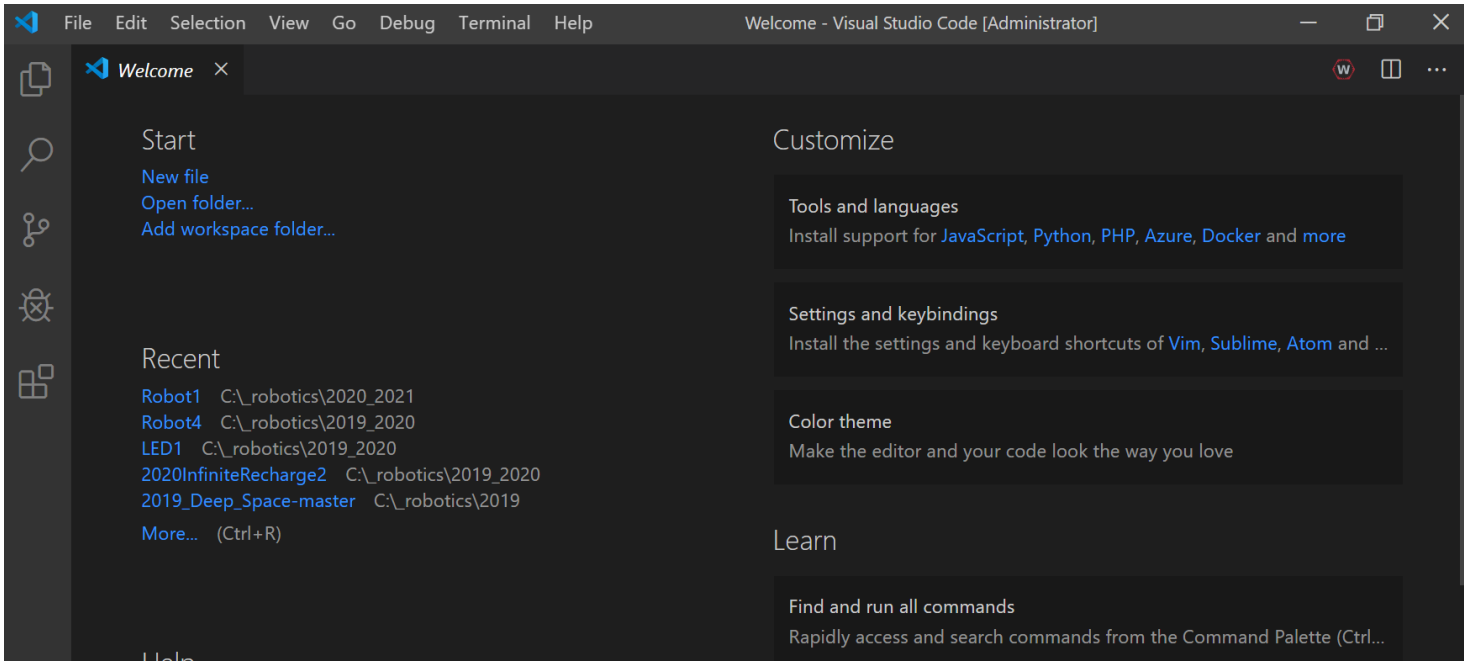
FRC Process to Write JAVA code for a Command Based Robot (July 1, 2021)

Process:

1) Logon to the laptop.
2) Double-click on the “FRC VS Code 2021” icon on the desktop.



3) Click on the WPI Icon (Red hex with a white letter W in the center). 



4) Enter “**Create a new project**” and press **enter.**
5) Click on “**Select a project type**” and select **Template**.
6) Select the language of **java**.
7) Select “**Command Robot**”.
8) Select the button “**Select a new project folder”** and browse to **c:_robotics\2021-2022** or whatever your desired path is and select **“Select Folder”** button to close the dialog box.
9) Under “**Enter a project name**” enter **Robot1** (or what every you want to call your robot such as **Johnny5**).
10) Leave the “**Create new folder? …**” checkbox **checked**.
11) Under “**Enter a team number**” enter **1895**.

12) Leave the “Enable Desktop Support” unchecked.

13) Select the **“Generate Project”** button.

14) Select “**Yes (New Window)”** in the dialog box.

15) Maximize the new window.

Install Third Party (TALON) Libraries

16) Select the WPI Icon (W) and enter “**Manage Vendor Libraries**” and press enter.
This should present 5 options.

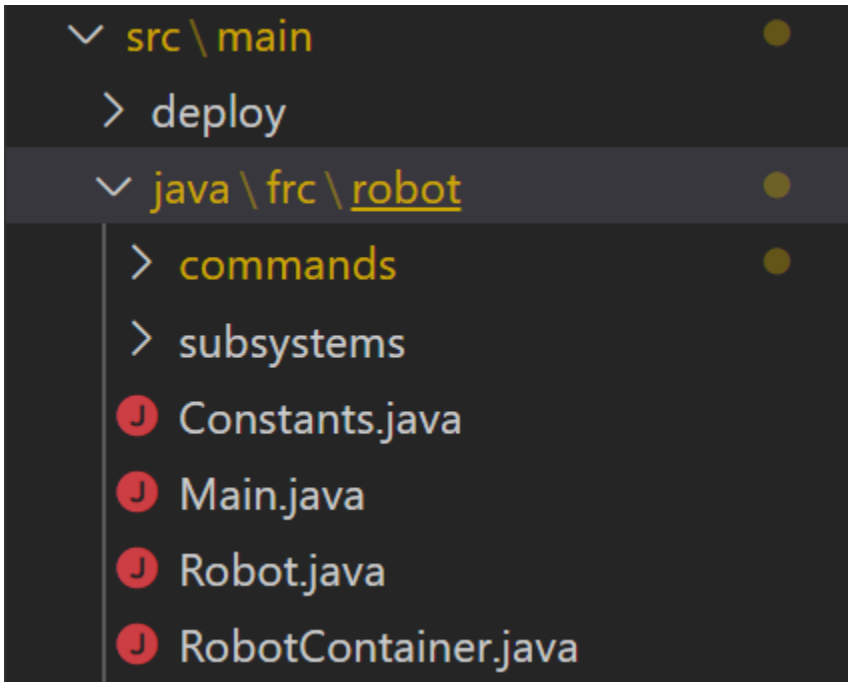
17) Select “**Install new libraries (online)**”.

18) Enter the following string (copy and paste works best) and select **enter**.
http://devsite.ctr-electronics.com/maven/release/com/ctre/phoenix/Phoenix-latest.json

19) Select **Yes** to rebuilding code.

20) In the left window, expand the “**src**” folder by selecting the arrow “>”.

21) Expand out **java** folder.



22) Observe the **command** and **subsystem** folders and four java files (Constants, Main, Robot, RobotContainer).

23) Expand out the **commands** folder and observe the **ExampleCommand** java file. (no action required)

24) Expand out the **subsystems** folder and observe the **ExampleSubsystem** java file. (no action required)

25) Highlight the **ExampleCommand** file, right click and select **delete**. Confirm the deletion (Move to recycle bin).

26) Highlight the **ExampleSubsystem** file, right click and select **delete**. Confirm the deletion (Move to recycle bin).

27) Double-click on the **Constants** java file. This file defines JAVA constant variables used to hold values that will not change.

The term **public** means it can be accessed from any other class
The term **static** means there is a single instance of this variable
The term **final** means it will not change
The term **int** means the value is a whole number.
Constant variable names are typed in all upper-case letters
The leading // means the text following it is comment and not code

Enter the following code within the "Constants" class between the code braces { }.

IT IS RECOMMENDED THAT YOU CUT AND PASTE ALL OF THE CODE IN THIS DOCUMENT.

```
// Operator Interface
public static final int DRIVER_REMOTE_PORT   = 0;
public static final int DRIVER_RIGHT_AXIS    = 4;
public static final int DRIVER_LEFT_AXIS     = 1;

// Talons
public static final int LEFT_TALON_LEADER    = 5;
public static final int RIGHT_TALON_LEADER   = 3;
```

28) The JAVA code can be reformatted to set correct indentations and code braces by selecting Shift-Alt F.

29) Select **File** => **Save** to save the changes to the current code.

Create the DriveTrain subsystem and motor controllers

30) In the left window, locate and select the folder called "**V subsystems**", right-click and select (at the bottom of the list) "**Create a new class/command**".

31) **Scroll down** in the list and select "**Subsystem (New)**" and enter the name **DriveTrainSubsystem** and press **Enter**.

32) Place your cursor near line 10 after the " */ " and select the **Enter** key three time to add space to create new code. *(The */ ends a comment block).*

33) Define variables which will become the motor controllers.

Purpose: (These line declare variables of the type WPI_TalonSRX)
(The private means the variable cannot be accessed. WPI_TalonSRX is a class within the Third-Party Library)
(The differential drive train provides the core functionality of the drive train)

At line 12 enter the following code:

```
private final WPI_TalonSRX m_rightLeader;    // Declare motor controllers variables
private final WPI_TalonSRX m_leftLeader;
private final DifferentialDrive m_safety_drive;  // Declare drive train core function
```

You will see errors indicated as red squiggly lines. The current code is missing import statements for the classes we just added. They are resolved with the next step.

34) Select **Control-Shift-P** and type **"Organize Imports"**. (You can also select Alt-Shift O)

Purpose: (This command create an import statement at the top of the file to link in the TalonSRX library)

You will now see warning indicated as yellow squiggly lines. This indicates we have not used the variables yet.

35) On line 8 (after the line starting with package frc.robot.subsystems), enter the following statement. This line makes all of the defined constants available.

```
import static frc.robot.Constants.*;
```

36) Locate the **Constructor** of the DriveTrainSubsystem. (Around line 20)
This is the line that starts with: `public DriveTrainSubsystem() {`

37) Within the class constructor, locate doubled code braces { }. Place the cursor between the braces and press enter two times to create space for new code.

38) Enter the following code on the next line after the first curly bracket within the constructor to instantiate the motor controllers.

```
m_rightLeader = new WPI_TalonSRX(RIGHT_TALON_LEADER);  // Instantiate motor controllers
m_leftLeader  = new WPI_TalonSRX(LEFT_TALON_LEADER);
```

39) We now create the Differential Drive train object within the constructor. This is the heart of the drive train functionality. Enter the following code within the constructor:

```
m_safety_drive = new DifferentialDrive(m_leftLeader, m_rightLeader);
/* Factory Default for Talons */
m_rightLeader.configFactoryDefault();
m_rightLeader.setSensorPhase(false);   //   Invert the Right Talon's quad encoder value
m_leftLeader.configFactoryDefault();
```

40) Create a **Class method** which allows Commands (which you will write later) to access to drivetrain of the robot. This statement creates a method which a Java **command** can call to drive the robot given a parameter which defines the forward speed called “move” and a parameter that defines the robot turn rate. Enter the following code BELOW the Constructor (and outside of the close bracket “}”):

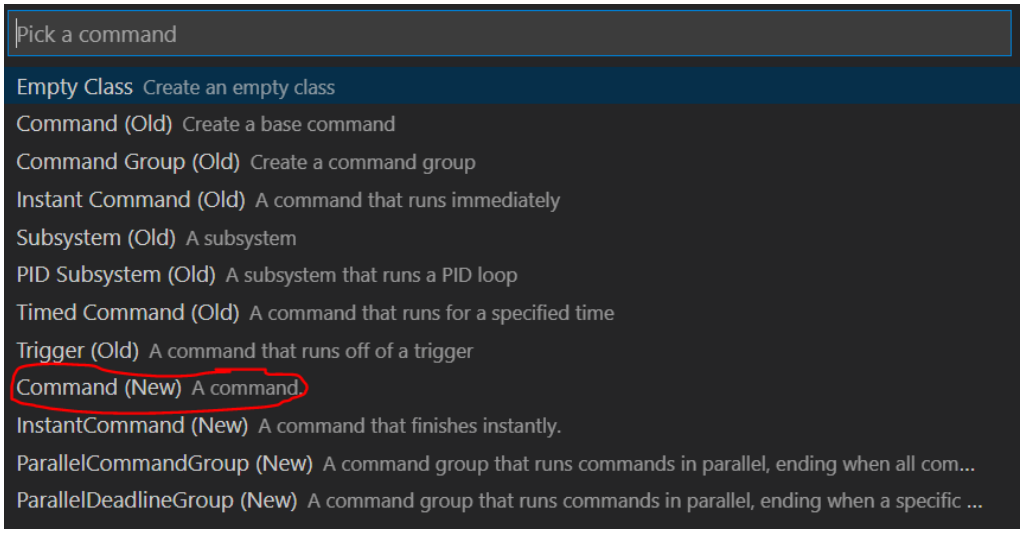
```
public void manualDrive (double move, double turn){
m_safety_drive.arcadeDrive(move, turn);
m_safety_drive.feed();
}
```

41) The JAVA code can be reformatted by selecting Shift-Alt F.

42) Select **File** => **Save** to save the changes to the current code.

Create the Drive Command

43) In the left window, locate the folder called " **V commands**" and select it, right-click and select (at the bottom of the list) "**Create a new class/command**".



44) Scroll down and select **Command (new)** and enter the command name of **DriveManuallyCommand**.

45) Double-click on the new **DriveManuallyCommand** and update the constructor to accept two parameters. This is the text "`DriveTrainSubsystem driveTrain, XboxController driverController`" between the parenthesis. These are the drivetrain object and the gamepad object. Locate the **constructor** by identifying a method with the same name as the command but does not have a return value of void. The resulting line (near line 11) should look like the following:
(You can ignore any errors at this time. We will resolve them later)

```
public DriveManuallyCommand(DriveTrainSubsystem driveTrain, XboxController driverController) {
```

46) Within the constructor add the following two lines to assign the drivetrain and XboxController objects passed in as parameters to the objects used within the class: (Near line 12)

```
m_driveTrain = driveTrain;
m_driverController = driverController;
```

47) Identify which **subsystem** this command will be using. This information is placed within the class constructor. Add the following command at the bottom of the class constructor (Around line 15):

```
addRequirements(m_driveTrain);
```

48) Define the DriveTrain and XboxController object variables within this command by inserting the following two lines of code at the top of the class. (Just before the constructor, about line 10):

```
// Reference to the constructed drive train from RobotContainer to be
// used to drive our robot
private final DriveTrainSubsystem m_driveTrain;
private final XboxController m_driverController;
```

49) Scroll down within the code and locate the execute method. The **execute()** method run 50 times per second. This is the code where we will pass the joystick positions to the drive train. The upper-case variables are constants defined in the Constants.java file. Add the following code to the execute method between the two curly brackets { } :

```
// Axis are inverted, negate them so positive is forward
    double turn  =  m_driverController.getRawAxis(DRIVER_RIGHT_AXIS); // Right X
    double move  = -m_driverController.getRawAxis(DRIVER_LEFT_AXIS); // Left Y

    m_driveTrain.manualDrive(move, turn);
```

50) Need to update the import files again. Select Control-Shift-P and type **"Organize Imports"**.

51) Manually add the following import statement to pick-up the Constants near the top of the class below the existing import statements. (Not sure why this is not done automatically):

```
import static frc.robot.Constants.*;
```

52) The JAVA code can be reformatted by selecting Shift-Alt F.

53) Save your work. Select **File** => **Save All.**

Update the RobotContainer file

54) In the left window, locate and double-click to open the **RobotContainer.java** file.

This is a very important file within the project.

55) Manually add the following import statement to pick-up the Constants near the top of the class below the existing import statements. (Not sure why this is not done automatically):

```
import static frc.robot.Constants.*;
```

56) Locate the two lines which **instantiate** the **ExampleSubsystem** and **ExampleCommand**. They likely have red lines under them. They start with "private" and include "= new " in the middle. (Near line 22).
(No action required.)

57) Create code to instantiate the **XboxController**. Enter the following code below the "ExampleCommand" line of code:

```
private final XboxController m_driver_controller = new XboxController(DRIVER_REMOTE_PORT);
```

58) Create a new line of code to instantiate the new subsystem called **DriveTrainSubsystem**. Enter the following line right below the "XboxController""

```
private final DriveTrainSubsystem m_driveTrainSubsystem = new DriveTrainSubsystem();
```

59) Create a new line of code to instantiate the new command called **DriveManuallyCommand**.
Notice that driveTrain and XboxController object parameters are passed into commands.
Enter the following line right below the "`DriveTrainSubsystem`"

```
private final DriveManuallyCommand m_DriveManuallyCommand = new DriveManuallyCommand(m_driveTrainSubsystem, m_driver_controller);
```

60) Locate the Constructor of the RobotContainer.
This is the line that contains: `public RobotContainer() {`

61) After the "`configureButtonBindings`();", enter the following command to set the **"`DriveManuallyCommand`"** as the **default command**. This means the **"`DriveManuallyCommand`"** command will run when no other command has been called (scheduled).

```
m_driveTrainSubsystem.setDefaultCommand(new DriveManuallyCommand(m_driveTrainSubsystem, m_driver_controller));
```

62) Locate and delete the two lines which **instantiate** the **ExampleSubsystem** and **ExampleCommand**. They likely have red lines under them. (Near lines 22-24)

63) Select Control-Shift-P and type **"Organize Imports"**. (You can also select Alt-Shift O)

64) Locate the method called getAutonomousCommand() near line 60. *This statement tells the drivetrain subsystem to run the* `m_DriveManuallyCommand` *command in autonomous mode.*
Update the return statement to the following code:

```
return m_DriveManuallyCommand;
```

First Build of Code and Robot Test

NOTE: The workstation must be connected to the internet for the first build to allow Gradle to be downloaded

65) The JAVA code can be reformatted by selecting Shift-Alt F.

66) Select **File** => **Save** to save the Java Project.

67) Place the robot on blocks so the wheels are not touching the ground or desktop.

68) Power on the Robot.

69) Connect the Workstation to the Robot using either a USB cable between the workstation and the RoboRIO or with the wireless WIFI connection.

70) Connect a Gamepad to the workstation.

71) Select the WPI Icon (W). Enter "**Deploy Robot Code**" and select **Enter**.

72) Start the **FRC Drivers Station** application on the desktop of the workstation.

73) Select **TeleOperated** Mode.

74) Select **Enable** on the Drivers station to start the Robot.

(THINK SAFETY: Be sure everyone is clear of the Robot)

75) Test the code by moving the joysticks. (Left joystick is forward and reverse, right joystick is rotate left and right)

Adding Robot Sensors:

To perform tasks autonomously, robots need sensors to determine how far they have moved or how much they have turned. In this next section, we will add shaft encoders and a gyro to the robot code. This document will describe how to add code for shaft encoders connected directly to the RoboRio DIO (Data Input/Outputs) and connected to the CTRE Talon SRX. This document will also describe how to add code for a common small board gyro (ADXRS450) and more capable NAV-X board gyro. Select the code your robot needs and delete or comment out the other code.

Adding Shaft Encoders to the Robot Code

[Encoders connected directly to the RoboRIO DIOs]

76) Locate and open the **Constants.java** file.

77) Add the following code at the end of the Constants file before the last curly bracket.

This code provides values used to identify which ports are used on the RoboRIO and if the encoder count direction should be flipped.

```
public static final class DriveConstants {
    // Shaft Encoders attached to the RoboRIO
    public static final int[] kLeftEncoderPorts = new int[] {0, 1}; // DIO ports
    public static final int[] kRightEncoderPorts = new int[] {2, 3};
    public static final boolean kLeftEncoderReversed = false;
    public static final boolean kRightEncoderReversed = true;

    public static final int kEncoderCPR = 1024;
    public static final double kWheelDiameterMeters = 0.15;
    public static final double kEncoderDistancePerPulse =
        // Assumes the encoders are directly mounted on the wheel shafts
        (kWheelDiameterMeters * Math.PI) / (double) kEncoderCPR;
}
```

78) Locate and open the **DriveTrainSubSystem.java** file.

79) Add the following code near line 17, after we have instantiated the m_safety_drive and before the DriveTrainSubsystem constructor.

```
// The left-side drive encoder (connected to the RoboRIO DIOs)
private final Encoder m_leftEncoder = new Encoder(DriveConstants.kLeftEncoderPorts[0],
    DriveConstants.kLeftEncoderPorts[1], DriveConstants.kLeftEncoderReversed);

// The right-side drive encoder (connected to the RoboRIO DIOs)
private final Encoder m_rightEncoder = new Encoder(DriveConstants.kRightEncoderPorts[0],
    DriveConstants.kRightEncoderPorts[1], DriveConstants.kRightEncoderReversed);
```

80) Within the DriveTrainSubSystem **Constructor**, add the following code just before the end of the constructor. This code is run one time when the code first starts.

```
// ================================================================
// Configure the external encoders
// Sets the distance per pulse for the encoders
m_leftEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);
m_rightEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);

// Reset the external encoders
resetEncoders();
```

81) Add the following code after constructor. This is a method that resets the RoboRIO connected encoders.

```
/** Resets the Talon drive encoders to currently read a position of 0. */
public void resetEncoders() {
  m_leftEncoder.reset();
  m_rightEncoder.reset();
}
```

82) Select Control-Shift-P and type **"Organize Imports"**. (You can also select Alt-Shift O)
Select "edu.wpi.first.wpilibj.Encoder" if prompted.

83) The JAVA code can be reformatted by selecting Shift-Alt F.

84) Select **File** => **Save** to save the Java Project.

[Encoders connected to the Talon SRX]

85) Add the following code after constructor. This creates methods to read the current value of the encoders connected to the Talon and a method to reset the encoders to a zero value.

```
// Talan Motor Control - Encoders
public double getLeftEncoderValue() {
  return m_leftLeader.getSelectedSensorPosition();
}

public double getRightEncoderValue() {
  return m_rightLeader.getSelectedSensorPosition();
}

public void reset_drivetrain_encoders() {
  m_leftLeader.setSelectedSensorPosition(0, 0, 0);
  m_rightLeader.setSelectedSensorPosition(0, 0, 0);
}
```

86) Select Control-Shift-P and type **"Organize Imports"**. (You can also select Alt-Shift O)
Select "edu.wpi.first.wpilibj.Encoder" if prompted.

87) The JAVA code can be reformatted by selecting Shift-Alt F.

88) Select **File** => **Save** to save the Java Project.

Adding a Gyro to the Robot Code

In this section we will add code to support two different types of gyros; the ADXRS450_Gyro and the NAV-X gyro.

[Stand Alone Gyro - ADXRS450]

89) Within the **Contants.java** file, add the following line of code at the end of the **DriveConstants** class (Before the two curly brackets at the end of the file).

```
public static final boolean kGyroReversed = true;
```

90) Locate and open the **DriveTrainSubSystem.java** file.

91) Within the **DriveTrainSubSystem.java** file, add the following code near line 33, after we have instantiated the encoders and before the DriveTrainSubsystem constructor. This code instantiates the stand-alone gyro.

```
// Add the stand-alone gyro sensor
  private final ADXRS450_Gyro m_gyro = new ADXRS450_Gyro();
```

92) Add the following code after **DriveTrainSubSystem** constructor. This creates methods to read the current value of the stand-alone gryo.

```
/**
 * Returns the heading of the robot.
 *
 * @return the robot's heading in degrees, from -180 to 180
 */
public double getHeading() {
  return Math.IEEEremainder(m_gyro.getAngle(), 360) * (DriveConstants.kGyroReversed ? -1.0 : 1.0);
}
```

93) Select Control-Shift-P and type **"Organize Imports"**. (You can also select Alt-Shift O)
Select "edu.wpi.first.wpilibj.Encoder" if prompted.

94) The JAVA code can be reformatted by selecting Shift-Alt F.

95) Select **File** => **Save** to save the Java Project.

[NAVX Gyro]

This section describes how add the Kauai Labs NAVX Gyro. This is a solid-state gyro mounted on the RoboRIO. A third-party library must be added to the project to support the Kauai Labs NAVX Gyro.

96) Select the WPI Icon (W) and enter "**Manage Vendor Libraries**" and press enter.

This should present 5 options.

97) Select "**Install new libraries (online)**".

98) Enter the following string (copy and paste works best) and select **enter**.

https://www.kauailabs.com/dist/frc/2020/navx_frc.json

99) Select **Yes** to rebuilding code.

100) At the top of the **DriveTrainSubSystem.java** file add the following import statement:

```
import com.kauailabs.navx.frc.AHRS;
```

101) Declare variable for AHRS Navx at the top of the class before the constructor.

```
private final AHRS ahrs;
```

102) Instantiate at the bottom of the DriveTrain Constructor.

```
ahrs = new AHRS(SPI.Port.kMXP);
```

103) At the bottom of the DriveTrain class but before the **Periodic()** method, add a method to reset the gyro.

```
public void reset_gyro(){
  ahrs.reset();
       }
```

104) At the bottom of the DriveTrain class, add a method to read the gyro.

```
public double get_current_heading(){
  return ahrs.getAngle();
}
```

105) Support testing of the Gyro Read method by adding code to the **manualDrive** method (near line 45) which displays the current gyro value onto the Shuffleboard:

```
// Test the Gyro by displaying the current value on the shuffleboard
 double currentHeading = get_current_heading();
 int currentHeadingInteger = (int) (currentHeading);
 SmartDashboard.putNumber("RobotHeading", currentHeadingInteger);
```

106) Select Control-Shift-P and type **"Organize Imports"**.

107) The JAVA code can be reformatted by selecting Shift-Alt F.

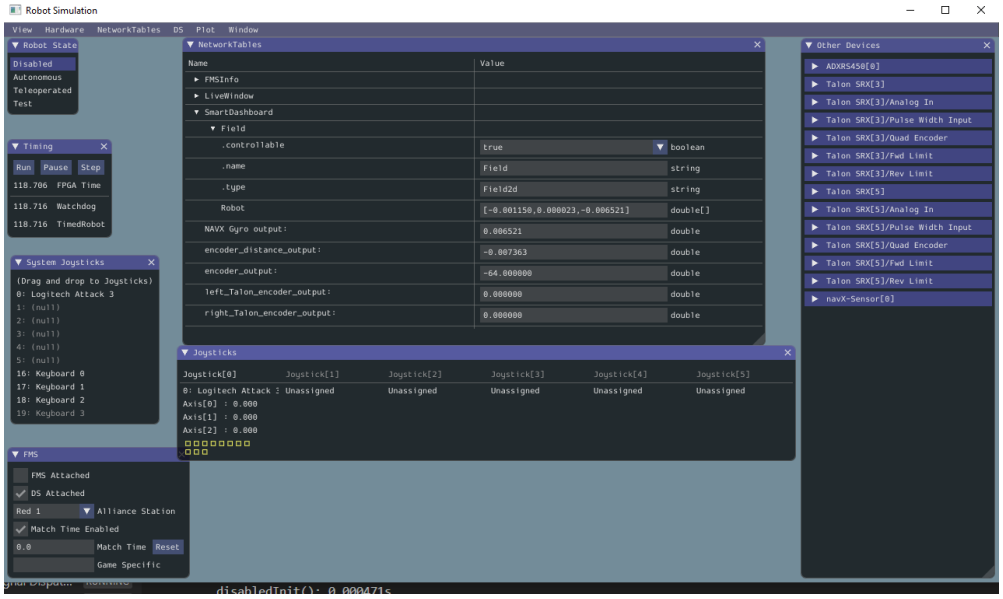
108) Select **File** => **Save** to save the Java Project.

Enabling the WPI Built-in two-dimensional Simulation

The WPI library simulation is described within the WPI documentation at URL:
https://docs.wpilib.org/en/stable/docs/software/wpilib-tools/robot-simulation/introduction.html

The simulation models the physical and logical aspects of your robot. Review the WPI documentation for details on how to characterize your robot. This document uses the provided example values. Third-party devices such as the CTRE Talon SRX and Kauai Labs NAVX Gyro are simulated with added Java libraries.

The simulation brings up a text-based GUI displaying information about your robot.

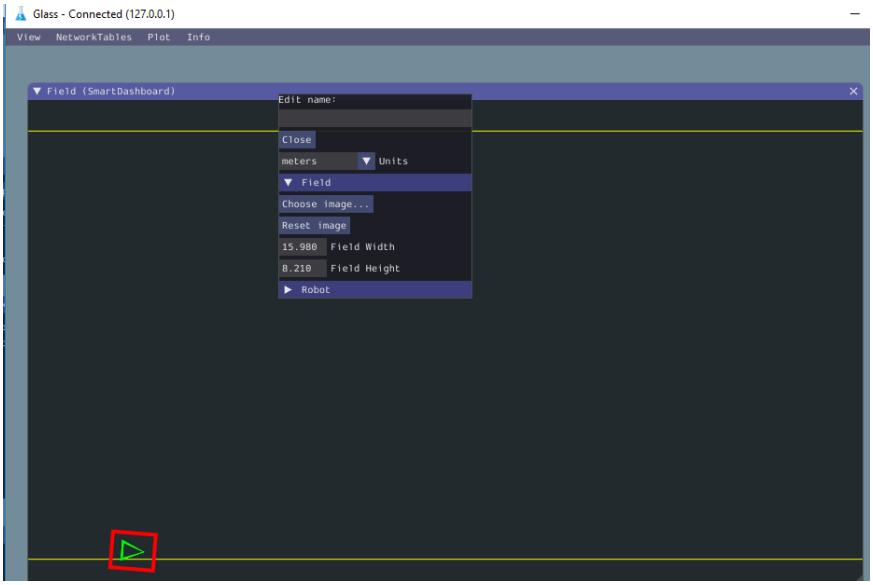


The simulation has an abbreviated driver station to select the mode (disabled, autonomous or teleop). If the FRC Driver Station application is started, this small panel is not presented and the driver station is used to control the robot. When the FRC Driver Station is started, the selected Dashboard (Shuffleboard, Default, …) is also started.

FRC Process to Write JAVA code for a Command Based Robot (July 1, 2021)

The simulation uses a WPI tool called “Glass” to display the robot position on the playing field. The tool “Glass” needs to be configured to listen to the networktables on the localhost. Use the following steps to set the mode and address:

- Start Glass (Select (**W**) then enter “Start Tool”, then select “Glass”
- Select the menu option “NetworkTables Settings”. Set to “Client” mode with an address of 127.0.0.1.
- To display the field, select menu option “NetworkTables” >> SmartDashboard >> Click on “Field”.



Glass can be configured to display a specific playing field by overlaying an image. The WPI documentation is at:
https://docs.wpilib.org/en/stable/docs/software/wpilib-tools/glass/field2d-widget.html

Download the field images from the WPI GitHub Dreamweaver page.
https://github.com/wpilibsuite/PathWeaver/tree/main/src/main/resources/edu/wpi/first/pathweaver
https://github.com/wpilibsuite/PathWeaver/blob/main/src/main/resources/edu/wpi/first/pathweaver/2021-field.png

Perform the following steps within Glass to add the image:
To add an image of the field, Right click on the top edge of the field, select “field”, Select “Choose Image” and browse to an image file.

You can check out the simulation using CTRE’s sample robot code called SixTalonArcadeDrive.
Download Example Program from CTRE with simulated Talon SRX (SixTalonArcadeDrive)
Bundle of examples at: https://github.com/CrossTheRoadElec/Phoenix-Examples-Languages

- Extract: SixTalonArcadeDrive
- Place in a folder for your robot code. (e.g. c:_robotics\2021-2022)
- Update the project to the 2021 code base
- Start the simulation

NOTE: It appears the workstation must be connected to the internet to operate.
Further investigation required.

Adding the simulation to the code for this walkthrough:

This section describes how to enable the WPI library simulation AND incorporate the simulation of the CTRE Talon SRX and NAVX Gyro. Four files are modified to enable the simulation: PhysicsSim.java, TalonSRXSimProfile.java, Constants.java, DriveTrainSubsystem.java

[Adding the CTRE PhysicsSim.java]

109) In the left "Explorer" window of VSCode, click on the word "**robot**" within folder named "java\frc**robot**", right-click and select "**New Folder**", type in "**sim**" and select **Enter**.

110) Click on the new "**sim**" folder and create a new class by right-clicking and selecting "Create New Class/Command" at the bottom of the list. Select "Empty Class" and type in exactly "**PhysicsSim**" for the class name.

111) Copy the following code and paste into the **PhysicsSim.java** file replacing the code that was there:

```
//====================================================================
// D.Frederick:
// This file is provided by CTRE to support the simulation of the Talon SRX
// Modified to remove the code for the Victor
//====================================================================
package frc.robot.sim;

import java.util.*;
import com.ctre.phoenix.motorcontrol.can.*;

/**
 * Manages physics simulation for CTRE products.
 */
public class PhysicsSim {
    private static final PhysicsSim sim = new PhysicsSim();

    /**
     * Gets the robot simulator instance.
     */
    public static PhysicsSim getInstance() {
        return sim;
    }

    /**
     * Adds a TalonSRX controller to the simulator.
     *
     * @param talon           The TalonSRX device
     * @param accelToFullTime The time the motor takes to accelerate from 0 to full,
     *                        in seconds
     * @param fullVel         The maximum motor velocity, in ticks per 100ms
     */
    public void addTalonSRX(TalonSRX talon, final double accelToFullTime, final double fullVel) {
        addTalonSRX(talon, accelToFullTime, fullVel, false);
    }

    /**
     * Adds a TalonSRX controller to the simulator.
     *
     * @param talon           The TalonSRX device
     * @param accelToFullTime The time the motor takes to accelerate from 0 to full,
     *                        in seconds
     * @param fullVel         The maximum motor velocity, in ticks per 100ms
     * @param sensorPhase     The phase of the TalonSRX sensors
     */
    public void addTalonSRX(TalonSRX talon, final double accelToFullTime, final double fullVel,
            final boolean sensorPhase) {
        if (talon != null) {
            TalonSRXSimProfile simTalon = new TalonSRXSimProfile(talon, accelToFullTime, fullVel, sensorPhase);
            _simProfiles.add(simTalon);
        }
    }

    /**
     * Runs the simulator: - enable the robot - simulate TalonSRX sensors
```

```
     */
    public void run() {
        // Simulate devices
        for (SimProfile simProfile : _simProfiles) {
            simProfile.run();
        }
    }

    private final ArrayList<SimProfile> _simProfiles = new ArrayList<SimProfile>();

    /*
     * scales a random domain of [0, 2pi] to [min, max] while prioritizing the peaks
     */
    static double random(double min, double max) {
        return (max - min) / 2 * Math.sin(Math.IEEEremainder(Math.random(), 2 * 3.14159)) + (max + min) / 2;
    }

    static double random(double max) {
        return random(0, max);
    }

    /**
     * Holds information about a simulated device.
     */
    static class SimProfile {
        private long _lastTime;
        private boolean _running = false;

        /**
         * Runs the simulation profile. Implemented by device-specific profiles.
         */
        public void run() {
        }

        /**
         * Returns the time since last call, in milliseconds.
         */
        protected double getPeriod() {
            // set the start time if not yet running
            if (!_running) {
                _lastTime = System.nanoTime();
                _running = true;
            }

            long now = System.nanoTime();
            final double period = (now - _lastTime) / 1000000.;
            _lastTime = now;

            return period;
        }
    }
}
```

[Adding TalonSRXSimProfile.java]

112) Click on the new "**sim**" folder and create a new class by right-clicking and selecting "Create New Class/Command" at the bottom of the list. Select "Empty Class" and type in exactly "**TalonSRXSimProfile**" for the class name.

113) Copy the following code and paste into the **TalonSRXSimProfile.java** file replacing the code that was there:

```
//============================================================
// D.Frederick:
// This file is provided by CTRE to support the simulation of the Talon SRX
//============================================================
package frc.robot.sim;

import frc.robot.sim.PhysicsSim.*;
import static frc.robot.sim.PhysicsSim.*; // random()

import com.ctre.phoenix.motorcontrol.can.*;

/**
 * Holds information about a simulated TalonSRX.
 */
```

```
class TalonSRXSimProfile extends SimProfile {
    private final TalonSRX _talon;
    private final double _accelToFullTime;
    private final double _fullVel;
    private final boolean _sensorPhase;

    /** The current position */
    private double _pos = 0;

    /** The current velocity */
    private double _vel = 0;

    /**
     * Creates a new simulation profile for a TalonSRX device.
     *
     * @param talon
     *        The TalonSRX device
     * @param accelToFullTime
     *        The time the motor takes to accelerate from 0 to full, in seconds
     * @param fullVel
     *        The maximum motor velocity, in ticks per 100ms
     * @param sensorPhase
     *        The phase of the TalonSRX sensors
     */
    public TalonSRXSimProfile(final TalonSRX talon, final double accelToFullTime, final double fullVel, final boolean sensorPhas
e) {
        this._talon = talon;
        this._accelToFullTime = accelToFullTime;
        this._fullVel = fullVel;
        this._sensorPhase = sensorPhase;
    }

    /**
     * Runs the simulation profile.
     *
     * This uses very rudimentary physics simulation and exists to allow users to test
     * features of our products in simulation using our examples out of the box.
     * Users may modify this to utilize more accurate physics simulation.
     */
    public void run() {
        final double period = getPeriod();
        final double accelAmount = _fullVel / _accelToFullTime * period / 1000;

        /// DEVICE SPEED SIMULATION

        double outPerc = _talon.getMotorOutputPercent();
        if (_sensorPhase) {
            outPerc *= -1;
        }
        // Calculate theoretical velocity with some randomness
        double theoreticalVel = outPerc * _fullVel * random(0.95, 1);
        // Simulate motor load
        if (theoreticalVel > _vel + accelAmount) {
            _vel += accelAmount;
        }
        else if (theoreticalVel < _vel - accelAmount) {
            _vel -= accelAmount;
        }
        else {
            _vel += 0.9 * (theoreticalVel - _vel);
        }
        _pos += _vel * period / 100;

        /// SET SIM PHYSICS INPUTS

        _talon.getSimCollection().addQuadraturePosition((int)(_vel * period / 100));
        _talon.getSimCollection().setQuadratureVelocity((int)_vel);

        double supplyCurrent = Math.abs(outPerc) * 30 * random(0.95, 1.05);
        double statorCurrent = outPerc == 0 ? 0 : supplyCurrent / Math.abs(outPerc);
        _talon.getSimCollection().setSupplyCurrent(supplyCurrent);
        _talon.getSimCollection().setStatorCurrent(statorCurrent);

        _talon.getSimCollection().setBusVoltage(12 - outPerc * outPerc * 3/4 * random(0.95, 1.05));
    }
}
```

114) Select **File** => **Save** to save the Java Project.

[Constants.java]

115) Within the **Contants.java** file, add the following line of code at the end of the **DriveConstants** class (Before the two curly brackets at the end of the file).

```
// Constants for Simulation
public static final double kTrackwidthMeters = 0.69;
public static final DifferentialDriveKinematics kDriveKinematics =
    new DifferentialDriveKinematics(kTrackwidthMeters);

// These are example values only - DO NOT USE THESE FOR YOUR OWN ROBOT!
// These characterization values MUST be determined either experimentally or theoretically
// for *your* robot's drive.
// The Robot Characterization Toolsuite provides a convenient tool for obtaining these
// values for your robot.
public static final double ksVolts = 0.22;
public static final double kvVoltSecondsPerMeter = 1.98;
public static final double kaVoltSecondsSquaredPerMeter = 0.2;

// These are example values only - DO NOT USE THESE FOR YOUR OWN ROBOT!
// These characterization values MUST be determined either experimentally or theoretically
// for *your* robot's drive.
// These two values are "angular" kV and kA
public static final double kvVoltSecondsPerRadian = 1.5;
public static final double kaVoltSecondsSquaredPerRadian = 0.3;

// Used for Simulation
public static final LinearSystem<N2, N2, N2> kDrivetrainPlant =
    LinearSystemId.identifyDrivetrainSystem(
        kvVoltSecondsPerMeter,
        kaVoltSecondsSquaredPerMeter,
        kvVoltSecondsPerRadian,
        kaVoltSecondsSquaredPerRadian);

// Example values only -- use what's on your physical robot!
public static final DCMotor kDriveGearbox = DCMotor.getCIM(2);
public static final double kDriveGearing = 8;
```

116) Select Control-Shift-P and type **“Organize Imports”**.

117) The JAVA code can be reformatted by selecting Shift-Alt F.

118) Select **File** => **Save** to save the Java Project.

[Updating: DriveTrainSubsystem.java]

119) Within the **DriveTrainSubsystem.java** file, add the following lines of code after instantiating the gyro and right before the constructor. (Near line 40). This code establishes variables for the simulation.

```
// ==(Added for simulation)========================================
private Field2d m_fieldSim;

// These classes help us simulate our drivetrain
public DifferentialDrivetrainSim m_drivetrainSimulator;
private EncoderSim m_leftEncoderSim;
private EncoderSim m_rightEncoderSim;
private ADXRS450_GyroSim m_gyroSim;

// Odometry class for tracking robot pose
private final DifferentialDriveOdometry m_odometry = new DifferentialDriveOdometry(
    Rotation2d.fromDegrees(getHeading()));
// ================================================================
```

120) Add the following lines of code within the DriveTrainSubsystem constructor. (Near line 74). This code instantiates simulated sensors used in the simulation.

```
// ================================================================
// Code for simulation within the DriveTrain Constructor
if (RobotBase.isSimulation()) { // If our robot is simulated
  // This class simulates our drivetrain's motion around the field.
  m_drivetrainSimulator = new DifferentialDrivetrainSim(DriveConstants.kDrivetrainPlant,
      DriveConstants.kDriveGearbox, DriveConstants.kDriveGearing, DriveConstants.kTrackwidthMeters,
      DriveConstants.kWheelDiameterMeters / 2.0, VecBuilder.fill(0, 0, 0.0001, 0.1, 0.1, 0.005, 0.005));

  // The encoder and gyro angle sims let us set simulated sensor readings
  m_leftEncoderSim = new EncoderSim(m_leftEncoder);
  m_rightEncoderSim = new EncoderSim(m_rightEncoder);
  m_gyroSim = new ADXRS450_GyroSim(m_gyro);

  // the Field2d class lets us visualize our robot in the simulation GUI.
  m_fieldSim = new Field2d();
  SmartDashboard.putData("Field", m_fieldSim);

  PhysicsSim.getInstance().addTalonSRX(m_rightLeader, 0.75, 4000);
  PhysicsSim.getInstance().addTalonSRX(m_leftLeader, 0.75, 4000);
} // end of constructor code for the simulation
```

121) Locate the periodic method within the DriveTrainSubsystem class (**public void periodic()**). (Near line 148). Replace the code starting with @Override and ending with the second from last curly bracket in the class with code following code.

This code instantiates simulated sensors used in the simulation.

```
// ==(Added for Simulation)=============================================
/**
 * Returns the currently-estimated pose of the robot.
 *
 * @return The pose.
 */
public Pose2d getPose() {
  return m_odometry.getPoseMeters();
}

@Override
public void periodic() {

  // ==(Added for Simulation)=============================================
  // Update the odometry in the periodic block
  // Read the current heading (not sure where from) and encoder values and feed to
  // Odometry model
  m_odometry.update(Rotation2d.fromDegrees(getHeading()), m_leftEncoder.getDistance(), m_rightEncoder.getDista
nce());

  // Get the pose from the drive train and send it to the simulated field.
  m_fieldSim.setRobotPose(getPose());

  // ==(Added for Testing and Troubleshooting)===================================

  int encoder_output = m_rightEncoder.getRaw();
  SmartDashboard.putNumber("encoder_output:  ", encoder_output);

  double encoder_distance_output = m_rightEncoder.getDistance();
  SmartDashboard.putNumber("encoder_distance_output:  ", encoder_distance_output);

  double left_Talon_encoder_output = getLeftEncoderValue();
  SmartDashboard.putNumber("left_Talon_encoder_output:  ", left_Talon_encoder_output);

  double right_Talon_encoder_output = getRightEncoderValue();
  SmartDashboard.putNumber("right_Talon_encoder_output:  ", right_Talon_encoder_output);
```

```
    SmartDashboard.putNumber("NAVX Gyro output:  ", ahrs.getAngle());

  }

  @Override
  public void simulationPeriodic() {
    // To update our simulation, we set motor voltage inputs, update the simulation,
    // and write the simulated positions and velocities to our simulated encoder and
    // gyro.
    // We negate the right side so that positive voltages make the right side
    // move forward.

    PhysicsSim.getInstance().run();

    m_drivetrainSimulator.setInputs(m_leftLeader.get() * RobotController.getBatteryVoltage(),
        -m_rightLeader.get() * RobotController.getBatteryVoltage());
    m_drivetrainSimulator.update(0.020);

    m_leftEncoderSim.setDistance(m_drivetrainSimulator.getLeftPositionMeters());
    m_leftEncoderSim.setRate(m_drivetrainSimulator.getLeftVelocityMetersPerSecond());
    m_rightEncoderSim.setDistance(m_drivetrainSimulator.getRightPositionMeters());
    m_rightEncoderSim.setRate(m_drivetrainSimulator.getRightVelocityMetersPerSecond());
    m_gyroSim.setAngle(-m_drivetrainSimulator.getHeading().getDegrees());

    int dev = SimDeviceDataJNI.getSimDeviceHandle("navX-Sensor[0]");
    SimDouble angle = new SimDouble(SimDeviceDataJNI.getSimValueHandle(dev, "Yaw"));
    angle.set(-m_drivetrainSimulator.getHeading().getDegrees());
  }
```

122) Select Control-Shift-P and type **“Organize Imports”**.

123) The JAVA code can be reformatted by selecting Shift-Alt F.

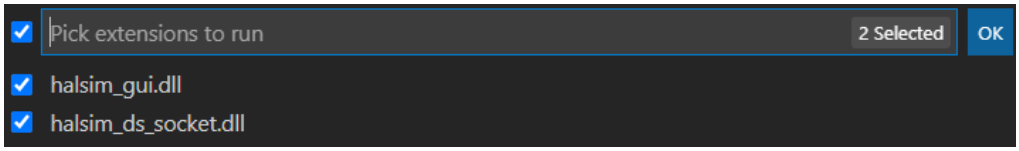
124) Select **File** => **Save** to save the Java Project.

Test out Simulation

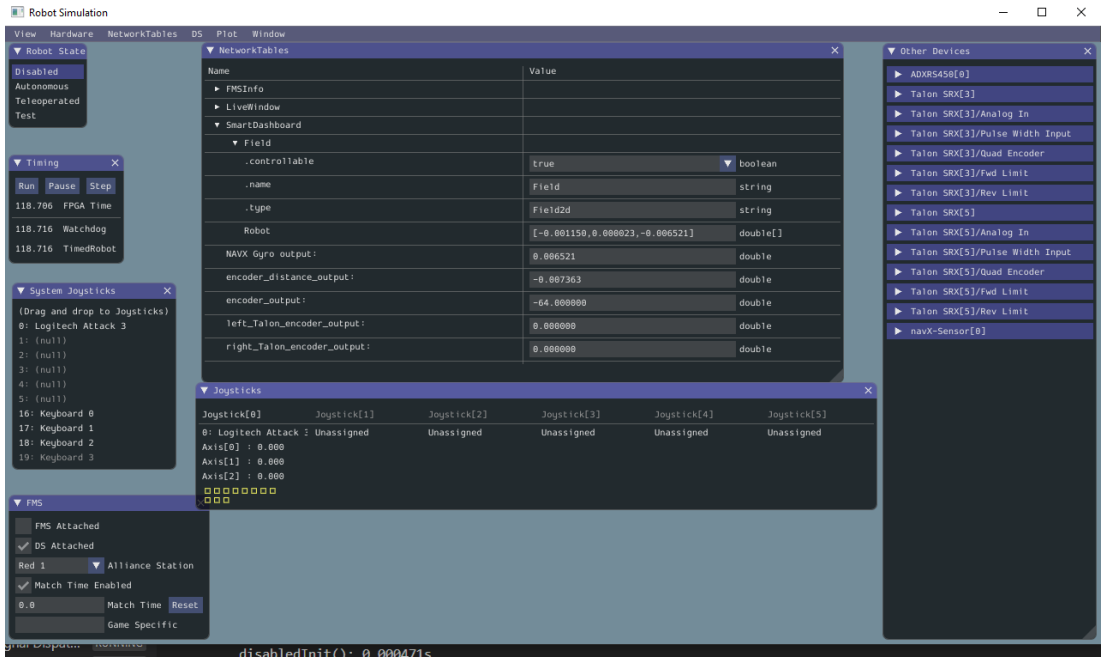
[Enable Desktop Simulation]

125) Select the WPI Icon (W) and enter “**Change Desktop Support Enabled Setting**” and press enter.
Select **Yes** to the prompt “**Enable Desktop Support for Project? Currently false**”

126) Select the WPI Icon (W) and enter “**Simulate Robot Code on Desktop**” and press enter.
The current code will compile and then present you with the following dialog box.
Select the **top** checkbox. This will place checkmarks in the bottom two entries.
Select **OK**.



If all goes well the “Robot Simulation” User Interface will appear.



127) Select the WPI Icon (W) and enter “**Start Tool**” and select the tool called “Glass”.

Expand the Robot Code to have a button-based command

The robot motors run at 50% speed when the X button is pressed.

Overview of the process to add a button-based command:
- Create Command (to run the motors at 50%)
- Link to Button (When the button is press, the command is called)

128) In the left window, locate the folder called " **V commands**" and select it, right-click and select (at the bottom of the list) "**Create a new class/command**".

129) Scroll down and select **Command (new)** and enter the command name of **DriveForward50**.

130) Update the **DriveForward50** constructor to accept one parameter. This is the drive train object. Add the following parameter within the parenthesis of the constructor:

```
public DriveManuallyCommand(DriveTrainSubsystem driveTrain) {
```

131) Within the constructor add the following line to assign the input parameter drivetrain object to the drivetrain object used within this command:

```
m_driveTrain = driveTrain;
```

132) Identify which subsystem this command will be using. This information is placed within the class constructor. Add the following command at the bottom of the class constructor (Around line 18):

```
addRequirements(m_driveTrain);
```

133) Define the DriveTrain object within this command by entering the following code at the top of the class just before the constructor:

```
// Reference to the constructed drive train from RobotContainer to be
// used to drive our robot
private final DriveTrainSubsystem m_driveTrain;
```

134) The `execute()` method runs 50 times per second. For this command, we will set the forward motion to 50% and the rotation motion to zero. Update the execute method with the code to drive the robot (Add one line of code between the open and close curly braces {} near line 29):

```
public void execute() {
  m_driveTrain.manualDrive (0.5,0);  // Drive straight forward at 50%
}
```

135) The `end()` method is run when the command completes. This is when the button is released. Update the end method with the code to **stop** the robot:
(Add one line of code between the open and close curly braces {} near line 35):

```
public void end(boolean interrupted) {
            m_driveTrain.manualDrive (0,0);
}
```

136) Need to update the import files again. Select Control-Shift-P and type **"Organize Imports"**.

137) Open up the **RobotContainer.java** file.

138) Add a joystick button object to the code. Add the following line near the top of the RobotContainer Class just below the instantiation of the **XboxController** and **DriveTrainSubsystem** objects and the **DriveManually** Command:

```
private JoystickButton driver_X;
```

139) Within the RobotContainer file, locate the "`configureButtonBindings()`" method. Add the following two lines which instantiate the button and **bind** the "**X Button**" to the command "**DriveForward50**". (Near line 46)

```
driver_X = new JoystickButton(m_driver_controller, XboxController.Button.kX.value);
driver_X.whileHeld(new DriveForward50( m_driveTrainSubsystem));
```

140) Need to update the import files again. Select Control-Shift-P and type **"Organize Imports"**.

141) The JAVA code can be reformatted by selecting Shift-Alt F.

142) Save your work. Select **File => Save All.**

143) Download the code to the Robot by selecting the WPI Icon and typing **Deploy Robot Code**.

144) Start the FRC Drivers Station application on the desktop of the workstation.

145) Select **TeleOperated** Mode.

146) Select **Enable** on the Drivers station to start the Robot.

(THINK SAFETY: Be sure everyone is clear of the Robot)

147) Test the code by pressing the **X button** on the Joystick and observing the robot drive forward and 50% speed.

Displaying Data for Troubleshooting

As the robot code get more complex, we will need methods to troubleshoot code. To troubleshoot code, we need to see what values variables are taking on. There are two options to see values. The second option is better.

A. Putting print statements in the code which writes values (for example: joystick positions, motor speed, …) to the robot driver-station console.
Implement this approach by just adding print statements in the code as follows:

```
System.out.printf ("Text to display:  %5.2f  %n", variableName);
```

B. Shuffleboard. This is a graphical display that can display values, graphs, diagrams, Booleans and much more. You can also run commands from the shuffleboard. The next section discusses how to use code to add items to the shuffleboard. (There are other manual approaches to add items to the shuffleboard, these are provided later in this document).

For the Desktop simulation, the Shuffleboard will appear when the FRC Driver Station application is started.

FRC Process to Write JAVA code for a Command Based Robot (July 1, 2021)

The following instructions will cause the code to programmatically add a new tab on the Shuffleboard called "**DriveTrainDisplay**" and add the values of the joystick axis and a button to run the **DriveForward50** command. You can use similar code to add other values.

After this section, a process is provided to describing how to manually add values to the Shuffleboard.

148) Open the RobotContainer.java file and locate the constructor. The Constructor method starts with

```
public RobotContainer() {       (No action at this time)   (Near line 36)
```

149) At the end of the **constructor**, add the following lines of code. This code adds Widgets of a specific type, location and size to Shuffleboard.

```
// Add information to the Shuffleboard for monitoring and Troubleshooting
Shuffleboard.getTab("DriveTrainDisplay");

// This code adds values to the "DriveTrainDisplay" Tab on the Shuffleboard.
// This code is dependent on two methods added to the RobotContainer to access the data to be displayed
Shuffleboard.getTab("DriveTrainDisplay").addNumber("Joystick-Y", this::getJoystickYValue)
        .withWidget(BuiltInWidgets.kNumberSlider)
        .withPosition(6,3);

Shuffleboard.getTab("DriveTrainDisplay").addNumber("Joystick-X", this::getJoystickXValue)
        .withWidget(BuiltInWidgets.kNumberSlider)
        .withPosition(6,4);
```

150) At the end of the constructor, add the following lines of code. This line adds a button to the Shuffleboard which will run a command when selected.

```
// This adds a command to the Shuffleboard
Shuffleboard.getTab("DriveTrainDisplay")
        .add(new DriveForward50(m_driveTrainSubsystem)).withPosition(6, 0).withSize(2, 2);
```

151) At the end of the RobotContainer class, add the following lines of code right before the last close brace. This code creates methods which read the joystick values for display on the shuffleboard.

```
// These are used by the commands to add data to the shuffleboard (See the RobotContainer Constructor)
    public double getJoystickXValue(){
        return m_driver_controller.getRawAxis(DRIVER_RIGHT_AXIS);
        }

    public double getJoystickYValue(){
        return m_driver_controller.getRawAxis(DRIVER_LEFT_AXIS);
        }
```

152) On line 10 (after the line starting with package frc.robot.subsystems), enter the following statement:

```
import static frc.robot.Constants.*;
```

153) The JAVA code can be reformatted by selecting Shift-Alt F.

154) Select Control-Shift-P and type **"Organize Imports"**. (You can also select Alt-Shift O)

155) Save your work. Select **File** => **Save All.**

156) Download the code to the Robot by selecting the WPI Icon and typing **Deploy Robot Code**.

157) Start the Drivers station code.

158) Select the configuration page by clicking on the “Gear” on the left side.

159) Set the “**Dashboard Type**” to “**Shuffleboard**”

160) Select **Enable** on the Drivers station to start the Robot.

161) Switch to the Shuffleboard application.

162) Select the new tab called “**DriveTrainDisplay**”. Observe the Widgets on the page.

163) Enable the Robot and move the joysticks to drive the wheels on the robot. Observe how the Widgets react.

164) Press the “DriveForward50” button and observe the robot wheels turn.

Moving the Robot Forward a Specific Distance using the Encoder connected to a Talon SRX

This section describes how to read a shaft encoder connected to the Talon SRX. Team 1895 uses Vex VersaPlanetary Integrated Encoder (AndyMark 217-5046) to monitor the rotations of our main drivetrain, climbers and shooters. Reading the encoder is useful in determining distance travelled and speed of rotation.

We will add a button-based command to move the robot forward 2 feet. To create this capability, we first create a drivetrain method which accepts a parameter defining how many inches to drive forward. Second, we create a command which calls the drivetrain method with an input parameter of 24 inches. This first implementation will use a simple approach, when the robot drives 24 inches, we will stop the motors. The robot will continue forward due to the momentum and therefore the robot will drift past the target position. The initial capability will only monitor the right-side shaft encoder. A better but more complex implementation will use a PID (Proportional, Integral, Derivative) control or Talon SRX based Motion Magic to have the robot stop exactly at the target position.

Description of the Drivetrain method:
- The command will be initiated when the “Y” button is pressed
- The command cannot be stopped once initiated.

High Level Steps to Implement:
1. Create the method in the drivetrain subsystem
2. Create the command which calls the method
3. Update the RobotContainer to bind the button to the command

This capability requires a quadrature shaft encoder to be placed on the gearbox of the robot to measure the movement of the robot. The shaft encoder provides a counter count as the shaft rotates. When this command is started, the shaft encoder count needs to be set to zero. The code drives the motors forward and monitors the shaft encoder to reach the desired count. The drivetrain subsystem method returns a Boolean value of true when the distance is reached.

165) Open the DriveTrainSubsystem.java file in VSCode.

166) Create a method near the end of the file using the following steps. This new method should be placed before the **periodic()** method The input parameters are the speed at which the robot should move, the distance in inches to move the robot and a Boolean flag to indicate to reset the encoder. Enter the following empty method code:

```
public Boolean driveForwardInches(double motorSpeed, double inchesToDrive, Boolean resetEncoder) {

  return driveComplete;
}
```

167) Within the **driveForwardInches** method, create a set of variables to use. Add the following variable declarations:

```
Boolean driveComplete = false;
double currentShaftEncoderValue = 0;
double targetShaftEncoderCount = 0;
 double convertRotationsToInches = 500;  // Will need tested, calibrated and revised
```

168) Reset the shaft encoder when indicated by the input parameter. Enter the following conditional command below the variables and before the return statement:

```
// Reset the shaft encoder value
if (resetEncoder == true) {
  reset_drivetrain_encoders();
}
```

169) Set the motor speed using arcade drive. Provide just the forward speed without the robot turning. Add this code after the code in the previous step.

```
// Set the speed of the motors using arcade drive with no turn
m_safety_drive.arcadeDrive(motorSpeed, 0);
m_safety_drive.feed();
```

170) Get the current shaft encoder value and calculate the target.
Add this code after the code in the previous step.

```
// Check the encoder
currentShaftEncoderValue = getRightEncoderValue();
targetShaftEncoderCount = inchesToDrive * convertRotationsToInches;
```

171) Compare the current shaft encoder value and target shaft encoder value.
Add this code after the code in the previous step.

```
if (currentShaftEncoderValue > targetShaftEncoderCount) {
  driveComplete = true;
}
```

172) Select Control-Shift-P and type **"Organize Imports"**. (You can also select Alt-Shift O)

173) The JAVA code can be reformatted by selecting Shift-Alt F.

174) Save your work. Select **File => Save All.**

Create the Command to Move the Robot (DriveForwardTwoFeetCommand)

175) In the left window, locate the folder called " **V commands**", right-click and select (at the bottom of the list) "**Create a new class/command**"

176) Scroll down and select **Command (new)** and enter the command name of **DriveForwardTwoFeetCommand** .

177) Update the **DriveForwardTwoFeetCommand** constructor to accept one parameter. This is the drive train object. Add the following parameters within the parenthesis of the constructor:

```
public DriveForwardTwoFeetCommand(DriveTrainSubsystem driveTrain) {
```

178) **Within the constructor** add the following code to assign the drivetrain conveyed in the input parameter to the drivetrain object used in this command:

```
m_driveTrain = driveTrain;
```

179) Identify which subsystem this command requires to operate. Add the following command at the bottom of the class constructor (Around line 15):

```
addRequirements(m_driveTrain);
```

180) Define variables used within this command. They are the DriveTrain object, a Boolean flag to indicate when the robot has moved the required distance and the drive speed. At the top of the class prior to the constructor, add the following code (near line 15):

```
  // Reference to the constructed drive train from RobotContainer to be
  // used to drive our robot
  private final DriveTrainSubsystem m_driveTrain;
  private boolean m_driveComplete;
  private double driveSpeed = 0.3;
  private double driveDistanceInches = 3 * 39.37;   // 39.37 inches per meter
```

181) Within the `initialize()` method, call the driveTrain method to start the robot moving and to reset the encoders. This method is called one time when the command is initiated. Notice the reset encoder flag is set to true.

```
m_driveComplete = m_driveTrain.driveForwardInches(driveSpeed, driveDistanceInches , true);
```

182) Within the `execute()` method (which runs 50 times per second), call the drivetrain method with defining the drive speed and distance in inches to travel. The method returns the status of completing the travel:

```
m_driveComplete = m_driveTrain.driveForwardInches(driveSpeed, driveDistanceInches , false);
```

183) Updated the `isFinished()` method to return the flag which indicates the robot has moved the required distance. This will stop the command from running.

```
public boolean isFinished() {
    return m_driveComplete;
  }
```

184) Need to update the import files again. Select Control-Shift-P and type **"Organize Imports"**.

185) Save your work. Select **File** => **Save All.**

186) Open the **RobotContainer.java** file.

187) Near the top of the file, define a variable for the “Y” button just below the definition of the “X” button.

```
private JoystickButton driver_Y;
```

188) Within the `configureButtonBindings()` method, add code to “Y” button and bind the button with the new command.

```
driver_Y = new JoystickButton(m_driver_controller, XboxController.Button.kY.value);
driver_Y.whenPressed (new DriveForwardTwoFeetCommand( m_driveTrainSubsystem));
```

189) Select Control-Shift-P and type **“Organize Imports”**.

190) The JAVA code can be reformatted by selecting Shift-Alt F.

191) Save your work. Select **File** => **Save All.**

192) The code can be tested by deploying the code to the robot, enabling the robot, and pressing the **Y** button.

Manually Displaying Data on the Shuffleboard:

While troubleshooting new code, it is beneficial to have variables displayed on the shuffleboard. One easy way to do this is to use the smartDashboard command to post data. There are two parts to achieving this goal; coding and manual configuration.

Coding Part:

193) Add the following code which declares an integer to hold the desired data near the top of the DriveTrainSubsystem class **before** the constructor. This step is only required if a new variable is required to show percentage complete. (Near line 27)

```
private int percentComplete = 0;
```

194) Add the following code to the **driveForwardInches** method, just before the return statement at the end of the method. This calculates the percent complete of reaching the target and converts it to an integer value between 0 and 100.

```
percentComplete = (int) (100 * (currentShaftEncoderValue / targetShaftEncoderCount));
SmartDashboard.putNumber("Encoder % Complete", percentComplete);
```

195) Select Control-Shift-P and type **“Organize Imports”**.

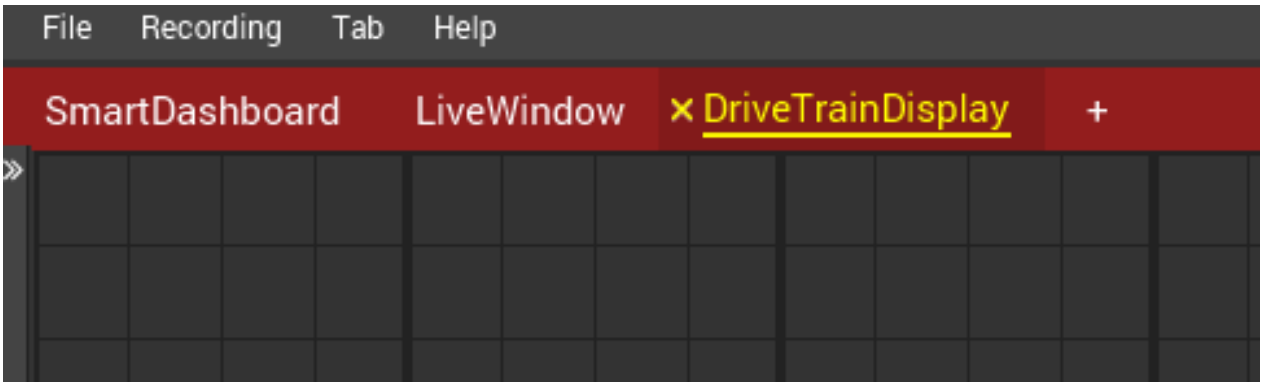
196) The JAVA code can be reformatted by selecting Shift-Alt F.

197) Save your work. Select **File** => **Save All.**

Manual Part of Setting up the Shuffleboard:

198) With the Shuffleboard open, select the “**DriveTrainDisplay**” tab.

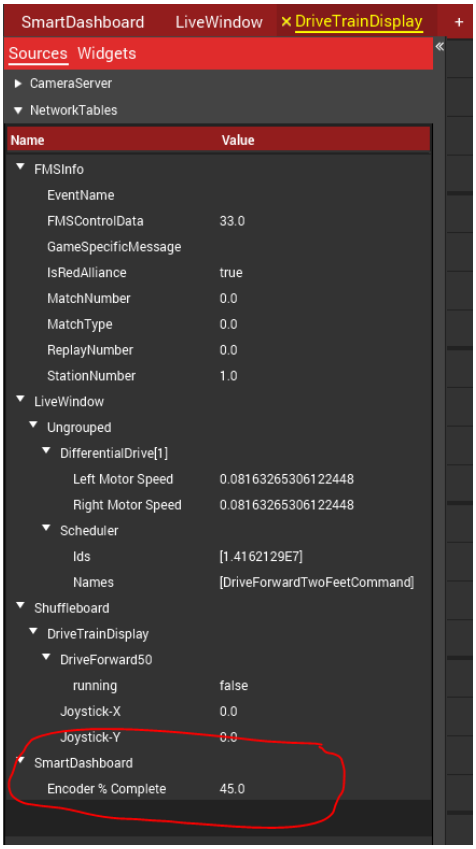
199) Select the tiny arrows in the top left corner "**>>**" under SmartDashboard. This will display all robot data available.



200) Enable the Robot.

Once the code is run one time with the "**SmartDashboard.putNumber**" code present, the "**Encoder % Complete**" will be displayed in the "**Sources / Widgets**" window. (See the location of the value in the diagram on the next page).

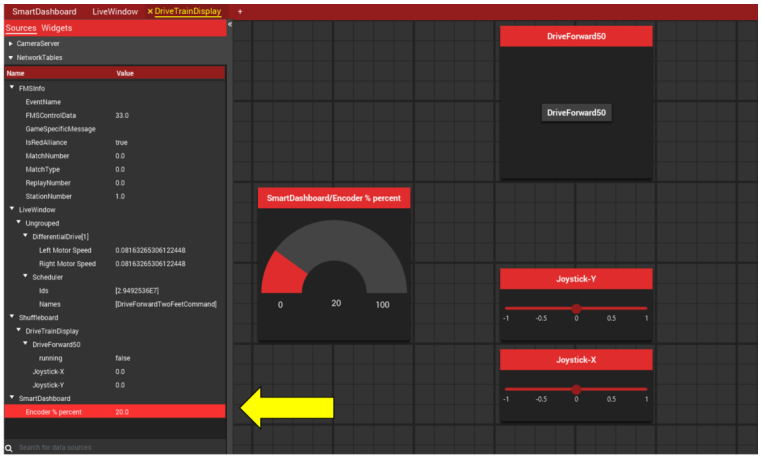
201) Select and drag the "**Encoder % Complete**" on to the Shuffleboard.



202) The format of the displayed data can be changed from just a number to a graphical Widget.

203) Right-click on the "**Encoder % Complete**" widget on the Shuffleboard and select "**Show As**" and select "**Simple Dial**"

204) Close the "**Sources / Widgets**" by clicking on the "<<" arrows.



Create a command to turn the robot

205) In the left window, locate and select the folder called " **V commands**", right-click and select (at the bottom of the list) "**Create a new class/command**"

206) Scroll down and select **Command (new)** and enter the command name of **TurnRobotToLeft90**.

207) Update the **TurnRobotToLeft90** constructor to accept one parameter. This is the drive train object. Add the following parameter within the parenthesis of the constructor:

```
public TurnRobotToLeft90(DriveTrainSubsystem driveTrain) {
```

208) Within the constructor add the following line to assign the input parameter drivetrain to the drivetrain object used in this command:

```
m_driveTrain = driveTrain;
```

209) Identify which subsystem this command will be using. This information is placed within the class constructor. Add the following command at the bottom of the class constructor (Around line 18):

```
addRequirements(m_driveTrain);
```

210) Define the DriveTrain object within this command by entering the following code at the top of the class before the constructor:

```
// Reference to the constructed drive train from RobotContainer to be
// used to drive our robot
private final DriveTrainSubsystem m_driveTrain;
private double currentGyroHeading = 0;
private double targetGyroHeading = -90;
private boolean TurnComplete = true;
```

211) Within the initialize() method, call the driveTrain method to reset the gyro. This method is called one time when the command is initiated.

```
public void initialize() {
  m_driveTrain.reset_gyro();
}
```

212) The `execute()` method runs 50 times per second. In this code, we read the gyro and compare the current gyro hearing to the target heading of 90 degrees. We set a Boolean flag to stop the command.

```
public void execute() {
       currentGyroHeading = m_driveTrain.get_current_heading();
       if (currentGyroHeading > targetGyroHeading) {
         m_driveTrain.manualDrive(0, 0.5); // Rotate at 50% power
         TurnComplete = false;
       }
       else{
         m_driveTrain.manualDrive(0, 0);
         TurnComplete = true;
       }

   }
```

213) The `end()` method is run when the command completes. This is when the button is released. Update the end method with the code to drive the robot:

```
public void end(boolean interrupted) {
            m_driveTrain.manualDrive (0,0);
}
```

214) The `isFinished()` method is run right after the execute method to determine when a command is complete. While the button is held, the robot compares the current gyro heading to a target value. Update the `isFinished()` method with the code to stop the robot:

```
public boolean isFinished() {
       return TurnComplete;
       }
```

215) Need to update the import files again. Select Control-Shift-P and type **"Organize Imports"**.

216) Open up the **RobotContainer.java** file.

217) Add a joystick button object to the code. Add the following line near the top of the RobotContainer Class just before the instantiation of the XboxController, DriveTrainSubsystem and DriveManuallyCommand command:

```
private JoystickButton driver_B;
```

218) Within the RobotContainer file, locate the "`configureButtonBindings()`" method. Add the following two lines to link the **"B Button"** to the command "**TurnRobotToLeft90**".

```
driver_B = new JoystickButton(m_driver_controller, XboxController.Button.kB.value);
driver_B.whenPressed (new TurnRobotToLeft90( m_driveTrainSubsystem));
```

219) Need to update the import files again. Select Control-Shift-P and type **"Organize Imports"**.

220) Save your work. Select **File** => **Save All.**

221) Download the code to the Robot by selecting the WPI Icon and typing **deploy Robot Code**.

222) Start the FRC Drivers Station application on the desktop of the workstation.

(THINK SAFETY: Be sure everyone is clear of the Robot)

223) Select **TeleOperated** Mode.

224) Select **Enable** on the Drivers station to start the Robot.

225) Press the Joystick B button and verify the robot turns 90 degrees to the left while the button is pressed.

226) If the robot does not turn, Use the steps in the section called "**Manually Displaying Data on the Shuffleboard**" to write the Gyro Heading value on the Shuffleboard after the code is complete.

Autonomous Mode and Command Groups

Now that we have a number of basic commands, we can chain these together to have the robot run autonomously. Let's have the robot drive in a square. The group of commands to drive the complete 4-sided square would be running the separate commands **Drive forward 2 feet** and **turn 90 degrees** repeated 4 times. The Sequential command group provides a way to run a set of commands in order.

227) In the left window, locate and select the folder called " **V commands**", right-click and select (at the bottom of the list) "**Create a new class/command**".

228) **Scroll down** and select **SequentialCommandGroup (new)** and enter the command name of **DriveInSquareCommandGroup**.

229) Update the **DriveInSquareCommandGroup** constructor to accept one parameter. This is the drive train object. Add the following parameter within the parenthesis of the constructor:

```
public DriveInSquareCommandGroup(DriveTrainSubsystem m_driveTrainSubsystem) {
```

230) Add the following code which sequentially lists the commands we would like the robot to run in autonomous mode. Note that the commands are separated by a comma.

```
addCommands(

new DriveForwardTwoFeetCommand( m_driveTrainSubsystem),
new TurnRobotToLeft90( m_driveTrainSubsystem),

new DriveForwardTwoFeetCommand( m_driveTrainSubsystem),
new TurnRobotToLeft90( m_driveTrainSubsystem),

new DriveForwardTwoFeetCommand( m_driveTrainSubsystem),
new TurnRobotToLeft90( m_driveTrainSubsystem),

new DriveForwardTwoFeetCommand( m_driveTrainSubsystem),
new TurnRobotToLeft90( m_driveTrainSubsystem)

);
```

231) Select Control-Shift-P and type **"Organize Imports"**.

232) The JAVA code can be reformatted by selecting Shift-Alt F.

233) Open the **RobotContain.java** file.

234) Near the top of the class, add the following code in instantiate the command below the other code which instantiates the subsystems and joystick buttons (Near line 40):

```
private final DriveInSquareCommandGroup autonomousCommandOne = new DriveInSquareCommandGroup(m_driveTrainSubsystem);
```

235) Near the middle of the class, locate the method **getAutonomousCommand()** and update the return statement to the autonomous command group we created. (Near line 95)

```
Return autonomousCommandOne;
```

236) Select Control-Shift-P and type **"Organize Imports"**.

237) The JAVA code can be reformatted by selecting Shift-Alt F.

238) Save your work. Select **File** => **Save All.**

239) Deploy the code to the robot.

240) Place the robot in a clear area where it can drive and not cause damage.

(THINK SAFETY: Be sure everyone is clear of the Robot)

241) Open the Drivers station dashboard.

242) With **Teleop** selected, enable the robot and verify the operation of the Robot.

243) On the dashboard, select **autonomous** mode and press enable.

The robot should drive in a square to the left
Keep your hand near the Enter Key.
If unexpected behavior occurs, just press enter to disable the robot.

244) Observe the behavior of the Robot and verify correct operation.

245) Update the code to change the behavior.

Read an Analog Input Range Finder

This section describes how to read an analog input range finder. Examples of range finders are the Maxbotix MaxSonar series and the Sharp IR Analog Distance Sensor (GP2Y0A21YK0F).

A command will be written that when the "A" button is pressed and held, the robot drives forward at 30% speed and stops when the robot is 3 feet from a wall. The robot will stop when EITHER the button is released OR the robot get to within 3 feet of the wall.

This code assumes a Maxbotix sensor is connected to the RoboRIO analog port 0.

246) Open the **Constants.java** file.

247) At the top of the Constants.java file before the Operator Interface constants, add the analog port value.

```
// Analog inputs
public static final int RANGE_FINDER_PORT = 0;
```

248) Open the **DriveTrainSubsystem.java** file.

249) At the top of the file, add import statements to include the constants file and analog class.

```
import static frc.robot.Constants.RANGE_FINDER_PORT;
import edu.wpi.first.wpilibj.AnalogInput;
import edu.wpi.first.wpilibj.SPI;
```

250) Create variable for analog input at the top of the class just below the WPI_TalonSRX. (near line 60)

```
private AnalogInput Rangefinder;
```

251) At the bottom of the DriveTrainSubsystem **constructor**, add the following code to instantiate the rangefinder

```
Rangefinder = new AnalogInput(RANGE_FINDER_PORT);
```

252) At the bottom of the DriveTrainSubsystem class but before the **periodic()** method, add a method to read range finder:

```
  // for finding the distance from the range finder
        public double getRangeFinderDistance() {
    double rangefinderVoltage = Rangefinder.getAverageVoltage();
    double distanceInInches = (rangefinderVoltage * 65.4) - 7.2;
    return distanceInInches;
  }
```

253) Support testing of the RangeFinder method by adding code to the **manualDrive** method which displays the current rangefinder value:

```
   // Test the Rangefinder by displaying the current value on the shuffleboard
    double rangeToWall = getRangeFinderDistance();
    int rangeToWallInteger = (int) (rangeToWall);
    SmartDashboard.putNumber("Range to Wall", rangeToWallInteger);
```

254) In the left window, locate the folder called " **V commands**", right-click and select (at the bottom of the list) "**Create a new class/command**".

255) **Scroll down** and select **Command (new)** and enter the command name of **DriveToWallCommand**.

256) Update the **DriveToWallCommand** constructor to accept one parameter. This is the drivetrain object. Add the following parameter within the parenthesis of the constructor:

```
public DriveToWallCommand(DriveTrainSubsystem driveTrain) {
```

257) Within the constructor add the following line to assign the input parameter drivetrain to the drivetrain object used in this command:

```
m_driveTrain = driveTrain;
```

258) Identify which subsystem this command will be using. This information is placed within the class constructor. Add the following command at the bottom of the class constructor (Around line 18):

```
addRequirements(m_driveTrain);
```

259) Define the DriveTrain object and a number of other variables within this command by entering the following code at the top of the class and **before the constructor:**

```
// Reference to the constructed drive train from RobotContainer to be
// used to drive our robot
private final DriveTrainSubsystem m_driveTrain;
private double currentRangeToWall = 0;
private double targetRangeToWall = 24; // inches from wall to stop
private boolean atRangeToWall = true;
```

260) The `execute()` method runs 50 times per second. In this code, we read the rangefinder and compare the current range to the target range. If we are further away than the target distance, we command the motors to move forward at 50% otherwise we stop. We set a Boolean flag to stop the command.

```
public void execute() {
        currentRangeToWall = m_driveTrain.getRangeFinderDistance();
        if (currentRangeToWall > targetRangeToWall) {
          m_driveTrain.manualDrive(0.5, 0); // Drive straight forward at 50%
          atRangeToWall = false;
        }
        else{
          m_driveTrain.manualDrive(0, 0); // Drive straight forward at 50%
          atRangeToWall = true;
        }
    }
```

261) The `end()` method is run when the command completes. This is when the button is released. Update the end method with the code to drive the robot:

```
public void end(boolean interrupted) {
            m_driveTrain.manualDrive (0,0);
}
```

262) The `isFinished()` method is run right after the execute method to determine when a command is complete. While the button is held, the robot compares the current ranger finder distance to the target value. Update the `isFinished()` method with the code to stop the robot (Replace the variable after return) :

```
public boolean isFinished() {
       return atRangeToWall;
       }
```

263) Need to update the import files again. Select Control-Shift-P and type **"Organize Imports"**.

264) Open up the **RobotContainer.java** file.

265) Add a joystick button object to the code. Add the following line near the top of the RobotContainer Class just above the instantiation of the XboxController, DriveTrainSubsystem and DriveManuallyCommand command:

```
private JoystickButton driver_A;
```

266) Within the RobotContainer file, locate the "`configureButtonBindings()`" method. Add the following two lines to link the **"A Button**" to the command "**DriveToWallCommand**"

```
driver_A = new JoystickButton(m_driver_controller, XboxController.Button.kA.value);
driver_A.whileHeld (new DriveToWallCommand( m_driveTrainSubsystem));
```

267) Need to update the import files again. Select Control-Shift-P and type **"Organize Imports"**.

268) The JAVA code can be reformatted by selecting Shift-Alt F.

269) Save your work. Select **File => Save All.**

270) Download the code to the Robot by selecting the WPI Icon and typing **deploy Robot Code**.

271) Start the FRC Drivers Station application on the desktop of the workstation.

272) Select **TeleOperated** Mode.

273) Select **Enable** on the Drivers station to start the Robot.

274) Use the "Manual" steps in the section called "**Manually Displaying Data on the Shuffleboard**" to write the rangefinder value on the Shuffleboard after the code is complete.

Creating and Controlling an LED Subsystem

The WPI 2020 library can drive an WS2812 LED strip through the PWM port. We will create a new subsystem called LEDStrip and create a set of methods to set the colors. The LED strip can be used to signal the driver for items like distance to wall or direction the robot is travelling. The LED strips cost between $10 and $30 per meter on amazon depending on the number of LEDs per meter.

275) Open the **Constants** java file.

276) Add the following code at the top of the "Constants" class.

```
// LED
public static final int LED_STRIP_PORT = 0;
public static final int NUMBER_OF_LEDS = 6;
```

277) In the left window, locate the folder called " **V subsystems**", right-click and select (at the bottom of the list) "**Create a new class/command**".

278) **Scroll down** in the list and select "**Subsystem (New)**" and enter the name **LEDSubsystem** and press **Enter**.

279) Place your cursor on line 10 after the " */ " and select the Enter key three time to add space to create new code. *(The */ ends a comment block).*

280) Define variables which will become the LED Object. At line 11 enter the following code:

```
private AddressableLED m_led;
private AddressableLEDBuffer m_ledBuffer;
```

281) Select Control-Shift-P and type **"Organize Imports"**.

282) Locate the **Constructor** of the **LEDSubsystem**.
This is the line that starts with: `public LEDSubsystem() {`

283) Enter the following code on the next line within the constructor to instantiate the AddressableLED object..

```
// PWM port is defined by the constant LED_STRIP_PORT
m_led = new AddressableLED(LED_STRIP_PORT);

// Set the number of LEDs
m_ledBuffer = new AddressableLEDBuffer(NUMBER_OF_LEDS);
m_led.setLength(m_ledBuffer.getLength());

// Set the data
m_led.setData(m_ledBuffer);
m_led.start();
```

284) Next, we will create a method which can be called to set the LEDs to a color specified by RGB parameters passed in. Below the constructor, add the following code:

```
public void SetLEDColor(int red, int green, int blue) {

  for (var index = 0; index < m_ledBuffer.getLength(); index = index + 1) {

    m_ledBuffer.setRGB(index, red, green, blue); // Red, Green, Blue
  }
```

```
    m_led.setData(m_ledBuffer);
    m_led.start();
  }
```

285) On line 10 (after the line starting with package frc.robot.subsystems), enter the following statement. This line makes all of the defined constants available.

```
import static frc.robot.Constants.*;
```

286) The JAVA code can be reformatted by selecting Shift-Alt F.

287) Select **File** => **Save** to save the changes to the current code.

288) Open the DriveTrainSubsystem.java file.

289) At the top of the **DriveTrainSubsystem** class, create a variable for the LEDstrip object. (Near line 64)

```
private final LEDSubsystem m_LEDSubsystem;
```

290) Within the **DriveTrainSubsystem** class constructor, instantiate the LEDstrip object.

```
m_LEDSubsystem = new LEDSubsystem();
```

291) Test the LED Strip method by adding code to the **DriveTrainSubsystem** class **manualDrive** method which will set the LED color based on the position of the left joystick:

```
// Test the LED Strip
m_LEDSubsystem.SetLEDColor( ((int)(64-move*64)), ((int)(64+move*64)), 0); // Red Green Blue
```

292) Select Control-Shift-P and type **"Organize Imports"**.

293) The JAVA code can be reformatted by selecting Shift-Alt F.

294) Select **File** => **Save** to save the changes to the current code.

295) Download the code to the Robot by selecting the WPI Icon and typing **deploy Robot Code**.

296) Start the FRC Drivers Station application on the desktop of the workstation.

297) Select **TeleOperated** Mode.

298) Select **Enable** on the Drivers station to start the Robot.

299) Move the left Joystick and observe the LED strip change colors. When the joystick is full forward the LED Strip should be green and red when the Joystick is pulled back.

Complete Code Listing:

Files:

Constants.java
Main.java
Robot.java
RobotContainer.java

Subsystems:
- DriveTrainSubsystem.java
- LEDSubsystem.java

Commands:
- DriveForward50.java
- DriveForwardTwoFeetCommand.java
- DriveInSquareCommandGroup.java
- DriveManuallyCommand.java
- DriveToWallCommand.java
- TurnRobotToLeft90.java

sim:
- PhysicsSim.java
- TalonSRXSimProfile.java

Contents: ==========

==========

Constants.java

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package frc.robot;

import edu.wpi.first.wpilibj.kinematics.DifferentialDriveKinematics;
import edu.wpi.first.wpilibj.system.LinearSystem;
import edu.wpi.first.wpilibj.system.plant.DCMotor;
import edu.wpi.first.wpilibj.system.plant.LinearSystemId;
import edu.wpi.first.wpiutil.math.numbers.N2;

/**
 * The Constants class provides a convenient place for teams to hold robot-wide
 * numerical or boolean constants. This class should not be used for any other
 * purpose. All constants should be declared globally (i.e. public static). Do
 * not put anything functional in this class.
 *
 * <p>
 * It is advised to statically import this class (or one of its inner classes)
 * wherever the constants are needed, to reduce verbosity.
 */
public final class Constants {

        // LED
        public static final int LED_STRIP_PORT = 0;
        public static final int NUMBER_OF_LEDS = 6;

        // Analog inputs
        public static final int RANGE_FINDER_PORT = 0;
        // Operator Interface
        public static final int DRIVER_REMOTE_PORT = 0;
        public static final int DRIVER_RIGHT_AXIS = 0; // Use 4 for Gamepad and 0 for Joystick
        public static final int DRIVER_LEFT_AXIS = 1;

        // Talons
        public static final int LEFT_TALON_LEADER = 5;
        public static final int RIGHT_TALON_LEADER = 3;

        public static final class DriveConstants {
                // Shaft Encoders attached to the RoboRIO
                public static final int[] kLeftEncoderPorts = new int[] { 0, 1 }; // DIO ports
                public static final int[] kRightEncoderPorts = new int[] { 2, 3 };
                public static final boolean kLeftEncoderReversed = false;
                public static final boolean kRightEncoderReversed = true;

                public static final int kEncoderCPR = 1024;
                public static final double kWheelDiameterMeters = 0.15;
                public static final double kEncoderDistancePerPulse =
                                // Assumes the encoders are directly mounted on the wheel shafts
                                (kWheelDiameterMeters * Math.PI) / (double) kEncoderCPR;

                public static final boolean kGyroReversed = true;

                // Constants for Simulation
                public static final double kTrackwidthMeters = 0.69;
                public static final DifferentialDriveKinematics kDriveKinematics = new DifferentialDriveKinematics(
                                kTrackwidthMeters);

                // These are example values only - DO NOT USE THESE FOR YOUR OWN ROBOT!
                // These characterization values MUST be determined either experimentally or
                // theoretically
                // for *your* robot's drive.
                // The Robot Characterization Toolsuite provides a convenient tool for obtaining
                // these
                // values for your robot.
                public static final double ksVolts = 0.22;
                public static final double kvVoltSecondsPerMeter = 1.98;
                public static final double kaVoltSecondsSquaredPerMeter = 0.2;

                // These are example values only - DO NOT USE THESE FOR YOUR OWN ROBOT!
                // These characterization values MUST be determined either experimentally or
                // theoretically
                // for *your* robot's drive.
                // These two values are "angular" kV and kA
                public static final double kvVoltSecondsPerRadian = 1.5;
                public static final double kaVoltSecondsSquaredPerRadian = 0.3;

                // Used for Simulation
                public static final LinearSystem<N2, N2, N2> kDrivetrainPlant = LinearSystemId.identifyDrivetrainSystem(
                                kvVoltSecondsPerMeter, kaVoltSecondsSquaredPerMeter, kvVoltSecondsPerRadian,
                                kaVoltSecondsSquaredPerRadian);

                // Example values only -- use what's on your physical robot!
                public static final DCMotor kDriveGearbox = DCMotor.getCIM(2);
                public static final double kDriveGearing = 8;

        }

}
```

==========

main.java

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package frc.robot;

import edu.wpi.first.wpilibj.RobotBase;

/**
 * Do NOT add any static variables to this class, or any initialization at all. Unless you know what
 * you are doing, do not modify this file except to change the parameter class to the startRobot
 * call.
 */
public final class Main {
  private Main() {}

  /**
   * Main initialization function. Do not perform any initialization here.
   *
   * <p>If you change your main robot class, change the parameter type.
   */
  public static void main(String... args) {
    RobotBase.startRobot(Robot::new);
  }
}
```

==========

Robot.java

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package frc.robot;

import edu.wpi.first.wpilibj.TimedRobot;
import edu.wpi.first.wpilibj2.command.Command;
import edu.wpi.first.wpilibj2.command.CommandScheduler;

/**
 * The VM is configured to automatically run this class, and to call the functions corresponding to
 * each mode, as described in the TimedRobot documentation. If you change the name of this class or
 * the package after creating this project, you must also update the build.gradle file in the
 * project.
 */
public class Robot extends TimedRobot {
  private Command m_autonomousCommand;

  private RobotContainer m_robotContainer;

  /**
   * This function is run when the robot is first started up and should be used for any
   * initialization code.
   */
  @Override
  public void robotInit() {
    // Instantiate our RobotContainer.  This will perform all our button bindings, and put our
    // autonomous chooser on the dashboard.
    m_robotContainer = new RobotContainer();
  }

  /**
   * This function is called every robot packet, no matter the mode. Use this for items like
   * diagnostics that you want ran during disabled, autonomous, teleoperated and test.
   *
   * <p>This runs after the mode specific periodic functions, but before LiveWindow and
   * SmartDashboard integrated updating.
   */
  @Override
  public void robotPeriodic() {
    // Runs the Scheduler.  This is responsible for polling buttons, adding newly-scheduled
    // commands, running already-scheduled commands, removing finished or interrupted commands,
    // and running subsystem periodic() methods.  This must be called from the robot's periodic
    // block in order for anything in the Command-based framework to work.
    CommandScheduler.getInstance().run();
  }

  /** This function is called once each time the robot enters Disabled mode. */
  @Override
  public void disabledInit() {}

  @Override
  public void disabledPeriodic() {}

  /** This autonomous runs the autonomous command selected by your {@link RobotContainer} class. */
  @Override
  public void autonomousInit() {
    m_autonomousCommand = m_robotContainer.getAutonomousCommand();

    // schedule the autonomous command (example)
    if (m_autonomousCommand != null) {
      m_autonomousCommand.schedule();
    }
  }

  /** This function is called periodically during autonomous. */
  @Override
  public void autonomousPeriodic() {}

  @Override
  public void teleopInit() {
    // This makes sure that the autonomous stops running when
    // teleop starts running. If you want the autonomous to
    // continue until interrupted by another command, remove
    // this line or comment it out.
    if (m_autonomousCommand != null) {
      m_autonomousCommand.cancel();
    }
  }

  /** This function is called periodically during operator control. */
  @Override
  public void teleopPeriodic() {}

  @Override
  public void testInit() {
    // Cancels all running commands at the start of test mode.
    CommandScheduler.getInstance().cancelAll();
  }

  /** This function is called periodically during test mode. */
  @Override
  public void testPeriodic() {}
}
```

==========

RobotContainer.java

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package frc.robot;

import static frc.robot.Constants.DRIVER_LEFT_AXIS;
import static frc.robot.Constants.DRIVER_REMOTE_PORT;
import static frc.robot.Constants.DRIVER_RIGHT_AXIS;

import edu.wpi.first.wpilibj.GenericHID;
import edu.wpi.first.wpilibj.XboxController;
import edu.wpi.first.wpilibj.shuffleboard.BuiltInWidgets;
import edu.wpi.first.wpilibj.shuffleboard.Shuffleboard;
import edu.wpi.first.wpilibj2.command.Command;
import edu.wpi.first.wpilibj2.command.button.JoystickButton;
import frc.robot.commands.DriveForward50;
import frc.robot.commands.DriveForwardTwoFeetCommand;
import frc.robot.commands.DriveInSquareCommandGroup;
import frc.robot.commands.DriveManuallyCommand;
import frc.robot.commands.DriveToWallCommand;
import frc.robot.commands.TurnRobotToLeft90;
import frc.robot.subsystems.DriveTrainSubsystem;

/**
 * This class is where the bulk of the robot should be declared. Since
 * Command-based is a "declarative" paradigm, very little robot logic should
 * actually be handled in the {@link Robot} periodic methods (other than the
 * scheduler calls). Instead, the structure of the robot (including subsystems,
 * commands, and button mappings) should be declared here.
 */
public class RobotContainer {
  // The robot's subsystems and commands are defined here...

  private final XboxController m_driver_controller = new XboxController(DRIVER_REMOTE_PORT);
  private final DriveTrainSubsystem m_driveTrainSubsystem = new DriveTrainSubsystem();
  private final DriveManuallyCommand m_DriveManuallyCommand = new DriveManuallyCommand(m_driveTrainSubsystem,
      m_driver_controller);
  private JoystickButton driver_X;
  private JoystickButton driver_Y;
  private JoystickButton driver_B;
  private JoystickButton driver_A;

  private final DriveInSquareCommandGroup autonomousCommandOne = new DriveInSquareCommandGroup(m_driveTrainSubsystem);

  /**
   * The container for the robot. Contains subsystems, OI devices, and commands.
   */
  public RobotContainer() {
    // Configure the button bindings
    configureButtonBindings();
    m_driveTrainSubsystem.setDefaultCommand(new DriveManuallyCommand(m_driveTrainSubsystem, m_driver_controller));

    // Add information to the Shuffleboard for monitoring and Troubleshooting
    Shuffleboard.getTab("DriveTrainDisplay");

    // This code adds values to the "DriveTrainDisplay" Tab on the Shuffleboard.
    // This code is dependent on two methods added to the RobotContainer to access
    // the data to be displayed
    Shuffleboard.getTab("DriveTrainDisplay").addNumber("Joystick-Y", this::getJoystickYValue)
        .withWidget(BuiltInWidgets.kNumberSlider).withPosition(6, 3);

    Shuffleboard.getTab("DriveTrainDisplay").addNumber("Joystick-X", this::getJoystickXValue)
        .withWidget(BuiltInWidgets.kNumberSlider).withPosition(6, 4);

    // This adds a command to the Shuffleboard
    Shuffleboard.getTab("DriveTrainDisplay").add(new DriveForward50(m_driveTrainSubsystem)).withPosition(6, 0)
        .withSize(2, 2);

  }

  /**
   * Use this method to define your button->command mappings. Buttons can be
   * created by instantiating a {@link GenericHID} or one of its subclasses
   * ({@link edu.wpi.first.wpilibj.Joystick} or {@link XboxController}), and then
   * passing it to a {@link edu.wpi.first.wpilibj2.command.button.JoystickButton}.
   */
  private void configureButtonBindings() {
    driver_X = new JoystickButton(m_driver_controller, XboxController.Button.kX.value);
    driver_X.whileHeld(new DriveForward50(m_driveTrainSubsystem));

    driver_Y = new JoystickButton(m_driver_controller, XboxController.Button.kY.value);
    driver_Y.whenPressed(new DriveForwardTwoFeetCommand(m_driveTrainSubsystem));

    driver_B = new JoystickButton(m_driver_controller, XboxController.Button.kB.value);
    driver_B.whenPressed(new TurnRobotToLeft90(m_driveTrainSubsystem));

    driver_A = new JoystickButton(m_driver_controller, XboxController.Button.kA.value);
    driver_A.whileHeld(new DriveToWallCommand(m_driveTrainSubsystem));
  }

  /**
   * Use this to pass the autonomous command to the main {@link Robot} class.
   *
   * @return the command to run in autonomous
   */
  public Command getAutonomousCommand() {
    // An ExampleCommand will run in autonomous

    return autonomousCommandOne;
  }

  // These are used by the commands to add data to the shuffleboard (See the
  // RobotContainer Constructor)
  public double getJoystickXValue() {
    return m_driver_controller.getRawAxis(DRIVER_RIGHT_AXIS);
  }

  public double getJoystickYValue() {
    return m_driver_controller.getRawAxis(DRIVER_LEFT_AXIS);
  }
}
```

==========

DriveTrainSubsystem.java

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package frc.robot.subsystems;

import static frc.robot.Constants.LEFT_TALON_LEADER;
import static frc.robot.Constants.RIGHT_TALON_LEADER;

import com.ctre.phoenix.motorcontrol.can.WPI_TalonSRX;
import com.kauailabs.navx.frc.AHRS;

import edu.wpi.first.hal.SimDouble;
import edu.wpi.first.hal.simulation.SimDeviceDataJNI;
import edu.wpi.first.wpilibj.ADXRS450_Gyro;
import edu.wpi.first.wpilibj.Encoder;
import edu.wpi.first.wpilibj.RobotBase;
import edu.wpi.first.wpilibj.RobotController;
import edu.wpi.first.wpilibj.SPI;
import edu.wpi.first.wpilibj.Timer;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import edu.wpi.first.wpilibj.geometry.Pose2d;
import edu.wpi.first.wpilibj.geometry.Rotation2d;
import edu.wpi.first.wpilibj.kinematics.DifferentialDriveOdometry;
import edu.wpi.first.wpilibj.simulation.ADXRS450_GyroSim;
import edu.wpi.first.wpilibj.simulation.DifferentialDrivetrainSim;
import edu.wpi.first.wpilibj.simulation.EncoderSim;
import edu.wpi.first.wpilibj.smartdashboard.Field2d;
import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;
import edu.wpi.first.wpilibj2.command.SubsystemBase;
import edu.wpi.first.wpiutil.math.VecBuilder;
import frc.robot.Constants.DriveConstants;
import frc.robot.sim.PhysicsSim;
import static frc.robot.Constants.RANGE_FINDER_PORT;
import edu.wpi.first.wpilibj.AnalogInput;
import edu.wpi.first.wpilibj.SPI;

public class DriveTrainSubsystem extends SubsystemBase {
  /** Creates a new DriveTrainSubsystem. */

  private final Timer timer1;
  private final WPI_TalonSRX m_rightLeader; // Declare motor controllers variables
  private final WPI_TalonSRX m_leftLeader;
  private final DifferentialDrive m_safety_drive; // Declare drive train core function

  // The left-side drive encoder (connected to the RoboRIO DIOs)
  private final Encoder m_leftEncoder = new Encoder(DriveConstants.kLeftEncoderPorts[0],
      DriveConstants.kLeftEncoderPorts[1], DriveConstants.kLeftEncoderReversed);

  // The right-side drive encoder (connected to the RoboRIO DIOs)
  private final Encoder m_rightEncoder = new Encoder(DriveConstants.kRightEncoderPorts[0],
      DriveConstants.kRightEncoderPorts[1], DriveConstants.kRightEncoderReversed);

  // Add the stand-alone gyro sensor
  private final ADXRS450_Gyro m_gyro = new ADXRS450_Gyro();

  private final AHRS ahrs;

  private int percentComplete = 0;

  private AnalogInput Rangefinder;

  private final LEDSubsystem m_LEDSubsystem;

  // ==(Added for simulation)================================================
  private Field2d m_fieldSim;

  // These classes help us simulate our drivetrain
  public DifferentialDrivetrainSim m_drivetrainSimulator;
  private EncoderSim m_leftEncoderSim;
  private EncoderSim m_rightEncoderSim;
  private ADXRS450_GyroSim m_gyroSim;

  // Odometry class for tracking robot pose
  private final DifferentialDriveOdometry m_odometry = new DifferentialDriveOdometry(
      Rotation2d.fromDegrees(getHeading()));
  // ========================================================================

  public DriveTrainSubsystem() {

    timer1 = new Timer();
    timer1.start();

    m_rightLeader = new WPI_TalonSRX(RIGHT_TALON_LEADER); // Instantiate motor controllers
    m_leftLeader = new WPI_TalonSRX(LEFT_TALON_LEADER);
    m_safety_drive = new DifferentialDrive(m_leftLeader, m_rightLeader);
    /* Factory Default for Talons */
    m_rightLeader.configFactoryDefault(); // Defaults to reading quad encoder
    m_leftLeader.configFactoryDefault();
    m_rightLeader.setSensorPhase(true); // Invert the Right Talon's quad encoder value

    // ========================================================================
    // Configure the external encoders
    // Sets the distance per pulse for the encoders
    m_leftEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);
```

```
    m_rightEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);

    // Reset the external encoders
    resetEncoders();

    ahrs = new AHRS(SPI.Port.kMXP);
    Rangefinder = new AnalogInput(RANGE_FINDER_PORT);
    m_LEDSubsystem = new LEDSubsystem();

    // ================================================================
    // Code for simulation within the DriveTrain Constructor
    if (RobotBase.isSimulation()) { // If our robot is simulated
      // This class simulates our drivetrain's motion around the field.
      m_drivetrainSimulator = new DifferentialDrivetrainSim(DriveConstants.kDrivetrainPlant,
          DriveConstants.kDriveGearbox, DriveConstants.kDriveGearing, DriveConstants.kTrackwidthMeters,
          DriveConstants.kWheelDiameterMeters / 2.0, VecBuilder.fill(0, 0, 0.0001, 0.1, 0.1, 0.005, 0.005));

      // The encoder and gyro angle sims let us set simulated sensor readings
      m_leftEncoderSim = new EncoderSim(m_leftEncoder);
      m_rightEncoderSim = new EncoderSim(m_rightEncoder);
      m_gyroSim = new ADXRS450_GyroSim(m_gyro);

      // the Field2d class lets us visualize our robot in the simulation GUI.
      m_fieldSim = new Field2d();
      SmartDashboard.putData("Field", m_fieldSim);

      PhysicsSim.getInstance().addTalonSRX(m_rightLeader, 0.75, 4000);
      PhysicsSim.getInstance().addTalonSRX(m_leftLeader, 0.75, 4000);
    } // end of constructor code for the simulation

  }

  /**
   * Returns the heading of the robot.
   *
   * @return the robot's heading in degrees, from -180 to 180
   */
  public double getHeading() {
    return Math.IEEEremainder(m_gyro.getAngle(), 360) * (DriveConstants.kGyroReversed ? -1.0 : 1.0);
  }

  // Talan Motor Control - Encoders
  public double getLeftEncoderValue() {
    return m_leftLeader.getSelectedSensorPosition();
  }

  public double getRightEncoderValue() {
    return m_rightLeader.getSelectedSensorPosition();
  }

  public void reset_drivetrain_encoders() {
    m_leftLeader.setSelectedSensorPosition(0, 0, 0);
    m_rightLeader.setSelectedSensorPosition(0, 0, 0);
  }

  /** Resets the Talon drive encoders to currently read a position of 0. */
  public void resetEncoders() {
    m_leftEncoder.reset();
    m_rightEncoder.reset();
  }

  public void manualDrive(double move, double turn) {
    m_safety_drive.arcadeDrive(move, turn);
    m_safety_drive.feed();

    // Test the Gyro by displaying the current value on the shuffleboard
    double currentHeading = get_current_heading();
    int currentHeadingInteger = (int) (currentHeading);
    SmartDashboard.putNumber("RobotHeading", currentHeadingInteger);

    // Test the LED Strip
    m_LEDSubsystem.SetLEDColor(((int) (64 - move * 64)), ((int) (64 + move * 64)), 0); // Red Green Blue

  }

  public void reset_gyro() {
    ahrs.reset();
  }

  public double get_current_heading() {
    return ahrs.getAngle();
  }

  // ==(Added for Simulation)=========================================
  /**
   * Returns the currently-estimated pose of the robot.
   *
   * @return The pose.
   */
  public Pose2d getPose() {
    return m_odometry.getPoseMeters();
  }

  public Boolean driveForwardInches(double motorSpeed, double inchesToDrive, Boolean resetEncoder) {

    Boolean driveComplete = false;
    double currentShaftEncoderValue = 0;
    double targetShaftEncoderCount = 0;
    double convertRotationsToInches = 164; // Will need tested, calibrated and revised
```

```
    // Reset the shaft encoder value
    if (resetEncoder == true) {
      reset_drivetrain_encoders();

      // System.out.println("Drive Forward - reset encoder: " + getTimer1Value() + "
      // Heading:" + get_current_heading() + " Enc: " + getRightEncoderValue());

    }

    // Set the speed of the motors using arcade drive with no turn
    m_safety_drive.arcadeDrive(motorSpeed, 0);
    m_safety_drive.feed();

    // Check the encoder
    currentShaftEncoderValue = getRightEncoderValue();
    targetShaftEncoderCount = inchesToDrive * convertRotationsToInches;

    if (currentShaftEncoderValue > targetShaftEncoderCount) {
      driveComplete = true;
      // System.out.println("Drive Forward - complete: " + getTimer1Value() + "
      // Heading:" + get_current_heading() + " Enc: " + getRightEncoderValue());
    }

    percentComplete = (int) (100 * (currentShaftEncoderValue / targetShaftEncoderCount));
    SmartDashboard.putNumber("Encoder % Complete", percentComplete);

    return driveComplete;
  }

  // for finding the distance from the range finder
  public double getRangeFinderDistance() {
    double rangefinderVoltage = Rangefinder.getAverageVoltage();
    double distanceInInches = (rangefinderVoltage * 65.4) - 7.2;
    return distanceInInches;
  }

  @Override
  public void periodic() {

    // ==(Added for Simulation)==========================================
    // Update the odometry in the periodic block
    // Read the current heading (not sure where from) and encoder values and feed to
    // Odometry model
    m_odometry.update(Rotation2d.fromDegrees(getHeading()), m_leftEncoder.getDistance(), m_rightEncoder.getDistance());

    // Get the pose from the drive train and send it to the simulated field.
    m_fieldSim.setRobotPose(getPose());

    // ==(Added for Testing and Troubleshooting)=================================

    int encoder_output = m_rightEncoder.getRaw();
    SmartDashboard.putNumber("encoder_output:  ", encoder_output);

    double encoder_distance_output = m_rightEncoder.getDistance();
    SmartDashboard.putNumber("encoder_distance_output:  ", encoder_distance_output);

    double left_Talon_encoder_output = getLeftEncoderValue();
    SmartDashboard.putNumber("left_Talon_encoder_output:  ", left_Talon_encoder_output);

    double right_Talon_encoder_output = getRightEncoderValue();
    SmartDashboard.putNumber("right_Talon_encoder_output:  ", right_Talon_encoder_output);

    SmartDashboard.putNumber("NAVX Gyro output:  ", ahrs.getAngle());

  }

  @Override
  public void simulationPeriodic() {
    // To update our simulation, we set motor voltage inputs, update the simulation,
    // and write the simulated positions and velocities to our simulated encoder and
    // gyro.
    // We negate the right side so that positive voltages make the right side
    // move forward.

    PhysicsSim.getInstance().run();

    m_drivetrainSimulator.setInputs(m_leftLeader.get() * RobotController.getBatteryVoltage(),
        -m_rightLeader.get() * RobotController.getBatteryVoltage());
    m_drivetrainSimulator.update(0.020);

    m_leftEncoderSim.setDistance(m_drivetrainSimulator.getLeftPositionMeters());
    m_leftEncoderSim.setRate(m_drivetrainSimulator.getLeftVelocityMetersPerSecond());
    m_rightEncoderSim.setDistance(m_drivetrainSimulator.getRightPositionMeters());
    m_rightEncoderSim.setRate(m_drivetrainSimulator.getRightVelocityMetersPerSecond());
    m_gyroSim.setAngle(-m_drivetrainSimulator.getHeading().getDegrees());

    int dev = SimDeviceDataJNI.getSimDeviceHandle("navX-Sensor[0]");
    SimDouble angle = new SimDouble(SimDeviceDataJNI.getSimValueHandle(dev, "Yaw"));
    angle.set(-m_drivetrainSimulator.getHeading().getDegrees());
  }

  public double getTimer1Value() {
    return timer1.get();
  }

}
```

==========

LEDSubsystem.java

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package frc.robot.subsystems;

import static frc.robot.Constants.LED_STRIP_PORT;
import static frc.robot.Constants.NUMBER_OF_LEDS;

import edu.wpi.first.wpilibj.AddressableLED;
import edu.wpi.first.wpilibj.AddressableLEDBuffer;
import edu.wpi.first.wpilibj2.command.SubsystemBase;

public class LEDSubsystem extends SubsystemBase {
  /** Creates a new LEDSubsystem. */
  private AddressableLED m_led;
  private AddressableLEDBuffer m_ledBuffer;



  public LEDSubsystem() {
         // PWM port is defined by the constant LED_STRIP_PORT
        m_led = new AddressableLED(LED_STRIP_PORT);

        // Set the number of LEDs
        m_ledBuffer = new AddressableLEDBuffer(NUMBER_OF_LEDS);
        m_led.setLength(m_ledBuffer.getLength());

        // Set the data
        m_led.setData(m_ledBuffer);
        m_led.start();

  }

  public void SetLEDColor(int red, int green, int blue) {

    for (var index = 0; index < m_ledBuffer.getLength(); index = index + 1) {

      m_ledBuffer.setRGB(index, red, green, blue); // Red, Green, Blue
    }
    m_led.setData(m_ledBuffer);
    m_led.start();
  }


  @Override
  public void periodic() {
    // This method will be called once per scheduler run
  }
}
```

==========

DriveForward50.java

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package frc.robot.commands;

import edu.wpi.first.wpilibj2.command.CommandBase;
import frc.robot.subsystems.DriveTrainSubsystem;

public class DriveForward50 extends CommandBase {
   // Reference to the constructed drive train from RobotContainer to be
  // used to drive our robot
  private final DriveTrainSubsystem m_driveTrain;


  /** Creates a new DriveForward50. */
  public DriveForward50(DriveTrainSubsystem driveTrain) {
    m_driveTrain = driveTrain;
    // Use addRequirements() here to declare subsystem dependencies.
    addRequirements(m_driveTrain);
  }

  // Called when the command is initially scheduled.
  @Override
  public void initialize() {}

  // Called every time the scheduler runs while the command is scheduled.
  @Override
  public void execute() {
    m_driveTrain.manualDrive (0.5,0);  // Drive straight forward at 50%
  }

  // Called once the command ends or is interrupted.
  @Override
  public void end(boolean interrupted) {
    m_driveTrain.manualDrive (0,0);
  }

  // Returns true when the command should end.
  @Override
  public boolean isFinished() {
    return false;
  }
}
```

==========

DriveForwardTwoFeetCommand.java

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package frc.robot.commands;

import edu.wpi.first.wpilibj2.command.CommandBase;
import frc.robot.subsystems.DriveTrainSubsystem;

public class DriveForwardTwoFeetCommand extends CommandBase {

  int counter = 0;
  // Reference to the constructed drive train from RobotContainer to be
  // used to drive our robot
  private final DriveTrainSubsystem m_driveTrain;
  private boolean m_driveComplete;
  private double driveSpeed = 0.3;
  private double driveDistanceInches = 3 * 39.37; // 39.37 inches per meter

  /** Creates a new DriveForwardTwoFeetCommand. */
  public DriveForwardTwoFeetCommand(DriveTrainSubsystem driveTrain) {
    // Use addRequirements() here to declare subsystem dependencies.

    m_driveTrain = driveTrain;
    addRequirements(m_driveTrain);
  }

  // Called when the command is initially scheduled.
  @Override
  public void initialize() {
    m_driveComplete = m_driveTrain.driveForwardInches(driveSpeed, driveDistanceInches, true);

    // System.out.println("Drive Forward - reset encoder in command " + m_driveTrain.getTimer1Value() + "  Enc: "
    //     + m_driveTrain.getRightEncoderValue());

  }

  // Called every time the scheduler runs while the command is scheduled.
  @Override
  public void execute() {
    m_driveComplete = m_driveTrain.driveForwardInches(driveSpeed, driveDistanceInches, false);

    counter = counter + 1;
    if (counter > 10) {
    //  System.out
    //      .println("Drive Forward: " + m_driveTrain.getTimer1Value() + "  Enc: " + m_driveTrain.getRightEncoderValue());
      counter = 0;
    }

  }

  // Called once the command ends or is interrupted.
  @Override
  public void end(boolean interrupted) {
    boolean not_used = m_driveTrain.driveForwardInches(0, 0, true);
    //System.out.println("Drive Forward - reset encoder in command " + m_driveTrain.getTimer1Value() + "  Enc: "
    //    + m_driveTrain.getRightEncoderValue());
  }

  // Returns true when the command should end.
  @Override
  public boolean isFinished() {
    return m_driveComplete;
  }
}
```

==========

DriveInSquareCommandGroup.java

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package frc.robot.commands;

import edu.wpi.first.wpilibj2.command.SequentialCommandGroup;
import frc.robot.subsystems.DriveTrainSubsystem;

// NOTE:  Consider using this command inline, rather than writing a subclass.  For more
// information, see:
// https://docs.wpilib.org/en/stable/docs/software/commandbased/convenience-features.html
public class DriveInSquareCommandGroup extends SequentialCommandGroup {
  /** Creates a new DriveInSquareCommandGroup. */
  public DriveInSquareCommandGroup(DriveTrainSubsystem m_driveTrainSubsystem) {
    // Add your commands in the addCommands() call, e.g.
    // addCommands(new FooCommand(), new BarCommand());
    addCommands(

        new DriveForwardTwoFeetCommand(m_driveTrainSubsystem),
        new TurnRobotToLeft90(m_driveTrainSubsystem),

        new DriveForwardTwoFeetCommand(m_driveTrainSubsystem),
        new TurnRobotToLeft90(m_driveTrainSubsystem),

        new DriveForwardTwoFeetCommand(m_driveTrainSubsystem),
        new TurnRobotToLeft90(m_driveTrainSubsystem),

        new DriveForwardTwoFeetCommand(m_driveTrainSubsystem),
        new TurnRobotToLeft90(m_driveTrainSubsystem)

    );

  }
}
```

==========

DriveManuallyCommand.java

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package frc.robot.commands;

import edu.wpi.first.wpilibj.XboxController;
import edu.wpi.first.wpilibj2.command.CommandBase;
import frc.robot.subsystems.DriveTrainSubsystem;
import static frc.robot.Constants.*;

public class DriveManuallyCommand extends CommandBase {
  /** Creates a new DriveManuallyCommand. */

  // Reference to the constructed drive train from RobotContainer to be
  // used to drive our robot
  private final DriveTrainSubsystem m_driveTrain;
  private final XboxController m_driverController;

  public DriveManuallyCommand(DriveTrainSubsystem driveTrain, XboxController driverController) {
    // Use addRequirements() here to declare subsystem dependencies.
    m_driveTrain = driveTrain;
    m_driverController = driverController;

    addRequirements(m_driveTrain);
  }

  // Called when the command is initially scheduled.
  @Override
  public void initialize() {
  }

  // Called every time the scheduler runs while the command is scheduled.
  @Override
  public void execute() {
    // Axis are inverted, negate them so positive is forward
    double turn = m_driverController.getRawAxis(DRIVER_RIGHT_AXIS); // Right X
    double move = -m_driverController.getRawAxis(DRIVER_LEFT_AXIS); // Left Y

    m_driveTrain.manualDrive(move, turn);

  }

  // Called once the command ends or is interrupted.
  @Override
  public void end(boolean interrupted) {
  }

  // Returns true when the command should end.
  @Override
  public boolean isFinished() {
    return false;
  }
}
```

==========

DriveToWallCommand.java

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package frc.robot.commands;

import edu.wpi.first.wpilibj2.command.CommandBase;
import frc.robot.subsystems.DriveTrainSubsystem;

public class DriveToWallCommand extends CommandBase {
  /** Creates a new DriveToWallCommand. */

  // Reference to the constructed drive train from RobotContainer to be
  // used to drive our robot
  private final DriveTrainSubsystem m_driveTrain;
  private double currentRangeToWall = 0;
  private double targetRangeToWall = 24; // inches from wall to stop
  private boolean atRangeToWall = true;

  public DriveToWallCommand(DriveTrainSubsystem driveTrain) {
    // Use addRequirements() here to declare subsystem dependencies.
    m_driveTrain = driveTrain;

    addRequirements(m_driveTrain);
  }

  // Called when the command is initially scheduled.
  @Override
  public void initialize() {

  }

  // Called every time the scheduler runs while the command is scheduled.
  @Override
  public void execute() {
    currentRangeToWall = m_driveTrain.getRangeFinderDistance();
    if (currentRangeToWall > targetRangeToWall) {
      m_driveTrain.manualDrive(0.5, 0); // Drive straight forward at 50%
      atRangeToWall = false;
    } else {
      m_driveTrain.manualDrive(0, 0); // Drive straight forward at 50%
      atRangeToWall = true;
    }
  }

  // Called once the command ends or is interrupted.
  @Override
  public void end(boolean interrupted) {
    m_driveTrain.manualDrive(0, 0);
  }

  // Returns true when the command should end.
  @Override
  public boolean isFinished() {
    return atRangeToWall;
  }
}
```

==========

TurnRobotToLeft90.java

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package frc.robot.commands;

import edu.wpi.first.wpilibj2.command.CommandBase;
import frc.robot.subsystems.DriveTrainSubsystem;

public class TurnRobotToLeft90 extends CommandBase {

  // Reference to the constructed drive train from RobotContainer to be
  // used to drive our robot
  private final DriveTrainSubsystem m_driveTrain;
  private double currentGyroHeading = 0;
  private double targetGyroHeading = -90;
  private boolean TurnComplete = true;

  /** Creates a new TurnRobotToLeft90. */
  public TurnRobotToLeft90(DriveTrainSubsystem driveTrain) {
    // Use addRequirements() here to declare subsystem dependencies.
    m_driveTrain = driveTrain;
    addRequirements(m_driveTrain);
  }

  // Called when the command is initially scheduled.
  @Override
  public void initialize() {
    m_driveTrain.reset_gyro();
    System.out.println("Turn to left -- Gyro Init: " + m_driveTrain.getTimer1Value() + "  " + m_driveTrain.get_current_heading());
  }

  // Called every time the scheduler runs while the command is scheduled.
  @Override
  public void execute() {
    currentGyroHeading = m_driveTrain.get_current_heading();
    if (currentGyroHeading > targetGyroHeading) {
      m_driveTrain.manualDrive(0, -0.15); // Rotate at 50% power  (Positive turns right)
      TurnComplete = false;
    } else {
      m_driveTrain.manualDrive(0, 0);
      TurnComplete = true;
      System.out.println("Turn to left -- Turn Complete: " + m_driveTrain.getTimer1Value() + "  " + m_driveTrain.get_current_heading());
    }

  }

  // Called once the command ends or is interrupted.
  @Override
  public void end(boolean interrupted) {
    m_driveTrain.manualDrive(0, 0);
  }

  // Returns true when the command should end.
  @Override
  public boolean isFinished() {
    return TurnComplete;
  }
}
```

==========

PhysicsSim.java

```
//=====================================================
// D.Frederick:
// This file is provided by CTRE to support the simulation of the Talon SRX
// Modified to remove the code for the Victor
//=====================================================
package frc.robot.sim;

import java.util.*;
import com.ctre.phoenix.motorcontrol.can.*;

/**
 * Manages physics simulation for CTRE products.
 */
public class PhysicsSim {
    private static final PhysicsSim sim = new PhysicsSim();

    /**
     * Gets the robot simulator instance.
     */
    public static PhysicsSim getInstance() {
        return sim;
    }

    /**
     * Adds a TalonSRX controller to the simulator.
     *
     * @param talon           The TalonSRX device
     * @param accelToFullTime The time the motor takes to accelerate from 0 to full,
     *                        in seconds
     * @param fullVel         The maximum motor velocity, in ticks per 100ms
     */
    public void addTalonSRX(TalonSRX talon, final double accelToFullTime, final double fullVel) {
        addTalonSRX(talon, accelToFullTime, fullVel, false);
    }

    /**
     * Adds a TalonSRX controller to the simulator.
     *
     * @param talon           The TalonSRX device
     * @param accelToFullTime The time the motor takes to accelerate from 0 to full,
     *                        in seconds
     * @param fullVel         The maximum motor velocity, in ticks per 100ms
     * @param sensorPhase     The phase of the TalonSRX sensors
     */
    public void addTalonSRX(TalonSRX talon, final double accelToFullTime, final double fullVel,
            final boolean sensorPhase) {
        if (talon != null) {
            TalonSRXSimProfile simTalon = new TalonSRXSimProfile(talon, accelToFullTime, fullVel, sensorPhase);
            _simProfiles.add(simTalon);
        }
    }

    /**
     * Runs the simulator: - enable the robot - simulate TalonSRX sensors
     */
    public void run() {
        // Simulate devices
        for (SimProfile simProfile : _simProfiles) {
            simProfile.run();
        }
    }

    private final ArrayList<SimProfile> _simProfiles = new ArrayList<SimProfile>();

    /*
     * scales a random domain of [0, 2pi] to [min, max] while prioritizing the peaks
     */
    static double random(double min, double max) {
        return (max - min) / 2 * Math.sin(Math.IEEEremainder(Math.random(), 2 * 3.14159)) + (max + min) / 2;
    }

    static double random(double max) {
        return random(0, max);
    }

    /**
     * Holds information about a simulated device.
     */
    static class SimProfile {
        private long _lastTime;
        private boolean _running = false;

        /**
         * Runs the simulation profile. Implemented by device-specific profiles.
         */
        public void run() {
        }

        /**
         * Returns the time since last call, in milliseconds.
         */
        protected double getPeriod() {
            // set the start time if not yet running
            if (!_running) {
                _lastTime = System.nanoTime();
                _running = true;
            }

            long now = System.nanoTime();
            final double period = (now - _lastTime) / 1000000.;
            _lastTime = now;

            return period;
        }
    }
}
```

==========

TalonSRXSimProfile.java

```
//=============================================================
// D.Frederick:
// This file is provided by CTRE to support the simulation of the Talon SRX
//=============================================================
package frc.robot.sim;

import frc.robot.sim.PhysicsSim.*;
import static frc.robot.sim.PhysicsSim.*; // random()

import com.ctre.phoenix.motorcontrol.can.*;

/**
 * Holds information about a simulated TalonSRX.
 */
class TalonSRXSimProfile extends SimProfile {
    private final TalonSRX _talon;
    private final double _accelToFullTime;
    private final double _fullVel;
    private final boolean _sensorPhase;

    /** The current position */
    private double _pos = 0;

    /** The current velocity */
    private double _vel = 0;

    /**
     * Creates a new simulation profile for a TalonSRX device.
     *
     * @param talon
     *        The TalonSRX device
     * @param accelToFullTime
     *        The time the motor takes to accelerate from 0 to full, in seconds
     * @param fullVel
     *        The maximum motor velocity, in ticks per 100ms
     * @param sensorPhase
     *        The phase of the TalonSRX sensors
     */
    public TalonSRXSimProfile(final TalonSRX talon, final double accelToFullTime, final double fullVel, final boolean sensorPhase) {
        this._talon = talon;
        this._accelToFullTime = accelToFullTime;
        this._fullVel = fullVel;
        this._sensorPhase = sensorPhase;
    }

    /**
     * Runs the simulation profile.
     *
     * This uses very rudimentary physics simulation and exists to allow users to test
     * features of our products in simulation using our examples out of the box.
     * Users may modify this to utilize more accurate physics simulation.
     */
    public void run() {
        final double period = getPeriod();
        final double accelAmount = _fullVel / _accelToFullTime * period / 1000;

        /// DEVICE SPEED SIMULATION

        double outPerc = _talon.getMotorOutputPercent();
        if (_sensorPhase) {
            outPerc *= -1;
        }
        // Calculate theoretical velocity with some randomness
        double theoreticalVel = outPerc * _fullVel * random(0.95, 1);
        // Simulate motor load
        if (theoreticalVel > _vel + accelAmount) {
            _vel += accelAmount;
        }
        else if (theoreticalVel < _vel - accelAmount) {
            _vel -= accelAmount;
        }
        else {
            _vel += 0.9 * (theoreticalVel - _vel);
        }
        _pos += _vel * period / 100;

        /// SET SIM PHYSICS INPUTS

        _talon.getSimCollection().addQuadraturePosition((int)(_vel * period / 100));
        _talon.getSimCollection().setQuadratureVelocity((int)_vel);

        double supplyCurrent = Math.abs(outPerc) * 30 * random(0.95, 1.05);
        double statorCurrent = outPerc == 0 ? 0 : supplyCurrent / Math.abs(outPerc);
        _talon.getSimCollection().setSupplyCurrent(supplyCurrent);
        _talon.getSimCollection().setStatorCurrent(statorCurrent);

        _talon.getSimCollection().setBusVoltage(12 - outPerc * outPerc * 3/4 * random(0.95, 1.05));
    }
}
```

FRC Process to Write JAVA code for a Command Based Robot (July 1, 2021)

Resources

FRC-Big WPI document (1000+ pages)

https://buildmedia.readthedocs.org/media/pdf/frc-docs/latest/frc-docs.pdf

Command Groups: page 866

JAVA DOCS
https://first.wpi.edu/FRC/roborio/release/docs/java/index.html

CTRE – WPI_TalonSRX javadoc
https://www.ctr-electronics.com/downloads/api/java/html/classcom_1_1ctre_1_1phoenix_1_1motorcontrol_1_1can_1_1_w_p_i___talon_s_r_x.html

Phoenix Documentation, CTRE, Apr 03, 2020

https://phoenix-documentation.readthedocs.io/_/downloads/en/latest/pdf/

Page 43. Installing VSCode support for Talon-SRX
Page 76. Troubleshooting CAN Bus issues
Page 106. Talon SRX programming details - 2.17.2 Configuration
Page 115. Talon Inverter and followers - 2.17.5 Open-Loop Features
Page 120. Deadband strategies

Page 124 2.17.6 Reading status signals
Page 128 2.18 Bring Up: Talon FX/SRX Sensors
Page 137 Defining Encoder in software
Page 155 Discussion on WPI_TalonSRX class
Page 158 Motion Magic

Decision to make for Talons:
Lead / Follow
No-Invert / Invert
Brake / Coast
Neutral Deadband
Ramp Rate
Peak/Nominal Outputs
Current Limit

Status that can be read from a Talon SRX:

FYI: Here are the Axis Mappings

```
// Axis control mappings
// Notes:
// - Left and right trigger use axis 3
// - Left trigger range: 0 to 1
// - Right trigger range: 0 to -1).
static public int LAxisX = 6;
static public int LAxisY = 1;
static public int LT = 2;
static public int RT = 3;
static public int RAxisX = 4;
static public int RAxisY = 5;
```

Preparation of Workstation for 2021 VSCode and Simulation

Reference: https://docs.wpilib.org/en/stable/docs/zero-to-robot/step-2/index.html

06/04/2021 03:52 PM 1,294,809,088 ni-frc-2020-game-tools_20.0.1_offline.iso
06/04/2021 03:54 PM 1,777,072,128 WPILib_Windows64-2021.3.1.iso
06/04/2021 08:04 PM 357,807,054 WPILib-VSCode-1.52.1.zip

06/04/2021 03:58 PM 114,191,943 CTRE_Phoenix_Framework_v5.19.4.1.exe
06/04/2021 04:00 PM 90,756,726 navx-mxp.zip

Process:
Install 2021 FRC Game Tools
Install WPILib

Install CTRE Phoenix Framework
Unzip and install NAVX