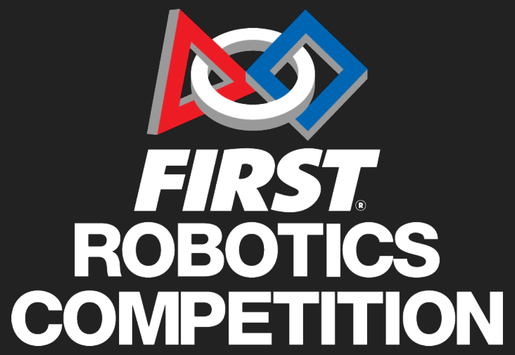


FRC JAVA PROGRAMMING

CREATED BY TAYLER UVA
FRC TEAM 3255 - THE SUPERNURDS

GITHUB.COM/TAYLERUVA
SUPERNURDS.COM

Updated for the 2019 Season

TABLE OF CONTENTS

- Learning the basics:
  - Part A: The roboRIO
  - Part B: Sensors
  - Part C: Java Programming Basics
  - Part D: WPILib Programming Basics
- Development Environment Setup
  - Part E: Software installation and setup
  - Part F: Imaging the roboRIO
  - Part G: Setting-up the Robot Radio
- Programming for an FRC Robot:
  - Part 1: Creating a Basic Driving Robot
  - Part 2: Using Pneumatics
  - Part 3: Using Sensors and Switches
  - Part 4: Using SmartDashboard
  - Part 5: Using RobotPreferences
  - Part 6: Creating an Autonomous Command
  - Part 7: Getting started with PID

Note: This tutorial builds on itself and will not repeat the same tasks for each new section.
Please read previous sections if there is a part you are confused with.

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

PARTS A–D:
BASICS OF WPILIB AND PROGRAMMING

LEARNING THE BASICS

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

PART A:
THE BRAINS OF THE BOT

THE ROBORIO

Table of Contents

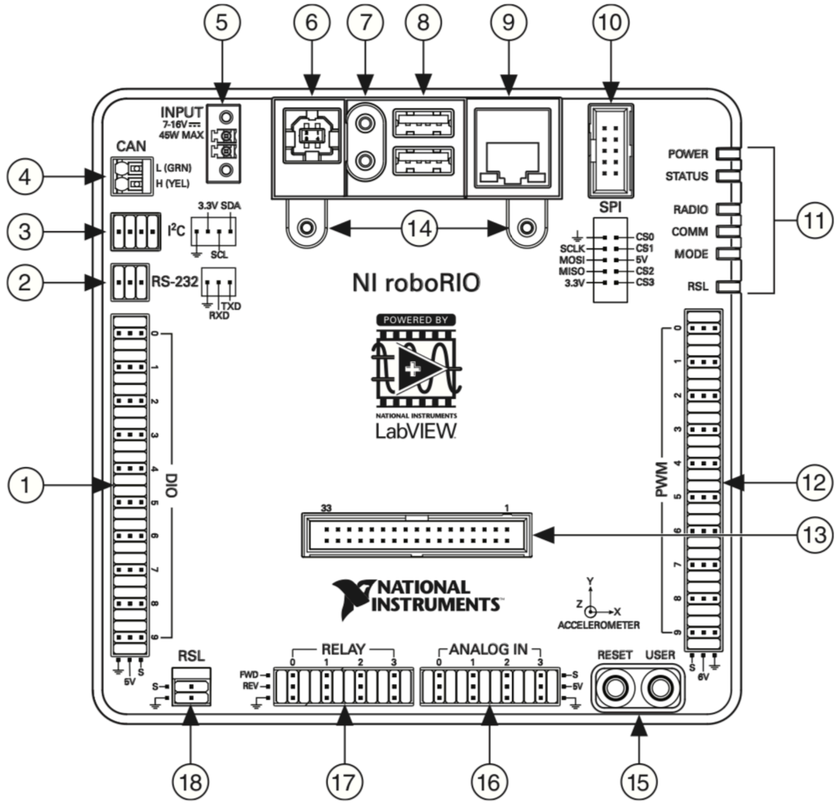
THE ROBORIO

- The roboRIO is the brain of an FRC robot.
- It is the main processing unit and is where the code is stored and run.
- It is very similar to something like a Raspberry Pi, it's a mini computer!
- The roboRIO can connect to many different devices such as **motor controllers**, **servos**, and **sensors** through its various interface connections such as:
  - **Digital I/O, PWM, CAN Bus, Ethernet, USB, MXP**

Table of Contents

THE ROBORIO IO

- **Digital IO** (DIO) used for sensors and switches
- **PWM** used for motor controllers and servos
- **CAN** used for motor controllers and sensors
- **MXP** used for functionality expansion
- Check the roboRIO user manual for more details



1	Digital input and output (DIO) port	10	Serial peripheral interface bus (SPI) port
2	RS-232 port	11	LEDs
3	I²C port	12	Pulse-width modulation (PWM) port
4	CAN port	13	myRIO Expansion Port (MXP)
5	Power connector	14	MXP retention mount
6	USB Device port	15	User and Reset buttons
7	USB Host retention mount	16	Analog input (AI) port
8	USB Host ports	17	Relay port
9	Ethernet port	18	Robot signal light (RSL) port

Table of Contents

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

PART B:
HOW DOES THE ROBOT SEE?

SENSORS

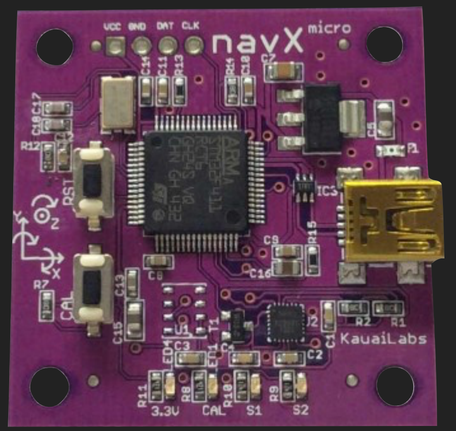
Table of Contents

SOME TYPES OF SENSORS

- **Limit Switches** - detects contact
- **Camera** - provides sight
- **Encoders** - measures rotational or linear motion
- **Ultrasonic** - measures distances
- **Gyroscope** - measures orientation
- **Processed Vision** - measures target's distance, angle, and offset from robot
  - See more info on NURDVision
- For more info on sensors see: High Tech High Top Hat Technicians - Electrical Tutorial

Limit Switch

Grayhill brand Quadrature Encoder



Kauai Labs navX
Gryo / Accelerometer

Table of Contents

PART C:
LEARNING WHATS WHAT

JAVA PROGRAMMING BASICS

Table of Contents

OVERVIEW

- Objects, variables, and classes (in Java) make up our programs. We define, modify, use these variables and objects to make our programs run.
- Programs use key words to define characteristics of variables or objects.
- Basic keywords:
  - `public` - an object accessible by other classes (files)
  - `private` - an object only accessible by its containing class (file).
  - `protected` - like private but can be seen by subclasses
  - `return` - value to return or give back after method execution (run).
  - `void` - a method that returns no value
- **IMPORTANT NOTE:** Java is case sensitive, meaning capitalization matters!

CLASSES

- Classes are the files that contain our programming
- A program can be made up of one class but can also be made up of many classes
  - All programs run a main class that can optionally load additional classes either directly or indirectly (i.e. main loads class1, class1 loads class2)
- Classes are made up of variables and methods and are often used to separate and organize your code.
- Classes can also **call** (use) variables or methods of other classes if those have been set to public.

CONSTRUCTORS

- Classes can also have a **constructor** which is a special type of **method** that has the **same name (case sensitive)** as the class file
  - Constructors are always called when the class is loaded into the program for the first time. This is often the only time they are called.
  - Constructors are called when trying to access the class in other files.
  - They can be called again if the class is programmed to be unloaded (destroyed) and reloaded.
  - Calls to methods, and assignment of values, within the constructor will run as soon as the class is called (loaded) in the code.

VARIABLES

- Variables are objects that contain data, they are characterized by data types
- Variables are assigned names and data types on creation
  - Names can be anything with the exception of pre-existing keywords such as `public` or `int`
- Data types define what type of data is being stored in the variables:
  - `int` - integers (whole numbers)
  - `double` - double precision floating point (fractional/decimal values)
  - `boolean` - true or false (true = 1 or false = 0) values.
  - `string` - text values contained in parentheses

VARIABLES

- Examples:
  - `int sum;` - A variable that can hold whole number values
  - `bool isFull = true;` - A variable can either hold a true or false value and is being assigned a true value;
- Most non-static variables can have their values assigned or assigned at any point elsewhere in your program

METHODS

- Methods, also known as functions, can be thought of as subprograms or routines that run inside of your main program.
- Methods are used when you want to run the same code multiple times. Copying and pasting code is **BAD!** Use methods instead!
- Methods are also useful to access only certain parts or functions of another class.
- Methods can also have their own variables (local) or use variables available throughout the whole class (global variables), this will be explained more in the scope section.
- Methods can call (use) other methods, even multiple times.

- Example:

```
int value;
void increment(){
   value++;
}
```

SCOPE

- When creating a variable, where you create it matters. This is known as the scope of a variable.
- The scope is where a variable can be seen within a class
  - A variable created in a method can only be seen in that method. This is a **local** variable.
  - A variable created outside a method can be seen in all methods of that class (file). This is a **global** variable.
    - It is good practice to put them all at the top before your first method.

- Example of Local Variable

```
public void
testMethod() {
    int example = 12;
  }
```

- Example of a Global Variable:

```
int example = 1;
public void testMethod(){

}
```

PARAMETERS

- Parameters are variables that are passed (sent) to a method for it to use.
- You can pass more than one parameter but order matters when calling the method.

- Example of the method:

```
double half(int num1){
double multiplier = 0.5;
return num1*multiplier;
}
```

- Example of the method being called (used) in a class:

```
int newNumber = half(12);
```

COMMENTS

- Comments are a programmer-readable explanation or annotation in the source code of a program.
- Comments do not affect what the code does.
- Comments are often used to leave notes or explanations of what methods or classes are doing so that it is easier to understand the code.
- Example:

```
//This is a comment
```

CONVENTIONS

- There are also many different conventions when programming, this ensures that programs are readable between different people.
- A common naming convention:
  - Programming is often done in CamelCase or lowerCamelCase
    - Instead of adding spaces, capitalize the first letter of each word
    - For example:
      - ThreeMotorDrive, driveForward, setSpeed
- There are other naming conventions, but for this tutorial we will use the camel cases

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

PART D:

MAKING FRC PROGRAMMING EASY

WPILIB PROGRAMMING BASICS

Table of Contents

WHAT IS WPILIB?

- The WPI Robotics library (WPILib) is a set of software classes that interfaces with the hardware and software in your FRC RoboRIO.
  - There are classes to handle sensors, motor speed controllers, the driver station, and a number of other utility functions.
- Documentation is available here: http://first.wpi.edu/FRC/roborio/release/docs/java
- WPILib adds those sensors and controllers as additional data types (like **int** or **double**) and classes.
  - Examples: **Talon, Solenoid, Encoder**…

COMMAND BASED ROBOT

- For our programming tutorial we will be creating a Command based robot
- Command Based Robots are much like Legos, with very basic pieces you can make something **simple** like a house or **complicated** like a replica Star Wars Millennium Falcon
- A command based robot is broken down into **subsystem** classes and **command** classes.
- In the code, a command based robot is made up of 3 **packages** (**folders**) labeled robot, commands, and subsystems
- There are other types of robots but we will use Command Based

SUBSYSTEMS

- A **subsystem** is a special template class made by FRC.
- In robotics, subsystems are sections of the whole robot.
  - For example every FRC robot has a **Drivetrain** subsystem which is what controls the robot's driving both physically and programmatically.
    - To avoid confusion between software and mechanical teams, subsystems should be called the same thing. If we have a ball intake system, we will both call it **Intake** or **Collector.**
  - Subsystems of a robot can contain parts to control or read data from.
    - The **Drivetrain** subsystem could contain **motor controllers** and **encoders** both physically and programmatically.
- Using a dog as an example: the **legs**, **tail**, and **head** are **subsystems**.
  - The **head** subsystem has the parts: **eyes**, **ears**, and **nose**.

SUBSYSTEMS

- When programming subsystems we use variables and methods to tell our subsystem what it has and what it is capable of or should do.
  - These variables will be the parts in the subsystem
  - These methods will define what those parts are capable of.
- Using a dog **head** subsystem as an example:
  - A some variables (parts) would be: **leftEye, rightEye, nose, leftEar, rightEar.**
  - Some example methods would be **closeEyes** or **openEyes** since these are things the dog are capable of.
    - These methods would use both the **leftEye** and **rightEye** and close them.

```
//This method closes the dog eyes

public void closeEyes(){
 leftEye.close();
 rightEye.close();
}
```

SUBSYSTEMS

- A robot example of a **Drivetrain** subsystem would have **leftMotor,** and **rightMotor** as variables and **setSpeed** as a method telling it how to set the speed of those motor controllers.

- Having the **setSpeed** method tells our program that out **Drivetrain** subsystem can set its speed.

```
//This method sets the speed of the drivetrain

public void setSpeed(double speed){
 leftMotor.set(speed);
 rightMotor.set(speed);
}
```

COMMANDS

- A **command** is a special template class (file) made by FRC.
- In robotics, commands are actions you want a robot to do (just like a real life command).
  - A **command** is an action a **subsystem(s)** performs.
  - For example you may want your robot to drive full speed forward so you make a command class called **DriveForward**.
    - Since a robot uses a **Drivetrain** subsystem to control its motors, this command would call our previously created **setSpeed** method from that subsystem.
- **IMPORTANT: Subsystems** define what the robot is made of and what it can do while **commands** actually tell the robot to do those things
- Using a dog as an example we can tell the dog to blink by creating a **BlinkEyes** command
  - The command would call the method, **closeEyes()** then the method **openEyes()**

COMMANDS

- When programming **commands** we call methods that we had previously created in our **subsystems**
  - When we run our command through autonomous or pushing a button, it will in turn run our subsystem methods.
- Using a dog **blinking** as an example:
  - Our command would be called **BlinkEyes**
  - The command would access the **head** subsystem and call both the **closeEyes()** and **openEyes()** methods we created

```
//This command will
continuously run the two
methods in execute

protected void execute()
{
 dog.head.closeEyes();
 dog.head.openEyes();
}
```

COMMANDS

- A robot example of a **DriveForward** command would call (use) the **setSpeed** methods that we created in the **Drivetrain** subsystem
- **DriveForward,** when executed, will tell our robot to drive forward using the **Drivetrain** subsystem

```
//This command tells the robot to drive forward
full speed

protected void initialize(){
 robot.drivetrain.setSpeed(1.0);
}
```

COMMANDS

- The template for FRC commands actually come with some pre-defined methods that have special properties for FRC robots, they are:
  - `initialize()` - Methods in here are called just before this Command runs the first time.
  - `execute()` - Methods in here are called repeatedly when this Command is scheduled to run
  - `isFinished()` - When this returns true, the Command stops running execute()
  - `end()` - Methods in here are called once after isFinished returns true
  - `interrupted()` - Methods in here are called when another command which requires one or more of the same subsystems is scheduled to run
    - It is good practice to call `end()` here

OVERVIEW OF EXECUTION

- In FRC programming our main class is **Robot.java** and all other classes (command files and subsystem files) must be loaded from **Robot.java** either directly or indirectly (i.e. **Robot.java** loads **OI.java**, **OI.java** loads **DriveForward.java**).
- All **subsystem** files must be added to **Robot.java**'s auto-created `robotInit()` method.
  - This loads our **subsystems** into the code and allow its public methods to be useable by other files such as commands later by typing `Robot.nameOfSubsystem.desiredMethod();`

NEW PROJECT FILES

- When creating a new command based robot project, the following classes (files) will be created:
  - **Robot.java** - The main class of the robot which is run when a robot boots up.
  - **OI.java** - This class binds our **commands** to a physical operator interface such as a joystick or controller.
    - This file is already in `robotInit()` by default so classes called here will also be loaded by the program
  - **RobotMap.java** - This class is used to hold all the ports or ID numbers of sensors or devices connected to the robot and assign them a variable name.
    - This provides flexibility changing wiring, makes checking the wiring easier and significantly reduces the number of magic numbers floating around.
  - **ExampleSubsystem.java** and **ExampleCommand.java** are auto-created examples.

SUMMARY

- Command based robots are broken down into **subsystems** and **commands**
- **Subsystems** define what the robot is made of and what it can do while **commands** actually tell the robot to do those things
- All classes must directly or indirectly connect to **Robot.java.**
  - All **Subsystems** must be added to **Robot.java's** `robotInit()`
- **RobotMap.java** holds port numbers and IDs accessible throughout the program by typing: `RobotMap.NameOfMotor()`
- **OI.java** connects our commands to physical controllers

PARTS E–G:
SOFTWARE INSTALLATION AND SETUP!

DEVELOPMENT ENVIRONMENT SETUP

Visual Studio Code

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

PART E:
LETS GET STARTED

INSTALLING NECESSARY SOFTWARE

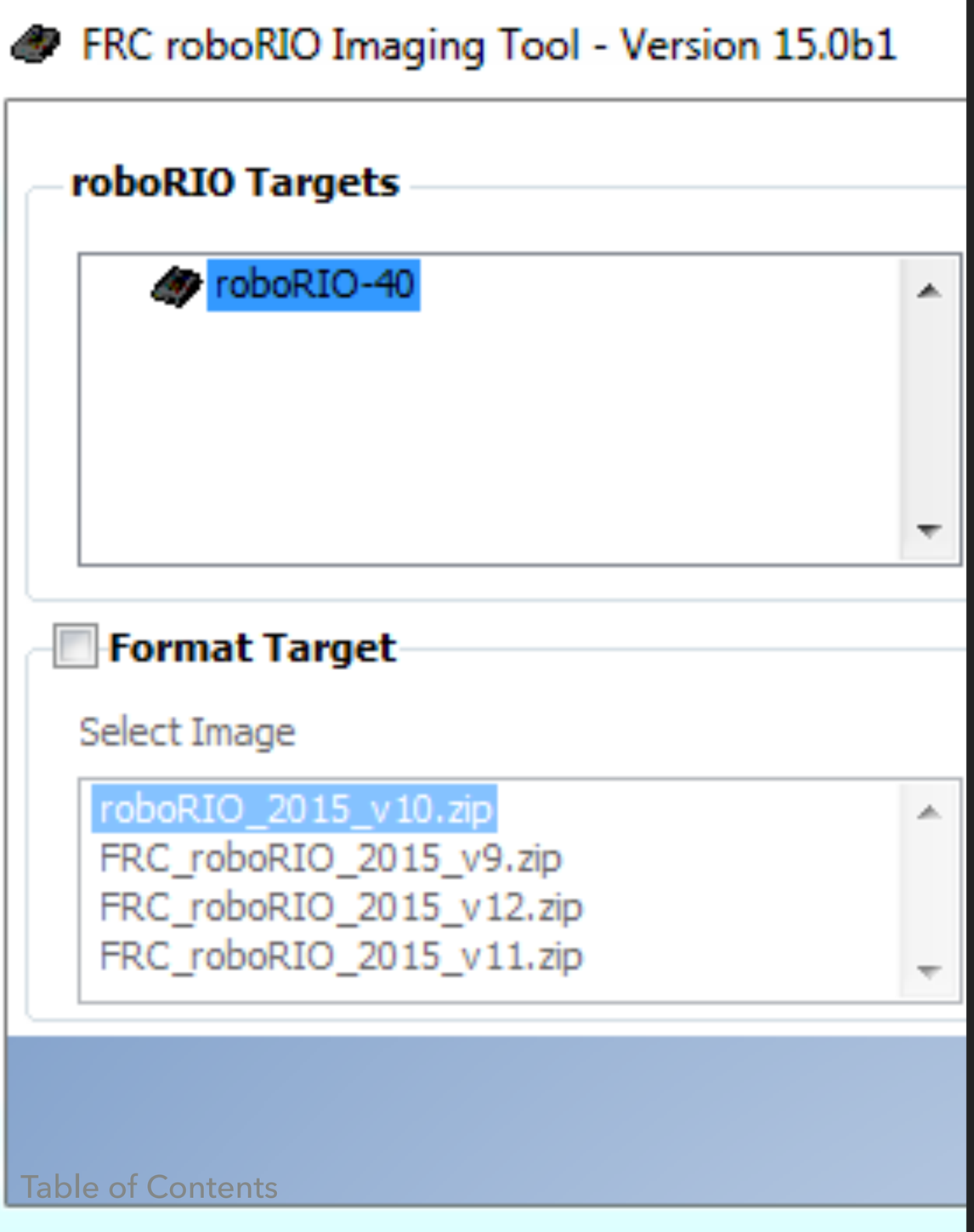
Table of Contents

THIS SECTION NEEDS TO BE CONFIRMED FOR 2019
PLEASE FOLLOW THESE INSTRUCTIONS IN THE MEANTIME:
HTTPS://WPILIB.SCREENSTEPSLIVE.COM/S/CURRENTCS/M/79833/L/932382-INSTALLING-VS-CODE

THIS SECTION NEEDS TO BE CONFIRMED FOR 2019
PLEASE FOLLOW THESE INSTRUCTIONS IN THE MEANTIME:

THE DRIVER STATION SOFTWARE IS NOT REQUIRED FOR PROGRAMMING THE ROBOT, ONLY TESTING IT. IT CAN BE INSTALLED ON A DIFFERENT COMPUTER WITH THESE INSTRUCTIONS:

HTTP://WPILIB.SCREENSTEPSLIVE.COM/S/CURRENTCS/M/GETTING_STARTED/L/599670-INSTALLING-THE-FRC-UPDATE-SUITE-ALL-LANGUAGES



SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

PART F:
CONFIGURING THE BRAIN!

IMAGING THE ROBORIO

IMPORTANT:

THIS SECTION REQUIRES A WINDOWS COMPUTER WITH THE FRC UPDATE SUITE

You also must have the **roboRIO** power properly wired to the **Power Distribution Panel** as described **here.**

Note: If the computer is not going to be used as a Driverstation laptop (or for imaging the roboRIO) only Visual Studio Code needs to be installed. However **AT LEAST ONE** computer must complete all these steps to have a functional, drivable robot.

IMAGING THE ROBORIO – TEXT INSTRUCTIONS

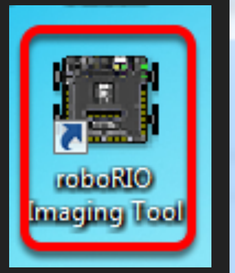
1. Connect the **roboRIO** to your computer with a **USB Type A** male (standard PC end) to **Type B** male cable (square with 2 cut corners)
   - This cable is most commonly found as a printer USB cable.
2. Open the roboRIO Imaging Tool
   1. Make sure the roboRIO is selected in the top left pane
   2. Enter your team number in the box in the top right
   3. Make sure the Disable RT Startup App box is unchecked in the bottom right
   4. Check the box to Format Target and select the latest image version in the box.
   5. Click Reformat to begin the imaging process.
3. Formatting will take 3-10 minutes and display a dialog when completed.
4. Click OK and reboot the roboRIO using the **Reset** button on the roboRIO or a power cycle.

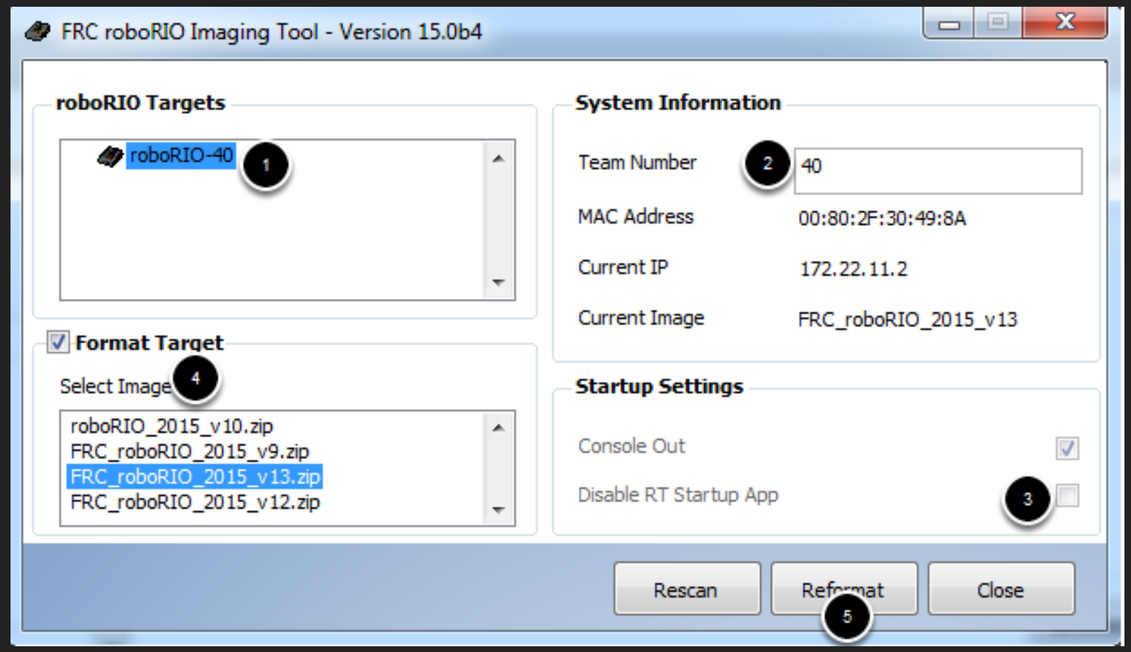
For more detailed information see: Imaging your roboRIO

IMAGING THE ROBORIO – VISUAL INSTRUCTIONS

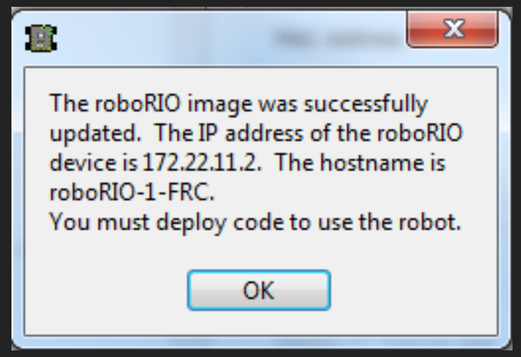
1.

2.





3.



For more detailed information see: Imaging your roboRIO

IMPORTANT:

IF YOU ARE USING DEVICES ON THE CAN BUS:

You will need to use the **Phoenix Lifeboat** tool from CTRE to install the webdash plugin after imaging the roboRIO.

This will allow you to modify CAN IDs of devices. Very important if you are using Talon SRXs.

Note: If the computer is not going to be used as a Driverstation laptop (or for imaging the roboRIO) only Visual Studio Code needs to be installed. However **AT LEAST ONE** computer must complete all these steps to have a functional, drivable robot.

This section is only necessary if you are using CAN Devices.

If you are not, click here to skip this section.

This step must be repeated after re-imaging a roboRIO

MAKING CAN DEVICE IDS EDITABLE!

INSTALLING THE CTRE WEBDASH PLUGIN

INSTALLING CTRE FRAMEWORK – TEXT INSTRUCTIONS

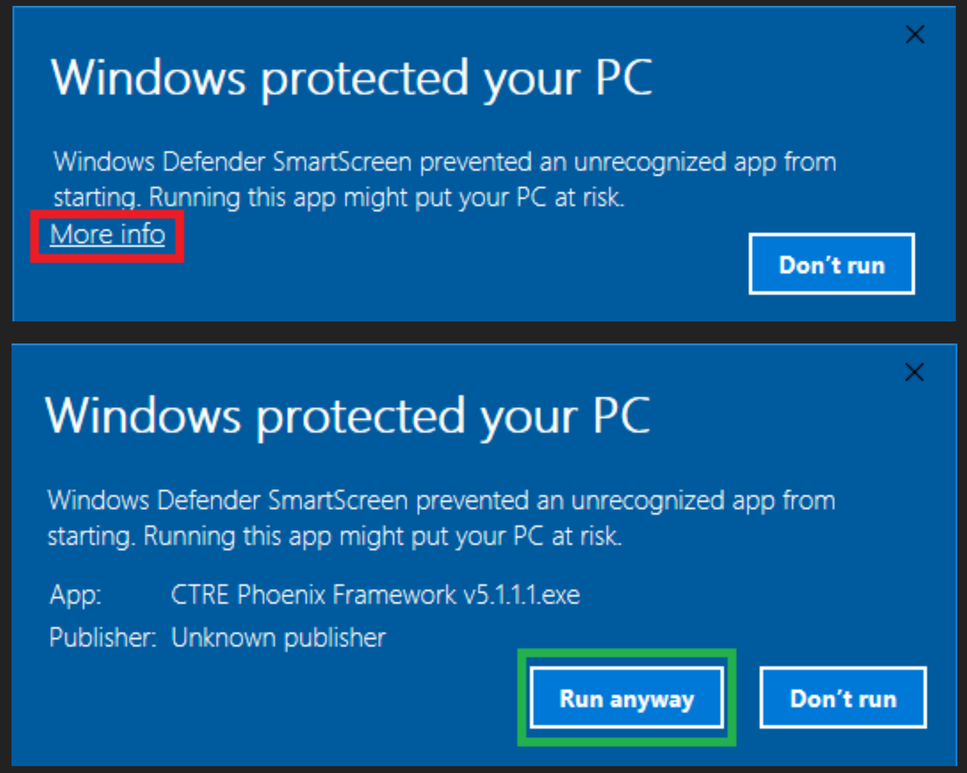
1. Download the CTRE Phoenix Framework Installer
2. Run the installer as an admin (right click and select run as administrator and enter your admin password)
    1. If Windows prompts with an "unpublished" dialog box, press "More Info" and "Allow App to Run".
3. A setup dialog will appear, click next
    1. Uncheck **LabVIEW** and **HERO C#**

For more detailed information see: Installing Pheonix Framework

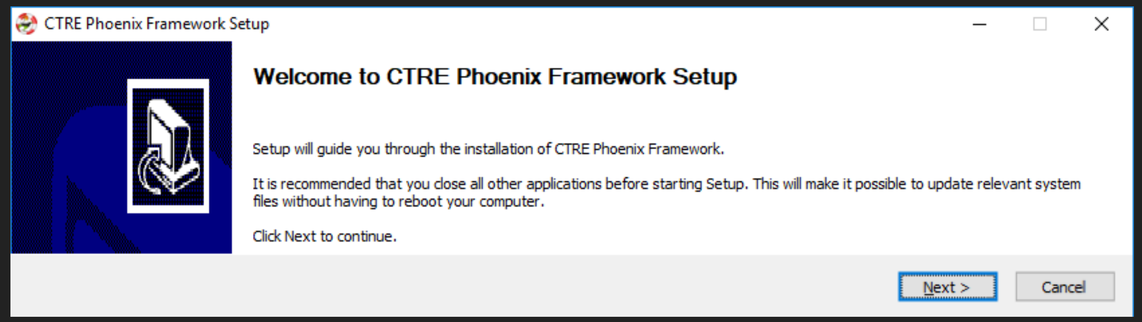
INSTALLING CTRE FRAMEWORK – VISUAL INSTRUCTIONS

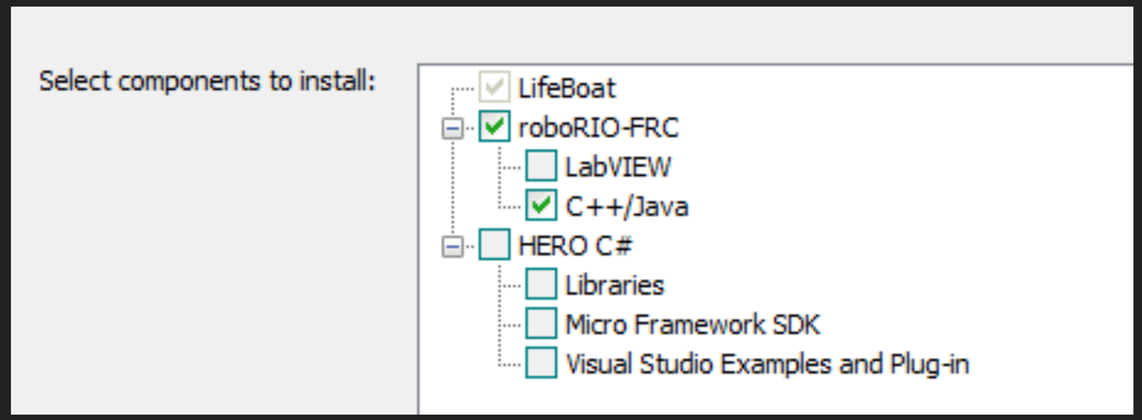
1. Installer | CTRE Phoenix Framework Installer 5.7.1.0 (.zip)

2.



3.





For more detailed information see: Installing Pheonix Framework

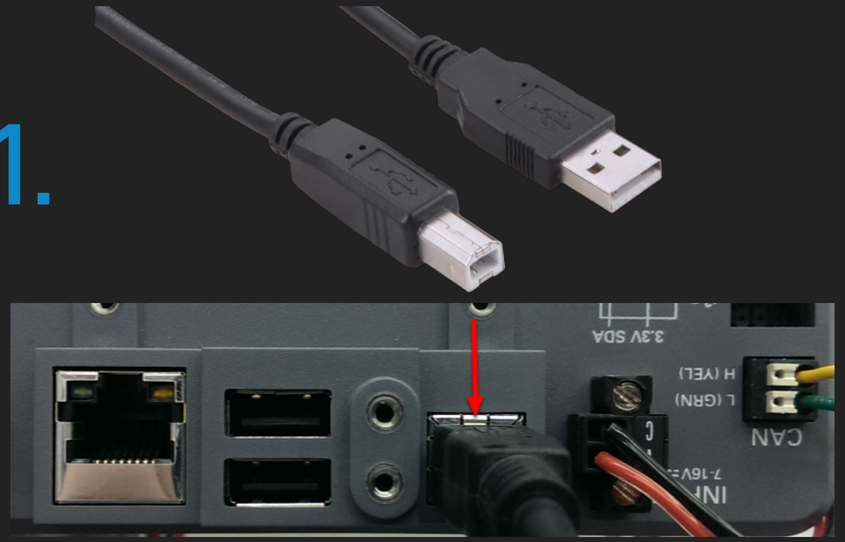
INSTALLING THE WEB PLUGIN – TEXT INSTRUCTIONS

1. Plug in the same USB cable used for imaging the roboRIO into the computer and the roboRIO
2. Open the **Phoenix LifeBoat** utility
3. Select the **roboRIO upgrade** tab
4. Select **Update RIO Web-based Config**

For more detailed information see: Installing Pheonix Framework

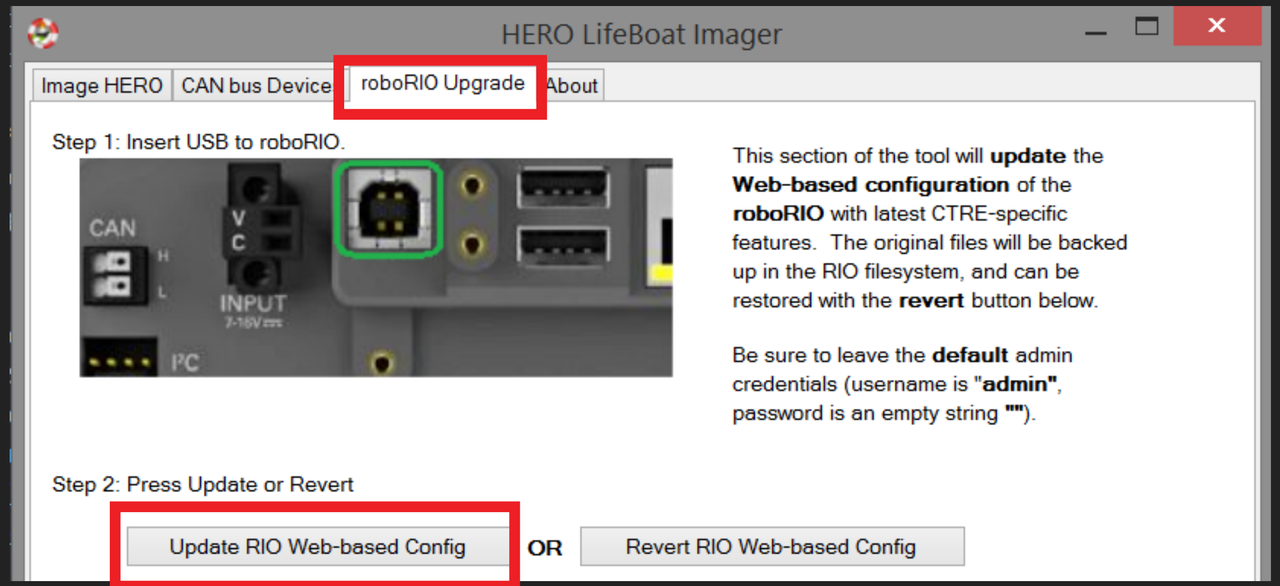
INSTALLING THE WEB PLUGIN – VISUAL INSTRUCTIONS

1.



2.

3.



4.

For more detailed information see: Imaging your roboRIO

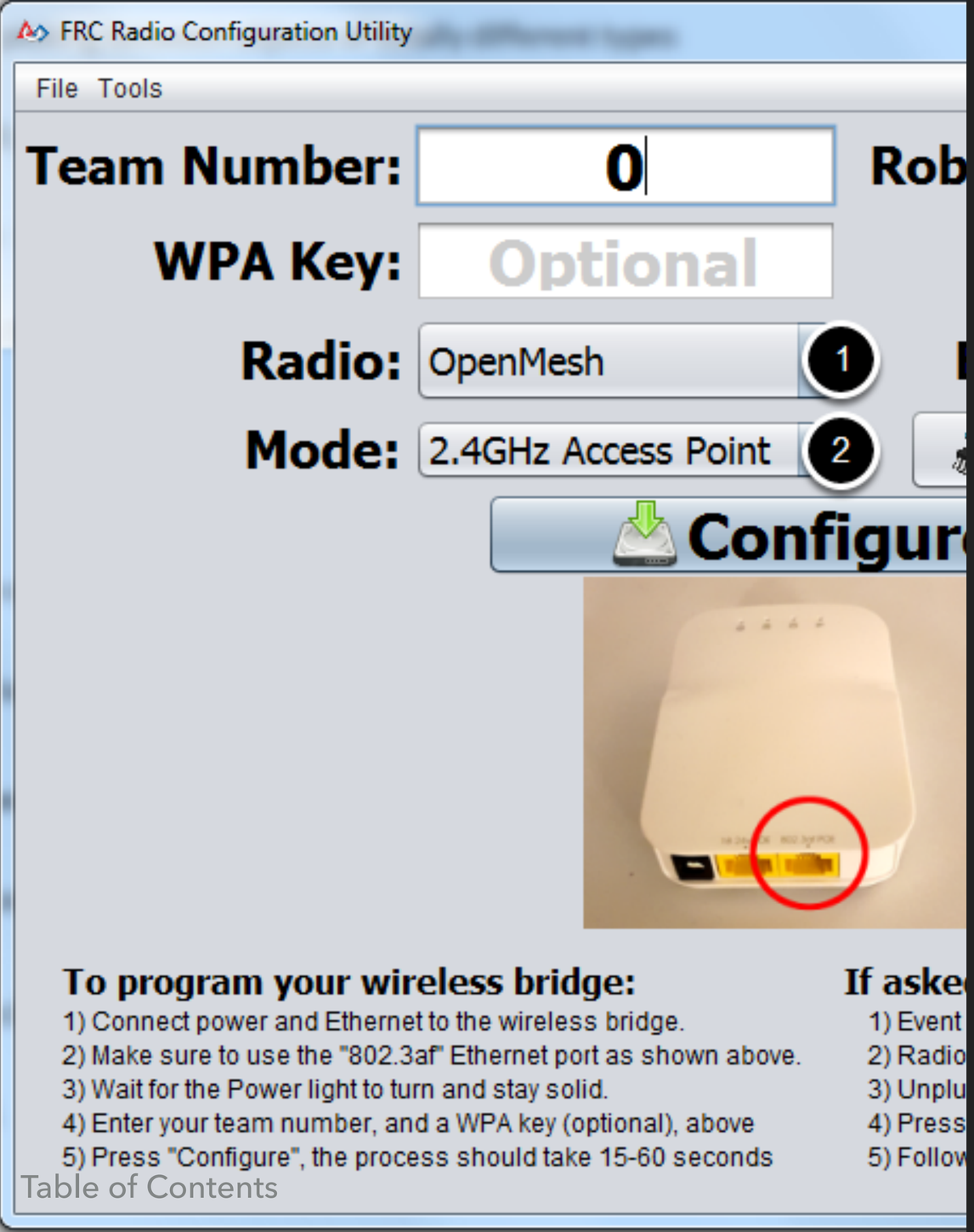


Table of Contents

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

PART G:
GOING WIRELESS!

SETTING-UP THE ROBOT RADIO

THIS SECTION NEEDS TO BE CONFIRMED FOR 2019
PLEASE FOLLOW THESE INSTRUCTIONS IN THE MEANTIME:
HTTP://WPILIB.SCREENSTEPSLIVE.COM/S/CURRENTCS/M/GETTING_STARTED/L/144986-PROGRAMMING-YOUR-RADIO

PARTS 1–7:
PROGRAMMING IN JAVA WITH WPILIB

PROGRAMMING FOR AN FRC ROBOT

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

PART 1:
LET'S GET MOVING

CREATING A BASIC DRIVING ROBOT

Table of Contents

IN THIS SECTION WE WILL BE GOING OVER:

- Creating new subsystems, in this case the basic Drivetrain subsystem
- Adding subsystems to Robot.java, the main file that the robot runs and all other files must be in some way connected to.
- Creating joysticks and using their axes to control the robot.
- Creating and using DifferentialDrive and ArcadeDrive to make controlling the robot's motors an easy process.
- Deploying code to the roboRIO

WHAT IS IN THIS ROBOT?

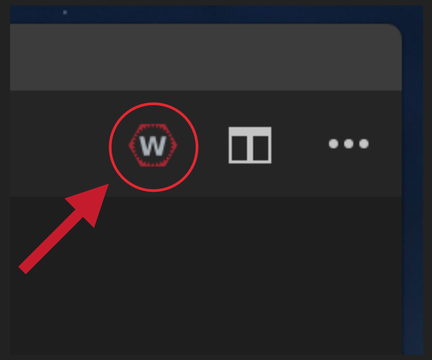
CREATING THE DRIVETRAIN SUBSYSTEM

CREATING A NEW PROJECT – TEXT INSTRUCTIONS

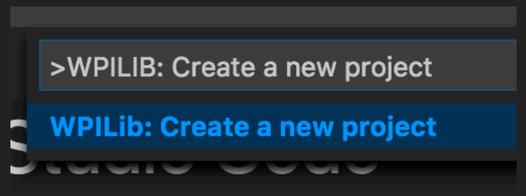
- Select the W icon from the tab bar or use the shortcut by holding down **Ctrl+Shift+P** at the same time. (Replace ctrl with cmd on macOS)
- Type and hit enter or select **WPILib: Create a new project**
- Click **Select a Project Type** and choose **Template**
- Click **Select a Language** and choose **Java**
- Click **Select a project base** and choose **Command Robot**
- Click **Select a new project folder** and choose where on your computer you would like to store the program
- **Enter a project name** in the text field labeled as such
- **Enter your team number** in the text field labeled as such
- Select **Generate Project**
- When prompted **"Would you like to open the folder?"**, select **Yes (Current Window)**

CREATING A NEW PROJECT – VISUAL INSTRUCTIONS

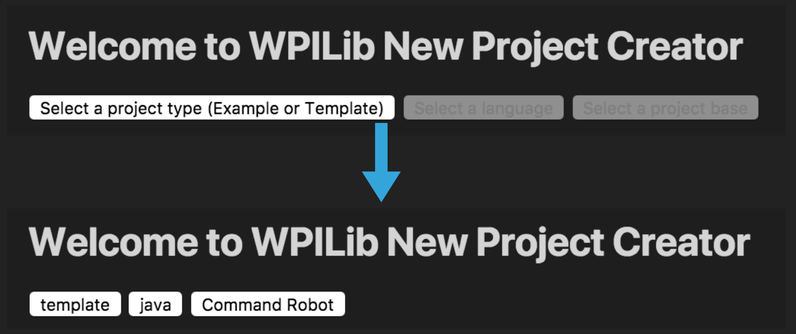
1.



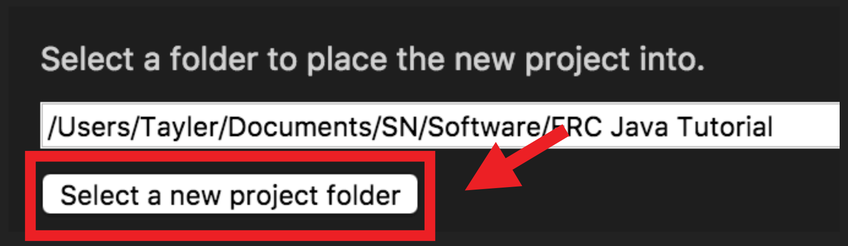
2.



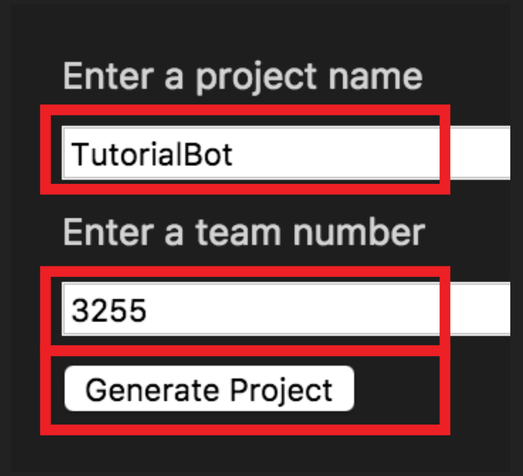
3.



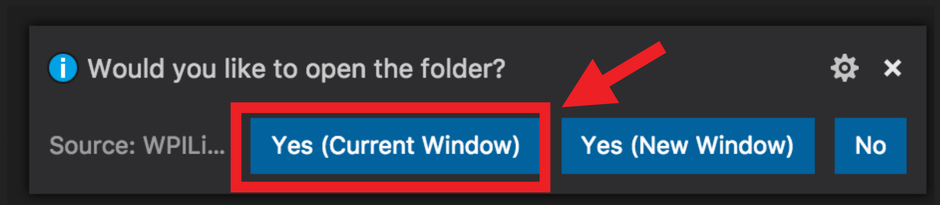
4.



5.



6.

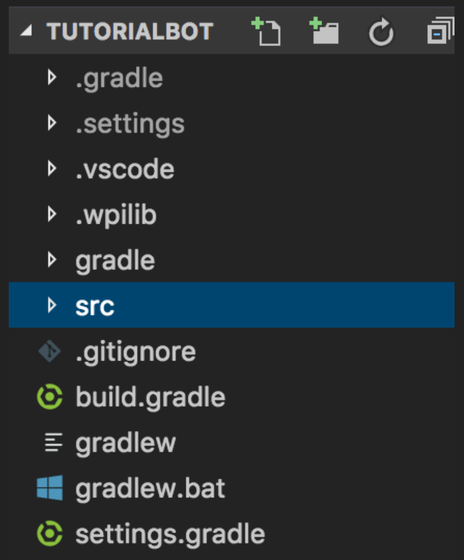


CREATING THE DRIVETRAIN SUBSYSTEM – TEXT INSTRUCTIONS

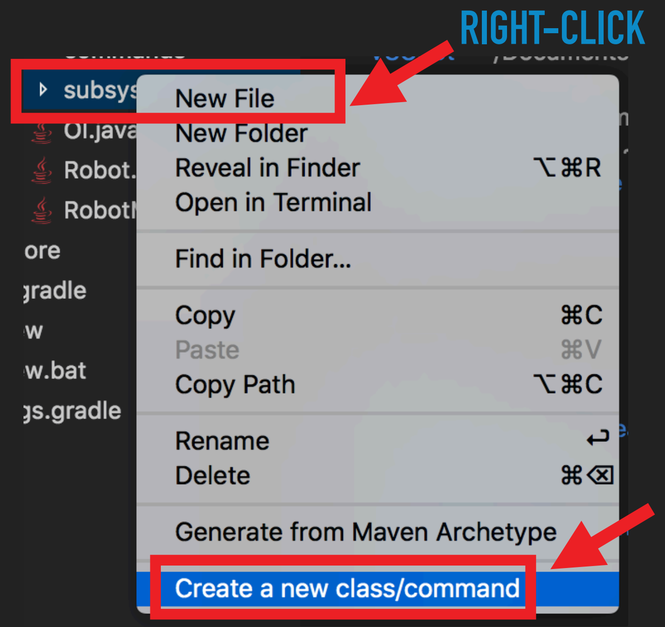
- Double click on the **src** folder to expand it. Do the same for **subsystems**
- Right click on **subsystems** and select **Create a new class/ command.**
- Select **Subsystem** and type **Drivetrain** for the name and hit enter on your keyboard.
- Click on the newly create **Drivetrain.java**

CREATING THE DRIVETRAIN SUBSYSTEM – VISUAL INSTRUCTIONS

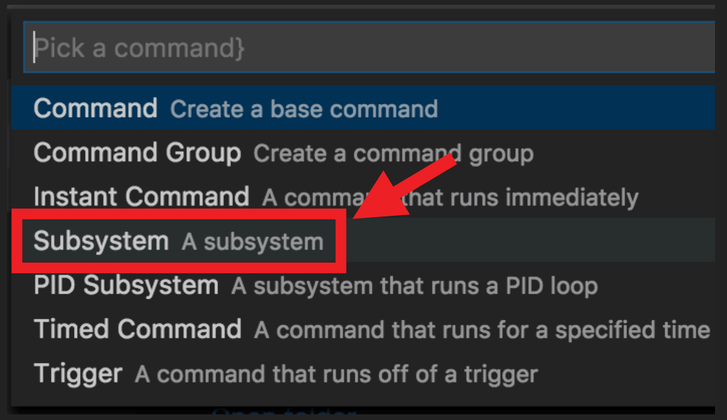
1.



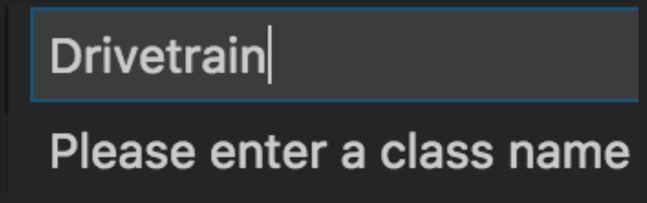
2.



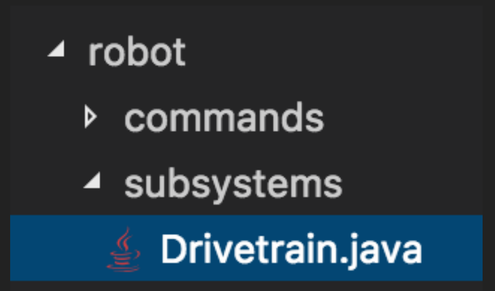
3.



4.



5.



WHAT'S IN THE SUBSYSTEM?

- Newly created subsystems are empty with the exception of the initDefaultCommand.
- We will create a constructor ourselves later.
- initDefaultCommand - a command that will run automatically every time the subsystem is called.
  - When another command that requires the same subsystem is called, the initDefaultCommand will stop and restart after the new command has finished.
    - This process calls the interrupted method of the command being called initDefaultCommand

WHAT WILL BE ADDED TO THE DRIVETRAIN?

- In the Drivetrain class we will tell the subsystem what type of components it will be using.
- A Drivetrain needs motor controllers. In our case we will use 4 Talon SRs (a brand of controller for motors).
  - You could use other motor controllers such as Victor SPs or Talon SRXs but we will be using Talon SRs
    - If you are, replace **Talon** with **TalonSRX**, **Victor**, or **VictorSP** in the code you write depending on the type you use.
  - You can use 2 motors (left and right), but for this tutorial we will use 4.

FOLLOW ALONG!

LETS START CODING!

- RED BULLETS ARE THE STEPS TO FOLLOW!

IMPORTANT: USING VISUAL STUDIO CODE'S LIGHTBULB

When auto complete is available, click *enter* on the correct completion to auto import classes.

If this is not done an error will occur denoted by an underline on what you just typed. To fix this, click on the error and click the ***lightbulb*** *that pops up. Then click import.*

From this point on this tutorial will assume you are doing this on your own so if errors occur, click the lightbulb and see if it suggest importing something.

The ***lightbulb*** *can also be used to make programming easier by auto creating things for us. This tutorial will go over that in future sections*

CREATING THE TALON VARIABLES – TEXT INSTRUCTIONS

- Let's create 4 global variables of data type **Talon** and name them: leftFrontTalon, rightFrontTalon, leftBackTalon, rightBackTalon
  - To get started type the word **Talon** followed by the name
    - i.e. Talon leftFrontTalon;
  - These will eventually hold the object values for Talons and their port numbers.
- Next assign their values to **null (LOWERCASE).**
  - **null** means they have a value of nothing and are currently empty.
    - We do this to make sure it is empty at this point.
    - When we assign the variables a value, we will be getting their port number out of RobotMap
      - This means we cannot assign them at the global level

CREATING THE TALON VARIABLES – VISUAL INSTRUCTIONS

IF NO ERRORS OCCURRED IT SHOULD LOOK LIKE THIS:

1.

```
import edu.wpi.first.wpilibj.Talon;
import edu.wpi.first.wpilibj.command.Subsystem;
import frc.robot.RobotMap;

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

  Talon leftFrontTalon = null;
  Talon leftBackTalon = null;
  Talon rightFrontTalon = null;
  Talon rightBackTalon = null;
```

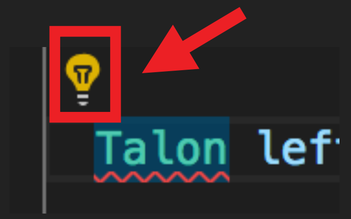
CREATING THE TALON VARIABLES – VISUAL INSTRUCTIONS

IF ERRORS OCCUR USE THE LIGHTBULB FIX BY CLICKING ON THE BULB:

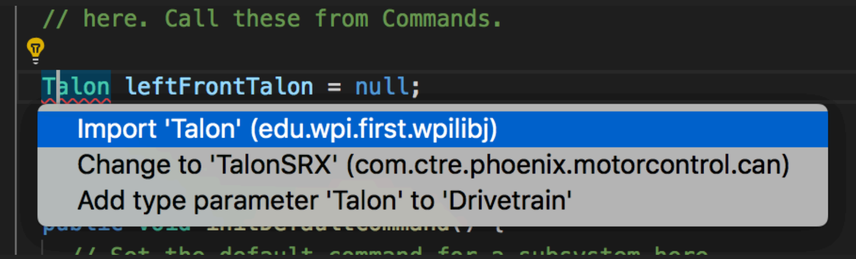
E1.

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

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

  Talon leftFrontTalon = null;
```

E2.



E3.



E4.

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

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

  Talon leftFrontTalon = null;
```

CREATING A CONSTRUCTOR

- Next we will create a constructor for the Drivetrain class (similar to a method). (See image)
  - The constructor is where we will assign values to our talon variables.
- Now that we have created the Talons we must initialize them and tell them what port on the roboRIO they are on.
- Let's initialize (set value of) leftFrontTalon to 'new Talon(0)'. This initializes a new talon, leftFrontTalon, in a new piece of memory and states it is on port 0.
  - This should be done within the constructor 'Drivetrain()'
  - This calls the constructor Talon(int) in the Talon class. The constructor Talon(int) takes a variable of type int. In this case the int (integer) refers to the port number on the roboRIO.

```
public class Drivetrain extends Subsystem {

    //Motor Controllers
    Talon leftFrontTalon = null;
    Talon leftBackTalon = null;
    Talon rightFrontTalon = null;
    Talon rightBackTalon = null;

    public Drivetrain() {
        //Talons
        leftFrontTalon = new Talon(0);
    }
```

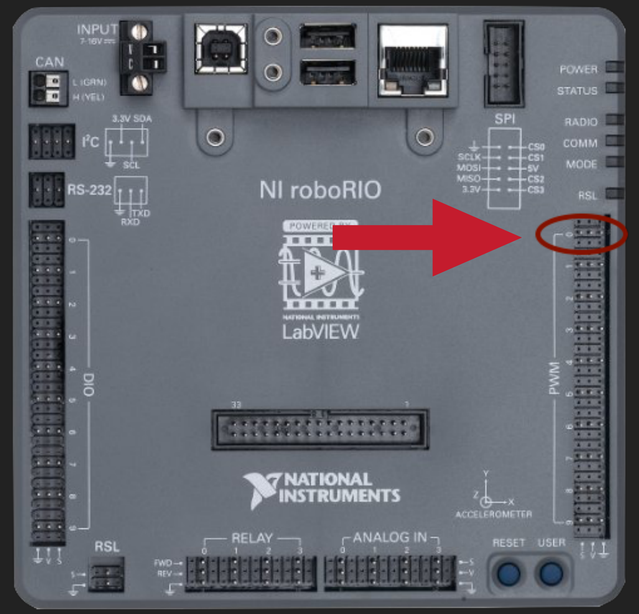


Table of Contents

USING ROBOTMAP – TEXT INSTRUCTIONS

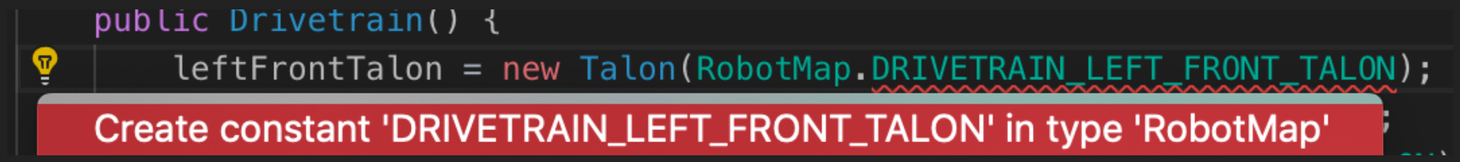
- Since each subsystem has its own components with their own ports, it is easy to lose track of which ports are being used and for what. To counter this you can use a class called **RobotMap** to hold all these values in a single location.
- To use RobotMap, instead of putting '0' for the port on the Talon type: 'RobotMap.DRIVETRAIN_LEFT_FRONT_TALON'.
  - Names should follow the pattern SUBSYSTEM_NAME_OF_COMPONENT
  - The name is all caps since it is a **constant** (can only be changed by the user) variable. This is the naming convention for constants.

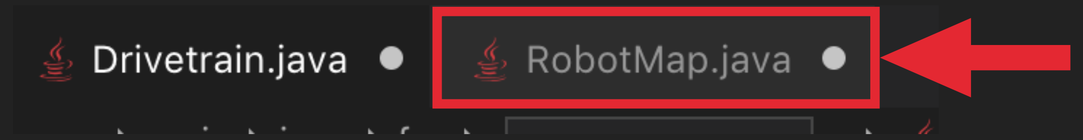
- Click on the **lightbulb** and select "create constant…" then click on RobotMap.java tab that just popped up.
  - Change the 0 to the actual port on your roboRIO **(IF INCORRECT YOU COULD BRAKE YOUR ROBOT!)**
- Repeat these steps for the remaining Talons.

USING ROBOTMAP – VISUAL INSTRUCTIONS

IN DRIVETRAIN.JAVA

1.
```
public Drivetrain() {
    leftFrontTalon = new Talon(RobotMap.DRIVETRAIN_LEFT_FRONT_TALON);
```

2.


3.


IN ROBOTMAP.JAVA

4.
```
public static final int DRIVETRAIN_LEFT_FRONT_TALON = 0;
```

YOUR CODE SHOULD LOOK SIMILAR TO THIS IN DRIVETRAIN.JAVA AND ROBOTMAP.JAVA

```java
public class Drivetrain extends Subsystem {
    // Put methods for controlling this subsystem
    // here. Call these from Commands.

    Talon leftFrontTalon = null;
    Talon leftBackTalon = null;
    Talon rightFrontTalon = null;
    Talon rightBackTalon = null;

    public Drivetrain() {
        // Talons
        leftFrontTalon = new Talon(RobotMap.DRIVETRAIN_LEFT_FRONT_TALON);
        leftBackTalon = new Talon(RobotMap.DRIVETRAIN_LEFT_BACK_TALON);
        rightFrontTalon = new Talon(RobotMap.DRIVETRAIN_RIGHT_FRONT_TALON);
        rightBackTalon = new Talon(RobotMap.DRIVETRAIN_RIGHT_BACK_TALON);
    }
```

```java
public class RobotMap {

    // For example to map the left and right motors, you could define
    // following variables to use with your drivetrain subsystem.
    // public static int leftMotor = 1;
    // public static int rightMotor = 2;

    // Talons
    public static final int DRIVETRAIN_LEFT_FRONT_TALON = 0;
    public static final int DRIVETRAIN_LEFT_BACK_TALON = 1;
    public static final int DRIVETRAIN_RIGHT_FRONT_TALON = 2;
    public static final int DRIVETRAIN_RIGHT_BACK_TALON = 3;
```

NOW THAT WE'VE CREATED OUR DRIVETRAIN CLASS,

WE NEED TO TELL THE OTHER PROGRAM FILES ABOUT IT

AN EASILY FORGOTTEN, BUT CRUCIAL STEP!

ADDING THE SUBSYSTEM TO ROBOT.JAVA

ADDING THE SUBSYSTEM TO ROBOT.JAVA – TEXT INSTRUCTIONS

- When a robot program runs on the roboRIO it only runs the main file Robot.java and anything Robot.java links to.
- We have created a new subsystem but we have not yet linked it to Robot.java. **WE MUST ALWAYS DO THIS!**
- In Robot.java we will create a new **global** variable of type **Drivetrain** named **m_drivetrain** and set its value to **null**.
- In the robotInit() method add: m_drivetrain = new Drivetrain();
  - **IMPORTANT**: This must always be done above OI and Telemetry/SmartDashboard (if present).
- Now when we use this subsystem in commands we must call Robot.drivetrain. to get access to it and its methods.

ADDING THE SUBSYSTEM TO ROBOT.JAVA – VISUAL INSTRUCTIONS

1.

```
public class Robot extends TimedRobot {
    public static ExampleSubsystem m_subsystem = new ExampleSubsystem();
    public static Drivetrain m_drivetrain = null;
    public static OI m_oi;
```

2.

```
@Override
public void robotInit() {
    m_drivetrain = new Drivetrain();
    m_oi = new OI();
    m_chooser.addDefault("Default Auto", new ExampleCommand());
    // chooser.addObject("My Auto", new MyAutoCommand());
    SmartDashboard.putData("Auto mode", m_chooser);
}
```

MAKING OUR ROBOT CONTROLLABLE

SETTING UP JOYSTICKS

CREATING A JOYSTICK – TEXT INSTRUCTIONS

- Open OI.java
- We need to create a new joystick
  - Type: `public Joystick driverController = new Joystick(RobotMap.OI_DRIVER_CONTROLLER);`
    - A variable driverController of type Joystick pointing to a joystick on port OI_DRIVER_CONTROLLER from RobotMap

  - Click the **Lightbulb** to create a new **CONSTANT** and set the value to the port number the joystick uses on the laptop (this can be found in the Driverstation software).

CREATING A JOYSTICK – VISUAL INSTRUCTIONS

1.

```
public class OI {
    public Joystick driverController = new Joystick(RobotMap.OI_DRIVER_CONTROLLER);
```

```
public class RobotMap {

    // For example to map the left and right motors, you could define the
    // following variables to use with your drivetrain subsystem.
    // public static int leftMotor = 1;
    // public static int rightMotor = 2;

    // Talons
    public static final int DRIVETRAIN_LEFT_FRONT_TALON = 0;
    public static final int DRIVETRAIN_LEFT_BACK_TALON = 1;
    public static final int DRIVETRAIN_RIGHT_FRONT_TALON = 2;
    public static final int DRIVETRAIN_RIGHT_BACK_TALON = 3;

    // Joystick
    public static final int OI_DRIVER_CONTROLLER = 0;
```

2.

TIME FOR ROBOTDRIVE

LET'S PROGRAM AN ARCADE DRIVE

WHAT IS ROBOTDRIVE?

- The FIRST RobotDrive class has many preconfigured methods available to us including DifferentialDrive, and many alterations of MecanumDrive.
- DifferentialDrive contains subsections such as TankDrive and ArcadeDrive.
  - For more information and details on drive bases see the WPILib documentation
- For our tutorial we will be creating an ArcadeDrive
- Arcade drives run by taking a moveSpeed and rotateSpeed. moveSpeed defines the forward and reverse speed and rotateSpeed defines the turning left and right speed.
- To create an arcade drive we will be using our already existing Drivetrain class and adding to it.

SETTING UP A ROBOT DRIVE – TEXT INSTRUCTIONS

- In the same place we created our talons (outside of the constructor) we will create a new RobotDrive
  - Type: DifferentialDrive differentialDrive = null;
- To use DifferentialDrive we need to first create SpeedControllerGroups for the left and right sides. In our constructor…
  - Type: SpeedControllerGroup leftMotors = new SpeedControllerGroup(leftFrontTalon, leftBackTalon);
  - Type: SpeedControllerGroup rightMotors = new SpeedControllerGroup(rightFrontTalon, rightBackTalon);
- We must also initialize the DifferentialDrive like we did the talons. In the constructor…
  - Type: differentialDrive = new DifferentialDrive(leftMotors, rightMotors);

SETTING UP A ROBOT DRIVE – VISUAL INSTRUCTIONS

```
public class Drivetrain extends Subsystem {
    // Put methods for controlling this subsystem
    // here. Call these from Commands.

    Talon leftFrontTalon = null;
    Talon leftBackTalon = null;
    Talon rightFrontTalon = null;
    Talon rightBackTalon = null;

1.  DifferentialDrive differentialDrive = null;
```

```
public Drivetrain() {
    leftFrontTalon = new Talon(RobotMap.DRIVETRAIN_LEFT_FRONT_TALON);
    leftBackTalon = new Talon(RobotMap.DRIVETRAIN_LEFT_BACK_TALON);
    rightFrontTalon = new Talon(RobotMap.DRIVETRAIN_RIGHT_FRONT_TALON);
    rightBackTalon = new Talon(RobotMap.DRIVETRAIN_RIGHT_BACK_TALON);

2.  SpeedControllerGroup leftMotors = new SpeedControllerGroup(leftFrontTalon, leftBackTalon);
    SpeedControllerGroup rightMotors = new SpeedControllerGroup(rightFrontTalon, rightBackTalon);

3.  differentialDrive = new DifferentialDrive(leftMotors, rightMotors);
}
```

Table of Contents

CREATING THE ARCADEDRIVE METHOD – TEXT INSTRUCTIONS

- Now it's time to make an arcadeDrive from our differentialDrive.
  - Let's create a public void method called "arcadeDrive" with type "double" parameters moveSpeed and rotateSpeed.
  - By putting something in the parentheses it makes the method require a parameter when it is used. When the method gets used and parameters are passed, they will be store in moveSpeed and rotateSpeed (in that order).
  - In the method type: differentialDrive.arcadeDrive(moveSpeed, rotateSpeed);
    - The arcadeDrive method takes parameters moveValue and rotateValue.
    - If you want to limit the max speed you can multiple the speeds by a decimal (i.e. 0.5*moveSpeed will make the motors only move half of their maximum speed)
      - You may want to do this for initial testing to make sure everything is going the right direction.

CREATING THE ARCADEDRIVE METHOD – VISUAL INSTRUCTIONS

1.

```
public void arcadeDrive(double moveSpeed, double rotateSpeed) {
    differentialDrive.arcadeDrive(moveSpeed, rotateSpeed);
}
```

2.

NOTE:

- At this point you could instead create a tank drive, however implementation differs slightly.
  - To do so do differentialDrive.tankDrive

GIVING THE ROBOT ORDERS:

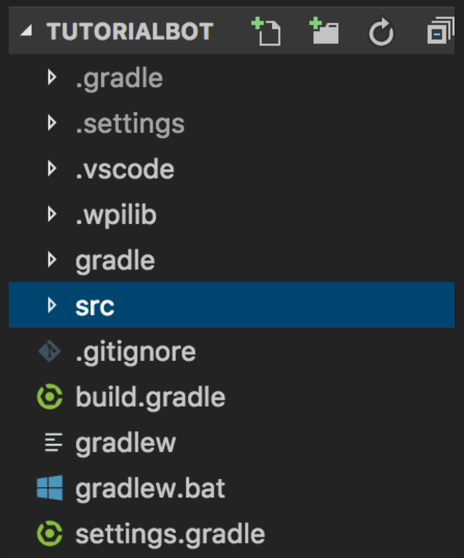
- **Methods** tell the robot what it can do but in order to make it do these things we must give it a **command**.
- Now that we have created the method, we need to create a command to call and use that method.
- Let's create a command called DriveArcade that calls arcadeDrive method we just created!

CREATING THE DRIVEARCADE COMMAND – TEXT INSTRUCTIONS

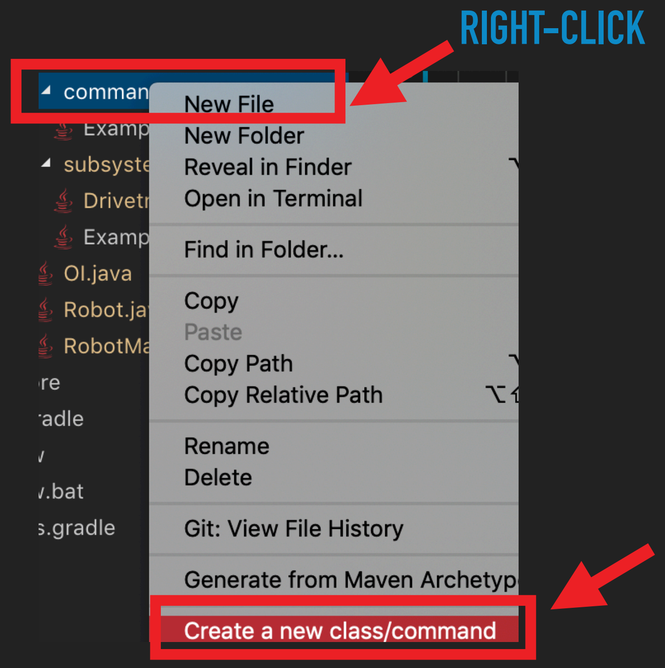
- Double click on the **src** folder to expand it. Do the same for **commands**
- Right click on **commands** and select **Create a new class/ command.**
- Select **Command** and type **DriveArcade** for the name and hit enter on your keyboard.
- Click on the newly create **DriveArcade.java**

CREATING THE DRIVEARCADE COMMAND - VISUAL INSTRUCTIONS

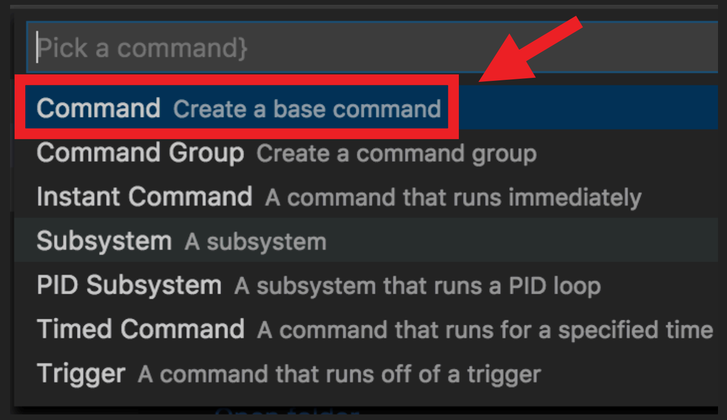
1.



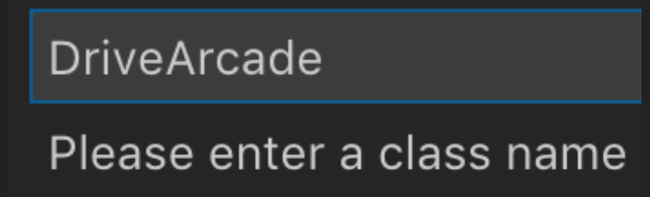
2.



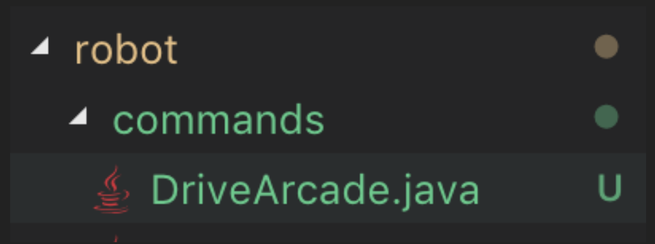
3.



4.



5.



WHAT A NEWLY CREATED COMMAND LOOKS LIKE THIS:

```java
public class DriveArcade extends Command {
  public DriveArcade() {
    // Use requires() here to declare subsystem dependencies
    // eg. requires(chassis);
  }

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

  // Called repeatedly when this Command is scheduled to run
  @Override
  protected void execute() {
  }

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

  // Called once after isFinished returns true
  @Override
  protected void end() {
  }

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

PARTS OF A COMMAND

- **Constructor** - Called when the robot program is first loaded.
  - Subsystem dependencies are declared here.
- **Initialize** - Called **ONCE** just before this Command runs the first time.
- **Execute** - Called **REPEATEDLY** when this Command is scheduled to run
- **isFinished** - Make this return **TRUE** when this Command no longer needs to run execute() (initialize always runs once regardless).
- **End** - Called **ONCE** after isFinished returns true
- **Interrupted** - Called when **another command** which requires one or more of the same subsystems is scheduled to run

FILLING OUT THE CONSTRUCTOR – TEXT INSTRUCTIONS

- In the constructor DriveArcade() type:
  `requires(Robot.m_drivetrain);`
  - This means, this command will end all other commands currently using drivetrain and will run instead when executed.
  - It also means, other commands that require drivetrain will stop this command and run instead when executed.
- **IMPORTANT:** If you use the **Lightbulb** to import 'Robot'. Be sure to import the one with "frc.robot"

FILLING OUT THE CONSTRUCTOR – VISUAL INSTRUCTIONS

1.

```
public class DriveArcade extends Command {
    public DriveArcade() {
        // Use requires() here to declare subsystem dependencies
        // eg. requires(chassis);
        requires(Robot.m_drivetrain);
    }
```

FILLING OUT THE EXECUTE METHOD – TEXT INSTRUCTIONS

- We will create 2 variables of type double called moveSpeed and rotateSpeed.
  - We want these variables to be the value of the axis of the controller we are using to drive the robot. So we will set them equal to that by using the joystick getRawAxis method.
  - Controllers return an axis value between 1 and -1 to indicate how far the joystick is pushed up or down. Our personal controller returns up as -1 so we want to invert it.
    - In Java you can put a negative " **-** "in front of a numeric value to invert it (value * -1)
    - The joystick's getRawAxis method will get the position value of the axis as you move it. The method takes parameter "axis number." (This can be found in the Driverstation software and we will store it in RobotMap).
- Below that we want to call the arcadeDrive method we created in Drivetrain and give it the variables moveSpeed and rotateSpeed we created as parameters.

FILLING OUT THE EXECUTE METHOD – VISUAL INSTRUCTIONS

1.
2.

```
protected void execute() {
    double moveSpeed = -Robot.m_oi.driverController.getRawAxis(RobotMap.DRIVER_CONTROLLER_MOVE_AXIS);
    double rotateSpeed = Robot.m_oi.driverController.getRawAxis(RobotMap.DRIVER_CONTROLLER_ROTATE_AXIS);

    Robot.m_drivetrain.arcadeDrive(moveSpeed, rotateSpeed);
}
```

FILLING OUT THE ISFINISHED METHOD – TEXT INSTRUCTIONS

- Since we will be using this command to control the robot we want it to run indefinitely.
  - To do this we are going to continue having isFinished return false, meaning the command will never finish. (We don't need to change anything as this is the default)
  - If we did want a command to finish, we make this return true.
    - This can be done by replacing false with true or…
    - Making a condition which can return true
      - For example (timePassed > 10) will return true after 10 seconds but return false anytime before 10 seconds have passed.

FILLING OUT THE ISFINISHED METHOD – VISUAL INSTRUCTIONS

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

NOTHING TO CHANGE, THE DEFAULT IS FINE.

FILLING OUT THE END METHOD – TEXT INSTRUCTIONS

- We will call the arcadeDrive method and give it 0 and 0 as the parameters.
  - This make the motors stop running when the command ends by setting the movement speed to zero and rotation speed to zero.

FILLING OUT THE INTERRUPTED METHOD – TEXT INSTRUCTIONS

- We will make it call end.
  - This makes the end method get called if the command gets interrupted. (No reason to re-write code, call end() to use what we already have!)

FILLING OUT THE END METHOD – TEXT INSTRUCTIONS

1.

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

FILLING OUT THE INTERRUPTED METHOD – TEXT INSTRUCTIONS

2.

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

YOUR CODE SHOULD LOOK SIMILAR TO THIS IN DRIVEARCADE.JAVA

```java
public class DriveArcade extends Command {
    public DriveArcade() {
        // Use requires() here to declare subsystem dependencies
        // eg. requires(chassis);
        requires(Robot.m_drivetrain);
    }

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

    // Called repeatedly when this Command is scheduled to run
    @Override
    protected void execute() {
        double moveSpeed = -Robot.m_oi.driverController.getRawAxis(RobotMap.DRIVER_CONTROLLER_MOVE_AXIS);
        double rotateSpeed = Robot.m_oi.driverController.getRawAxis(RobotMap.DRIVER_CONTROLLER_ROTATE_AXIS);

        Robot.m_drivetrain.arcadeDrive(moveSpeed, rotateSpeed);
    }

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

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

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

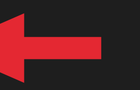
USING INITDEFAULTCOMMAND – TEXT INSTRUCTIONS

- Commands within this method run when the robot is enabled.
- They also run if no other commands using the subsystem are running.
  - This is why we write **requires(Robot.m_subsystemName)** in the commands we create, it ends currently running commands using that subsystem to allow a new command is run.
- Back in Drivetrain.java in the initDefaultCommand() method we will put our newly created command
  - Type: setDefaultCommand(new DriveArcade());

USING INITDEFAULTCOMMAND – VISUAL INSTRUCTIONS

1.

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



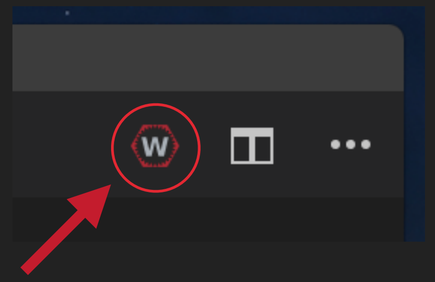
DEPLOY!

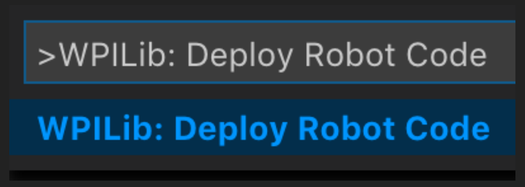
YOUR CODE IS NOW READY TO RUN.

DEPLOYING CODE – TEXT INSTRUCTIONS

- To deploy code, first make sure your computer is either connected to the robot via USB, Ethernet, or the Wireless Network it is broadcasting.
- Make sure your team number in **wpilib_preferences.json** in the **.wpilib** folder is set to the same team number your roboRIO was programmed for (it should be the number you set when creating the project and you will NOT need to check this every time as it should not change by itself).
- Select the W icon from the tab bar or use the shortcut by holding down **Ctrl+Shift+P** at the same time. (Replace ctrl with cmd on macOS)
- Type and hit enter or select **WPILib: Deploy Robot Code**

DEPLOYING CODE – VISUAL INSTRUCTIONS

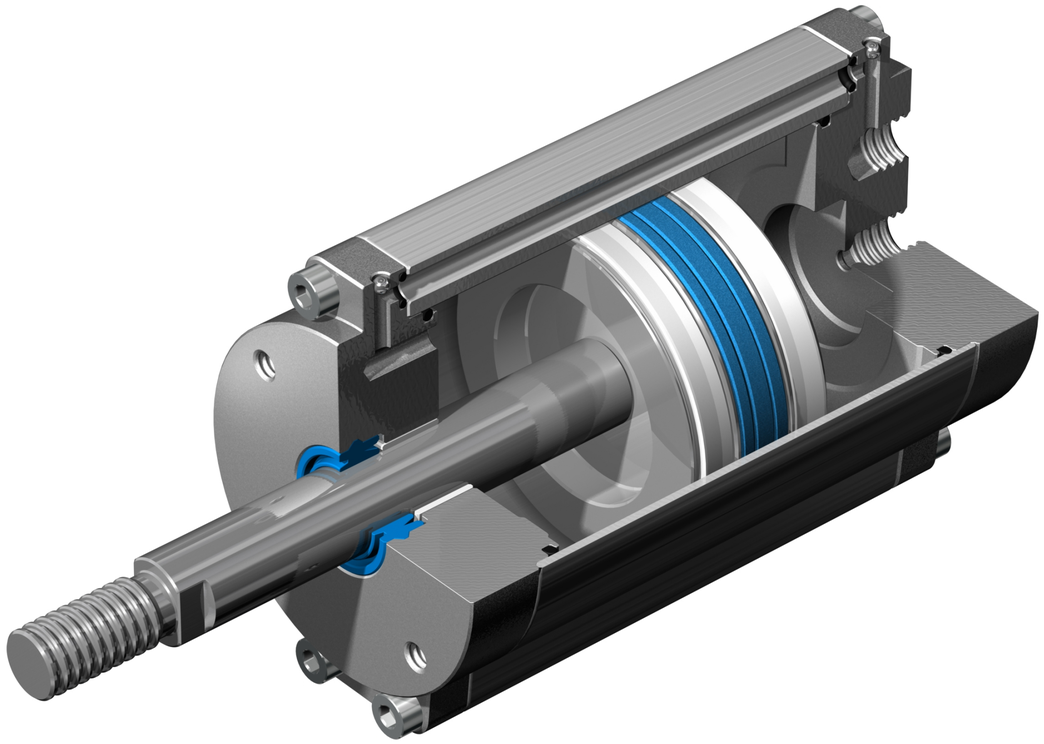
1. 

2. 

- ***Alternatively*** you can do one of the following:
  - Use **Shift+F5** at any time to deploy. (you may also need to hold fn depending on your computer configuration)
  - Right-click on the build.gradle file in the project hierarchy and select "Build Robot Code”
  - Open the shortcut menu indicated by the ellipses in the top right corner of the VS Code window and select "Build Robot Code"

TIME TO TEST!

- Open up the DriverStation software on any computer that has it installed.
- Enable the robot
- Try moving the joysticks on your controller when enabled.
  - If it doesn't, check your port numbers for your controller, axes, and motor controllers



SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

THIS SECTION HAS NOT YET BEEN CREATED

PART 2:
CHECK THE AIR PRESSURE

USING PNEUMATICS

WHAT ARE PNEUMATICS?

- You have probably heard of hydraulics before (which is based on water pressure). Pneumatics are essentially the same but with air pressure.
- Unlike motors and gears which are commonly infinitely positional, pneumatic cylinders are normally dual-positional or sometimes tri-positional.
- Pneumatic cylinders are actuated through devices called solenoids.
- Solenoids are used to control pneumatic pistons (air cylinders) similar to how Talons control motors.

WHAT ARE SOLENOIDS?

- Cylinders are actuated with either **single solenoids** or **double solenoids**.
- A **single solenoid** actuates with one air line, using air to switch to and hold the extended state and releasing air (sometimes paired with a spring) to allow the cylinder to return to the retracted state.
  - A single solenoid valve has one solenoid, and shifts when voltage is **CONSTANTLY** supplied to that solenoid. When voltage is removed, it shifts back to a "home" position.
- A **double solenoid** actuates with two air lines, using air to switch and hold states between retracted and extended.
  - A double solenoid has two solenoids, and when voltage is supplied to one (and not the other) the valve shifts.
- Solenoids are connected to the Pneumatics Control Module (PCM)
  - The PCM is connected to the roboRIO via the CAN bus.

PROGRAMMING SOLENOIDS

- For this tutorial we are going to create a new subsystem called shooter and add one pneumatic piston (cylinder) which will be used for changing the pitch of the shooter.
  - Create your new Shooter subsystem on your own now
    - **(DON'T FORGET TO ADD IT TO ROBOT.JAVA, click here for a refresher).**
- It will be controlled through a double solenoid.
- We are going to create a DoubleSolenoid named pitchSolenoid.
  - DoubleSolenoids have 2 controllable positions (deployed(forward) and retracted(reverse)).
  - The DoubleSolenoid constructor takes 2 parameters - (new DoubleSolenoid(port1, port2))
    - Forward control and Reverse control ports on the PCM.
    - Like all ports we use, we will store this in the RobotMap.
  - Create your DoubleSolenoid named pitchSolenoid now using the same technique used to create a talon but replacing Talon with DoubleSolenoid. (For single solenoids just use Solenoid).

YOUR SHOOTER.JAVA AND ROBOTMAP.JAVA SHOULD LOOK SIMILAR TO THIS

```java
public class Shooter extends Subsystem {

    DoubleSolenoid pitchSolenoid = null;

    public Shooter() {
        pitchSolenoid = new DoubleSolenoid(RobotMap.SHOOTER_PITCH_SOLENOID_DEPLOY, RobotMap.SHOOTER_PITCH_SOLENOID_RETRACT);
    }
```

```java
// Joysticks
public static final int OI_DRIVER_CONTROLLER = 0;
public static final int JOYSTICK_MOVE_AXIS = 1;
public static final int JOYSTICK_ROTATE_AXIS = 2;

// Solenoids
public static final int SHOOTER_PITCH_SOLENOID_DEPLOY = 0;
public static final int SHOOTER_PITCH_SOLENOID_RETRACT = 1;
```

CREATING THE PITCHUP AND PITCHDOWN METHODS

- Create a public void method called pitchUp.
  - Inside type: pitchSolenoid.set(Value.kForward);
    - This sets the value of the solenoid to forward (deployed)
    - **Note:** if you wanted multiple solenoids to deploy at the same time also have them do .set(Value.kForward);
- Do the same for the shiftLow method but change kFoward to kReverse.
- Now its time to make our commands that run these methods.

```
public void pitchUp() {
    pitchSolenoid.set(Value.kForward);
}
public void pitchDown() {
    pitchSolenoid.set(Value.kReverse);
}
```

CREATING THE COMMANDS TO DEPLOY AND RETRACT

- Now that we have created the methods we must create the commands to use them.
- In this case we will make two commands, ShooterUp and ShooterDown
- Since changing the state of a solenoid only requires us to send a signal once (not continuously) we can place our methods in the initialize portions of their respective commands.
  - For that same reason, we never need to run execute so the command isFinished as soon as it starts (and runs initialize).
    - So for these commands we will set isFinished to **true**.
- Don’t forget to add requires(Robot.shooter) to the constructor!

YOUR COMMANDS SHOULD LOOK SOMETHING LIKE THIS...

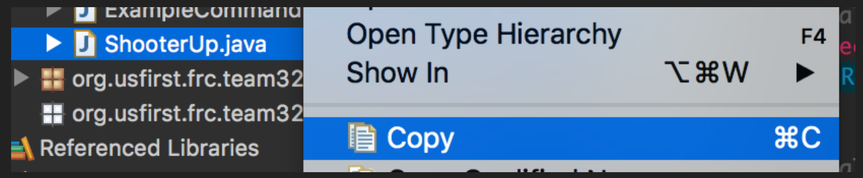
```java
public class ShooterUp extends Command {

    public ShooterUp() {
        // Use requires() here to declare
        // eg. requires(chassis);
        requires(Robot.shooter);
    }

    // Called just before this Command run
    protected void initialize() {
        Robot.shooter.pitchUp();
    }

    // Called repeatedly when this Command
    protected void execute() {
    }

    // Make this return true when this Com
    protected boolean isFinished() {
        return true;
    }
```

```java
public class ShooterDown extends Command {

    public ShooterDown() {
        // Use requires() here to declare s
        // eg. requires(chassis);
        requires(Robot.shooter);
    }

    // Called just before this Command runs
    protected void initialize() {
        Robot.shooter.pitchDown();
    }

    // Called repeatedly when this Command
    protected void execute() {
    }

    // Make this return true when this Comm
    protected boolean isFinished() {
        return true;
    }
```

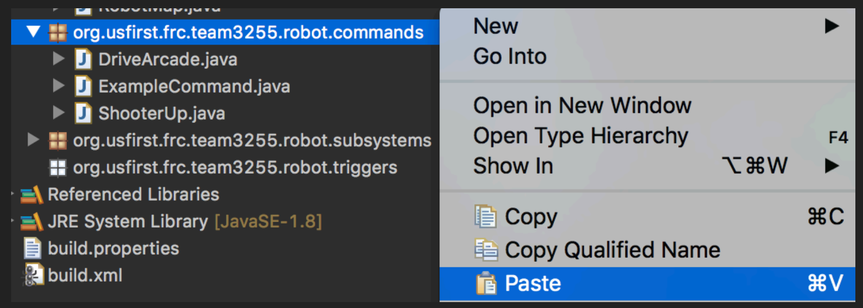
TIP: COPYING FILES

- Since the commands we just made were near identical we could have just written one, copied it, and modified only the part we need to. To duplicate a file:
- Select the file (in this case ShooterUp) and Run the copy command:
  - Right click -> Copy, **OR** from the menubar Edit -> Copy, **OR** use the keyboard shortcut Control C (Command C on Mac)
- Then paste the file (have the correct package selected, in this case commands)
  - Right click -> Paste, **OR** from the menubar Edit -> Paste, **OR** use the keyboard shortcut Control V (Command V on Mac)
- Then name the new file (in this case we will name it ShooterDown).
- Now we will change the parts that should differ (in this case change Robot.shooter.pitchUp(); to Robot.shooter.pitchDown();).

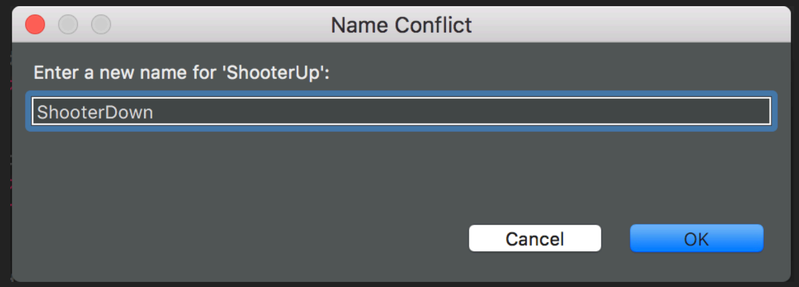
1)



2)



3)



4)

```
public class ShooterDown extends Command {

    public ShooterDown() {
        // Use requires() here to declare s
        // eg. requires(chassis);
        requires(Robot.shooter);
    }

    // Called just before this Command runs
    protected void initialize() {
        Robot.shooter.pitchDown();
    }
```

CREATING THE BUTTONS IN THE CODE

- Now that we have created our ShooterUp and ShooterDown commands we need a way to run them.
- Lets map them to buttons on our controller!
- Open OI.java
  - Under our created joystick we will create Button variables and assign them to a button on our joystick
  - Type: Button D1 = new JoystickButton(driverController, 1);
    - This creates a new Button named D1 (D representing driverController and 1 representing the button number) and sets it as a JoystickButton on the controller 'driverController' and button value 1 (this can be found in the Driverstation software).
    - Do this for the rest of the buttons on your controller.

```
public class OI {

    public Joystick driverController = new Joystick(RobotMap.OI_DRIVER_CONTROLLER);

    Button D1 = new JoystickButton(driverController, 1);
    Button D2 = new JoystickButton(driverController, 2);
    Button D3 = new JoystickButton(driverController, 3);
    Button D4 = new JoystickButton(driverController, 4);
    Button D5 = new JoystickButton(driverController, 5);
    Button D6 = new JoystickButton(driverController, 6);
    Button D7 = new JoystickButton(driverController, 7);
    Button D8 = new JoystickButton(driverController, 8);
    Button D9 = new JoystickButton(driverController, 9);
    Button D10 = new JoystickButton(driverController, 10);
```

MAPPING THE BUTTONS IN THE CODE

- Now that we have created the buttons in the code we can map certain commands to them.
- Create a constructor for OI
- In the constructor type: D1.whenPressed(new ShooterUp());
  - This means **when** the button D1 is **pressed** it runs the ShooterUp command and deploys our pneumatic piston.
  - Quick fix -> Import
  - There are other types of activations for buttons besides **whenPressed** like: **whenRelease, whileHeld, etc.** feel free to read more about them here.

```
public OI(){
    D1.whenPressed(new ShooterUp());
}
```

- **TIP:** you can change your import at the top of the file from "import org.usfirst.frc.team.robot.commands.ShooterUp;" to "import org.usfirst.frc.team.robot.commands.*;"
  - The asterisk makes it so call files in the .command package are imported. This way you only have to import once.

- **TIP:** You can read in detail what every method of the FIRST code does by reading the Java Docs

THIS SECTION HAS NOT YET BEEN CREATED

PART 3:
GIVING FEEDBACK

USING SENSORS AND SWITCHES

Table of Contents

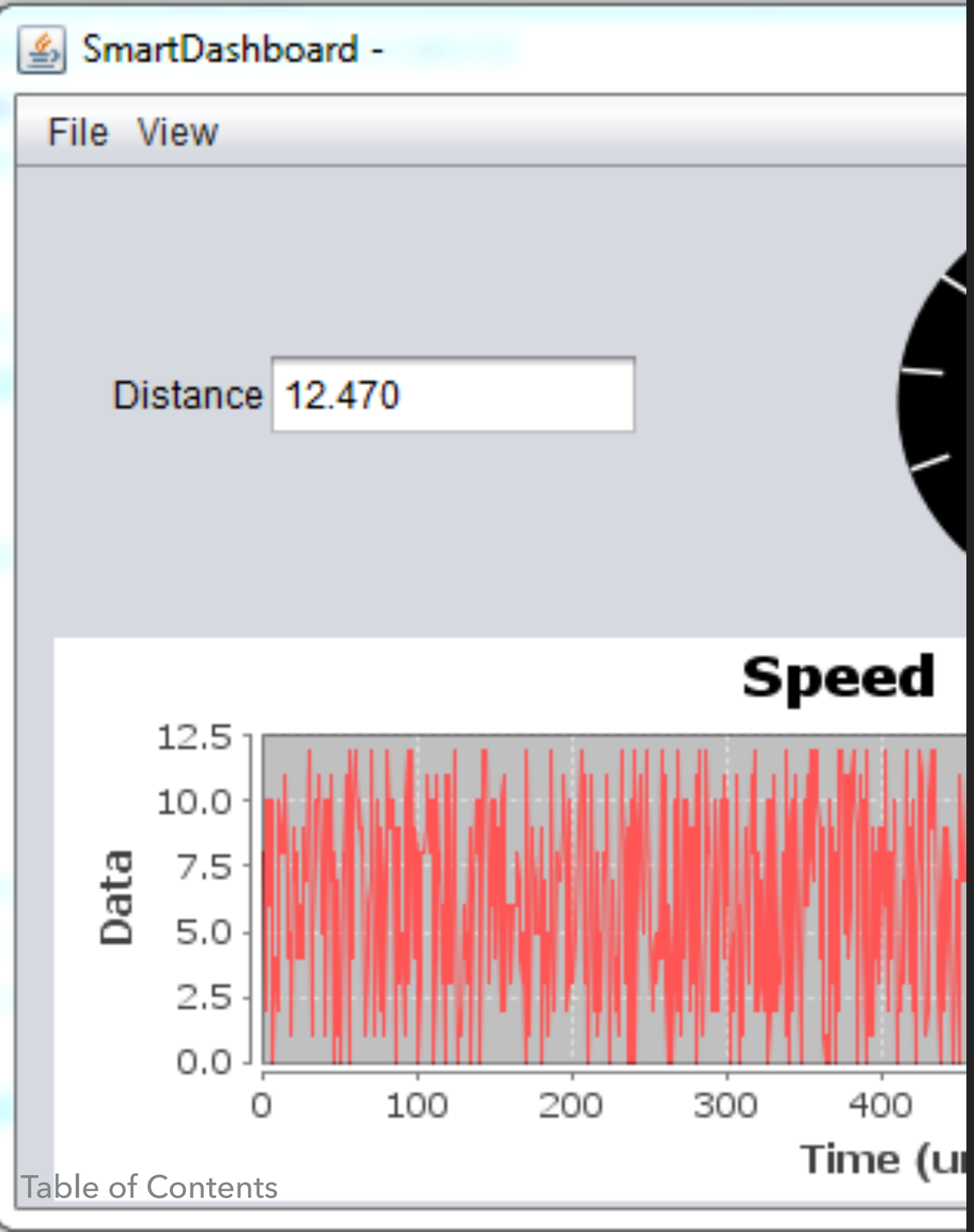


Table of Contents

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

THIS SECTION HAS NOT YET BEEN CREATED

PART 4:
GETTING FEEDBACK

USING THE SMARTDASHBOARD

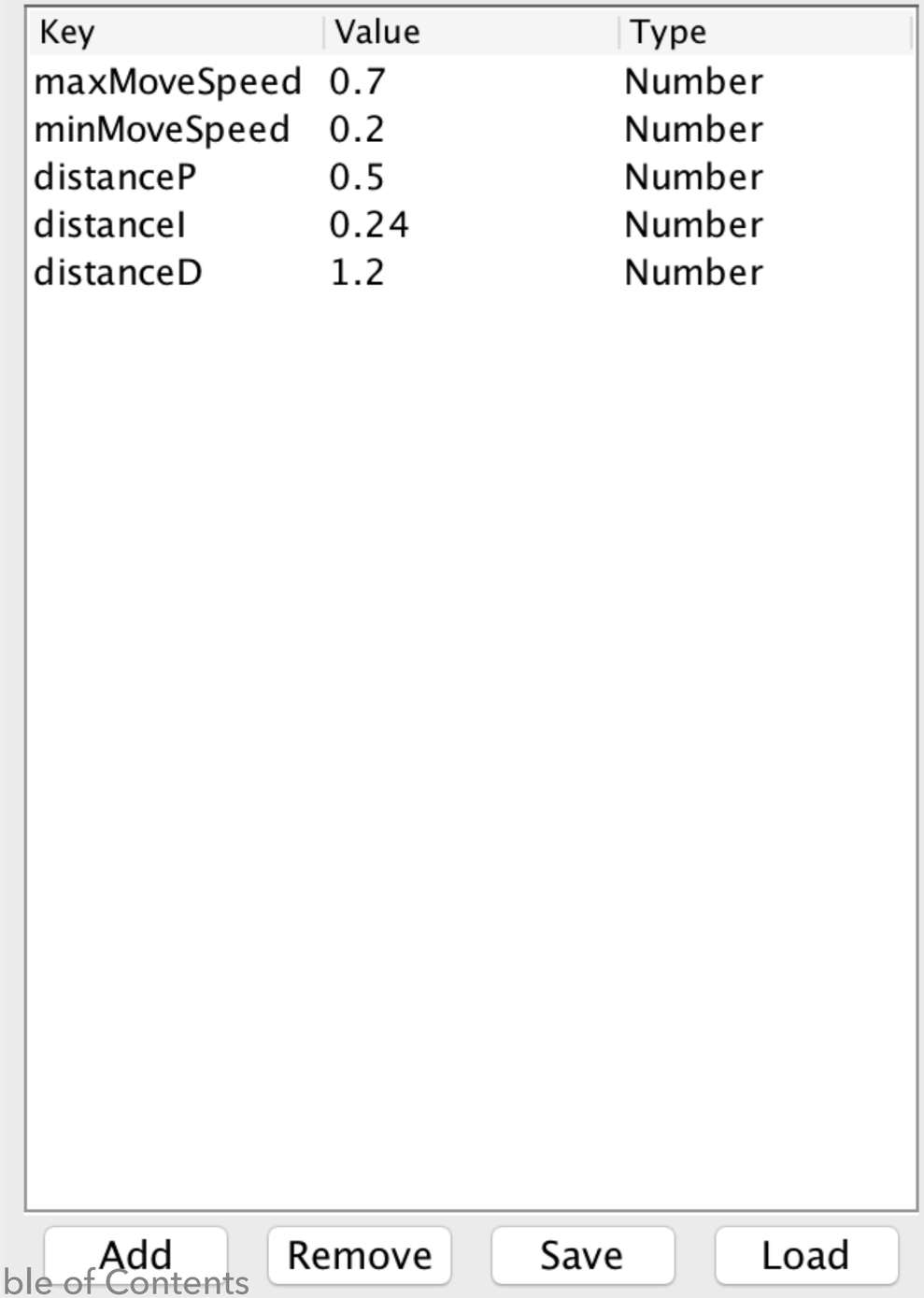


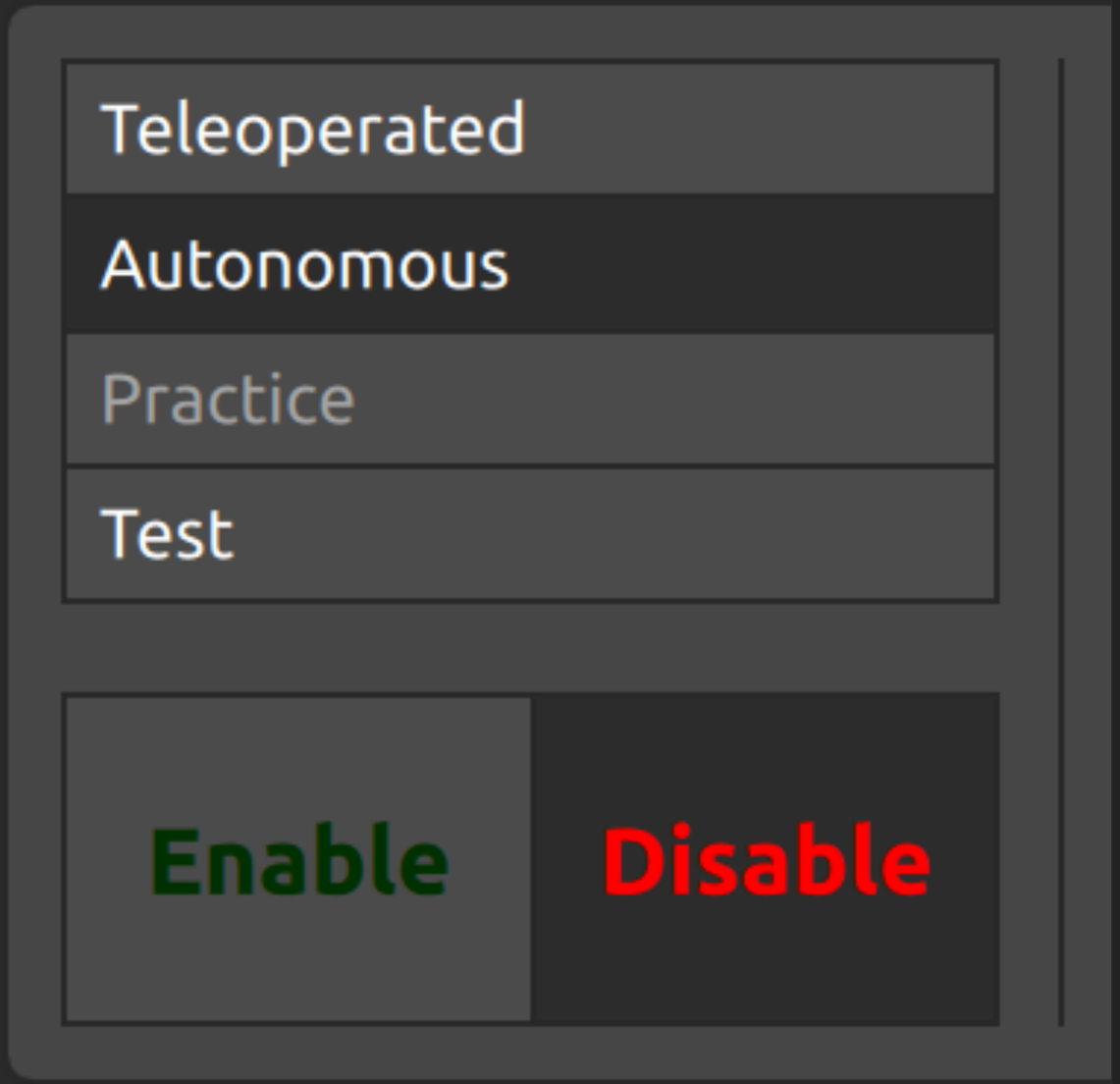
Table of Contents

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

THIS SECTION HAS NOT YET BEEN CREATED

PART 5:
SETTING SETTINGS

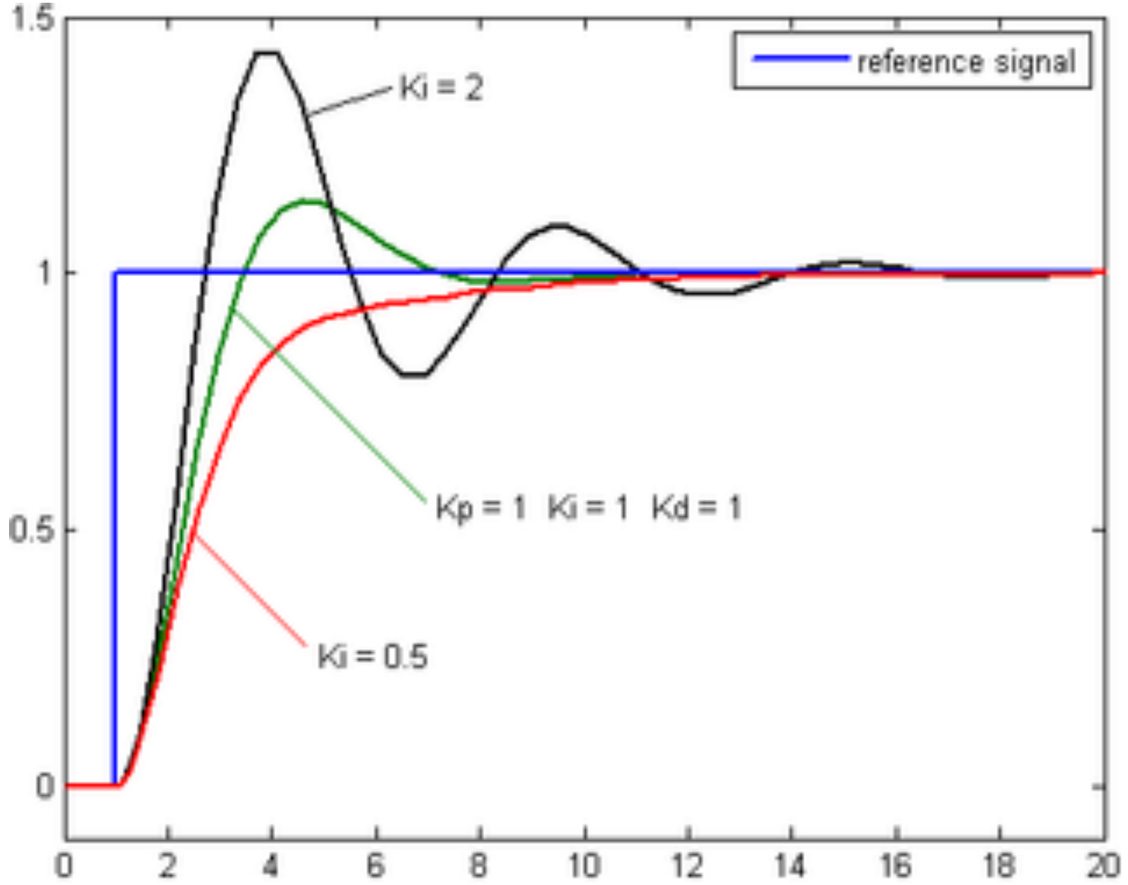
USING ROBOTPREFERENCES



THIS SECTION HAS NOT YET BEEN CREATED

PART 6:
AUTOMATICALLY MOVE!

CREATING AN AUTONOMOUS COMMAND



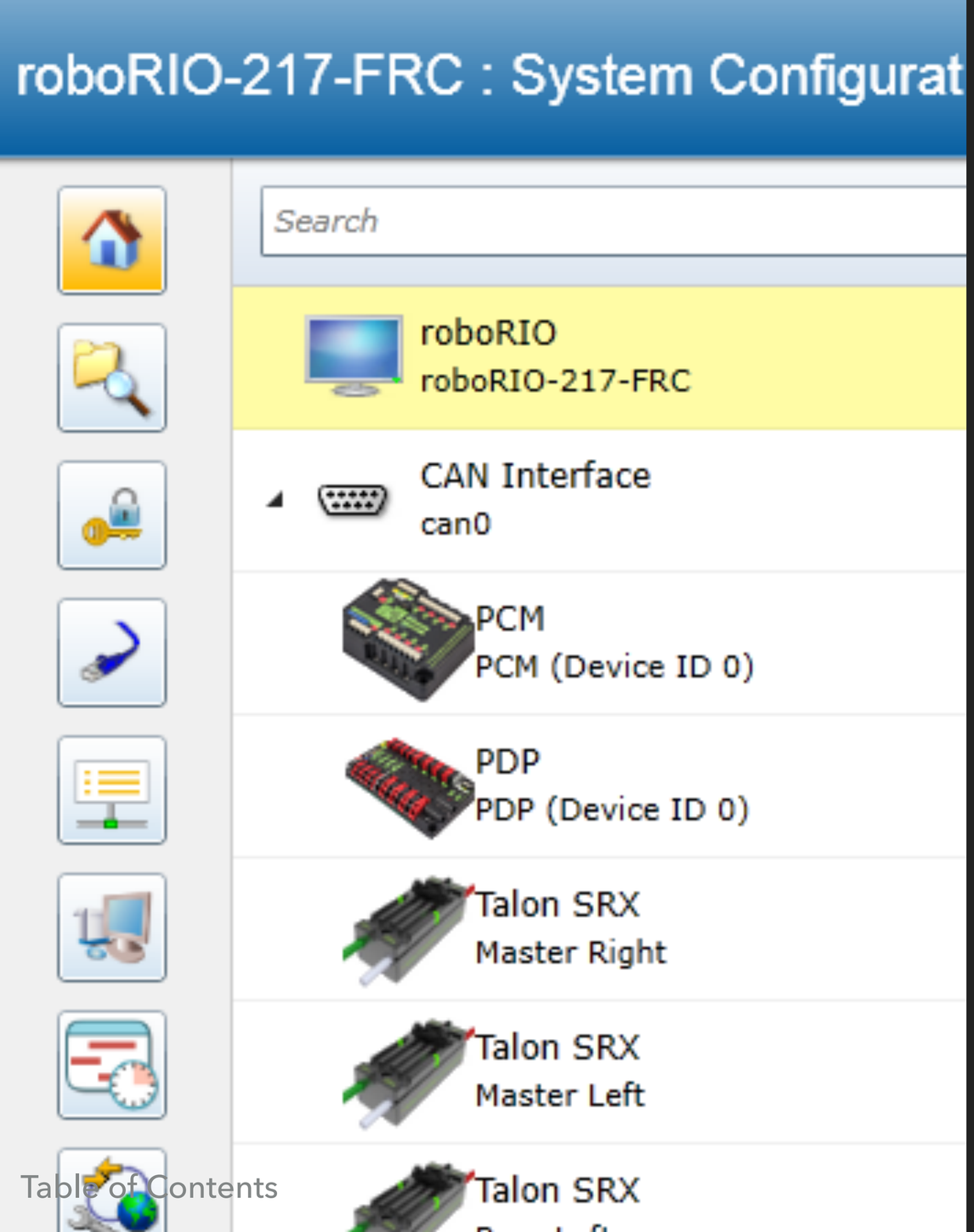
THIS SECTION HAS NOT YET BEEN CREATED

PART 7:

PROPORTIONAL INTEGRAL DERIVATIVE

GETTING STARTED WITH PID

Table of Contents



SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

THIS SECTION HAS NOT YET BEEN CREATED

PART #:
SETTING IDS

USING THE ROBORIO WEBDASH

Table of Contents

GitHub

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

THIS SECTION HAS NOT YET BEEN CREATED

VERSION CONTROL!

USING GITHUB!