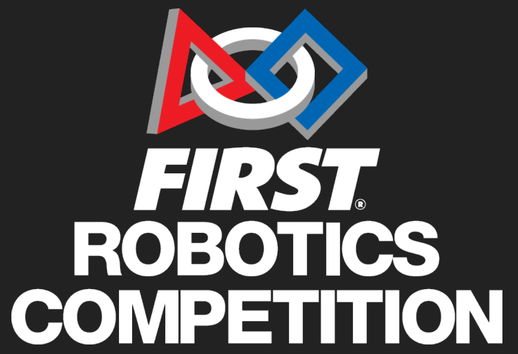


FRC JAVA PROGRAMMING

CREATED BY TAYLER UVA
FRC TEAM 3255 - THE SUPERNURDS

GITHUB.COM/TAYLERUVA
SUPERNURDS.COM

Updated for the 2017 Season

TABLE OF CONTENTS

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–F:

STEP AND BASICS OF WPILIB AND PROGRAMMING

LEARNING THE BASICS

PART A:

THE BRAINS OF THE BOT

THE ROBORIO

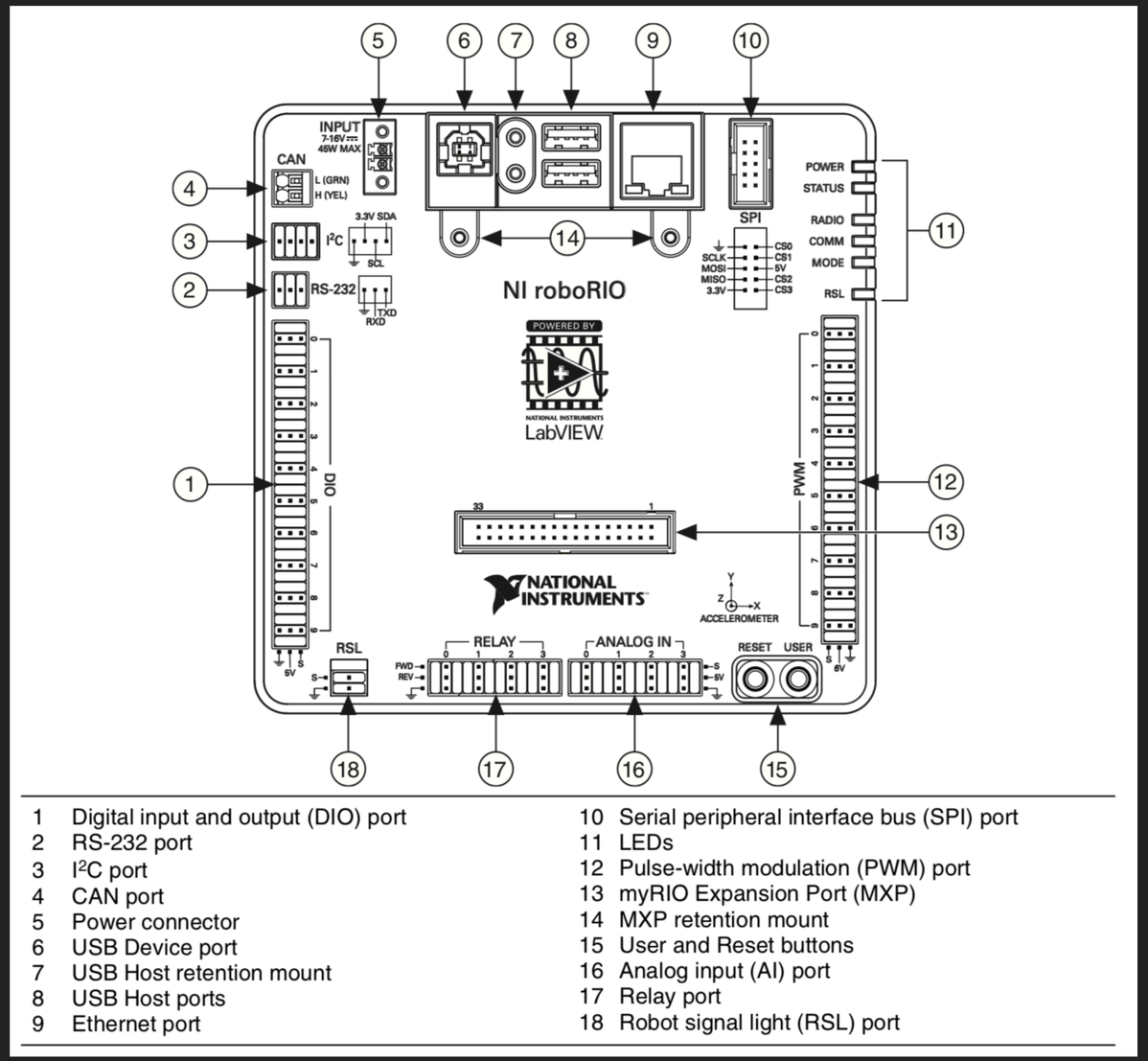
THE ROBORIO

- The roboRIO is the brain of a 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 has many different interfaces to connect it to motor controllers, servos, and sensors. Some include:
  - 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

PART B:

HOW DOES THE ROBOT SEE?

SENSORS

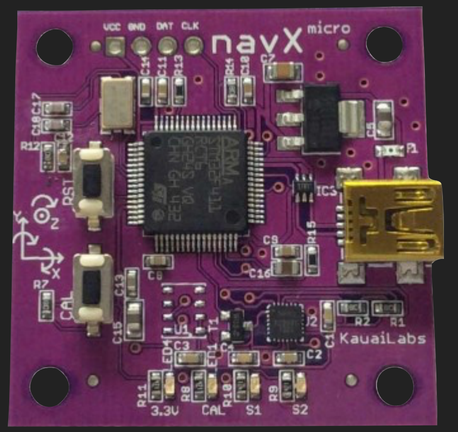
Table of Contents

SOME TYPES OF SENSORS

- Limit Switches - detects contact
- Camera - provide sight
- Encoders - measures rotational or linear motion
- Ultrasonic - measures distances
- Gyroscope - measures orientation
- Processed Vision - measures targets 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:

WHAT DOES THIS ALL MEAN?

WPILIB AND GENERAL PROGRAMMING BASICS

GENERAL PROGRAMMING BASICS

```
   int total;
   bool isFinished;

 int calculateSum();


 int add(num1, num2);

double half(int num1){
 double multiplier = 0.5;
 return num1*multiplier;
}
 int value;
 void increment(){
  value++;
 }

 //This is a comment
```

- Variables - objects that contain data, they are characterized by date type.
- Methods/Functions - can be thought of as subprograms or routines that run inside of your main program.
- Parameters - variables that are passed (sent to) a method for it to use.
- Scope - where a variable can be seen.
  - 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.
- Comment - a programmer-readable explanation or annotation in the source code of a program. It does not affect what the code does.

Table of Contents

GENERAL PROGRAMMING BASICS

- Programs use key words to define characteristics of variables or objects.
- Some data types for variables:
  - int - integers (whole numbers)
  - double - double precision floating point (fractional/decimal values)
  - boolean - true or false (true = 1 or false = 0) values.
- Other 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

Table of Contents

GENERAL PROGRAMMING BASICS

- 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
    - Multiple words that are joined together as a single word with the first letter of each of the multiple words capitalized so that each word that makes up the name can easily be read
    - For example:
      - ThreeMotorDrive
      - driveForward
      - setSpeed
- There are other naming conventions but for this tutorial we will use the camel cases

Table of Contents

WPILIB – COMMAND BASED ROBOT

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

WPILIB – COMMAND BASED ROBOT – SUBSYSTEMS

- A **subsystem** in programming (like in mechanical) is a subpart of the whole robot. (i.e. Drivetrain, Collector).
  - Here we define the components subsystem is made up of.
    - Defining motor controllers, sensors, etc.
  - Using a dog as an example: the **legs**, **tail**, and **head** are a **subsystems**.
    - The head is made up of eyes, nose, ears.

Table of Contents

WPILIB – COMMAND BASED ROBOT – COMMANDS

- A **command** tells the robot to do something. (i.e. MoveForward).
  - A **command** is an action a **subsystem(s)** performs.
  - Telling a dog the **command MoveForward** will make the **subsystem** "**legs**" move forward.
  - A **command** calls a **method(s)** from a **subsystem(s)** to make something happen.
    - **setSpeed(1.0)**
  - You teach a dog the command "**chase**" by having him **setSpeed(1.0)** (run) from his "**legs**" subsystem
  - When you yell **chase** he will start running at full speed
- For example, I have a **Drivetrain** subsystem with the method **setSpeed**. I then tell the robot to run the command **DriveFoward** which uses the **setSpeed** method from Drivetrain.

Table of Contents

WPILIB – COMMAND BASED ROBOT – METHODS

- Subsystems and Commands are made up of methods or functions.
- A **method** defines what a **subsystem** or **command** is able to do or modify (i.e. **setSpeed**, **resetEncoders**)
  - A dog's tail is able to wag and its legs are able to move at a certain speed
    - **Subsystem legs** has method **setSpeed**

Table of Contents

WPILIB – COMMAND BASED ROBOT – SUMMARY

- A **subsystem** is a part of a whole. A **method** is what each part can do. And a **command** tells to whole to do something and which parts (**methods**) its needs to do it.
  - A Dog has **legs**(**subsystem**). Its **legs** can **setSpeed**(**method**). You can tell him **chase(command)** which will make him use his **legs**(**subsystem**) to run at full speed (**setSpeed** (**method**))

Table of Contents

PART D:
LETS GET STARTED

INSTALLING NECESSARY SOFTWARE

Table of Contents

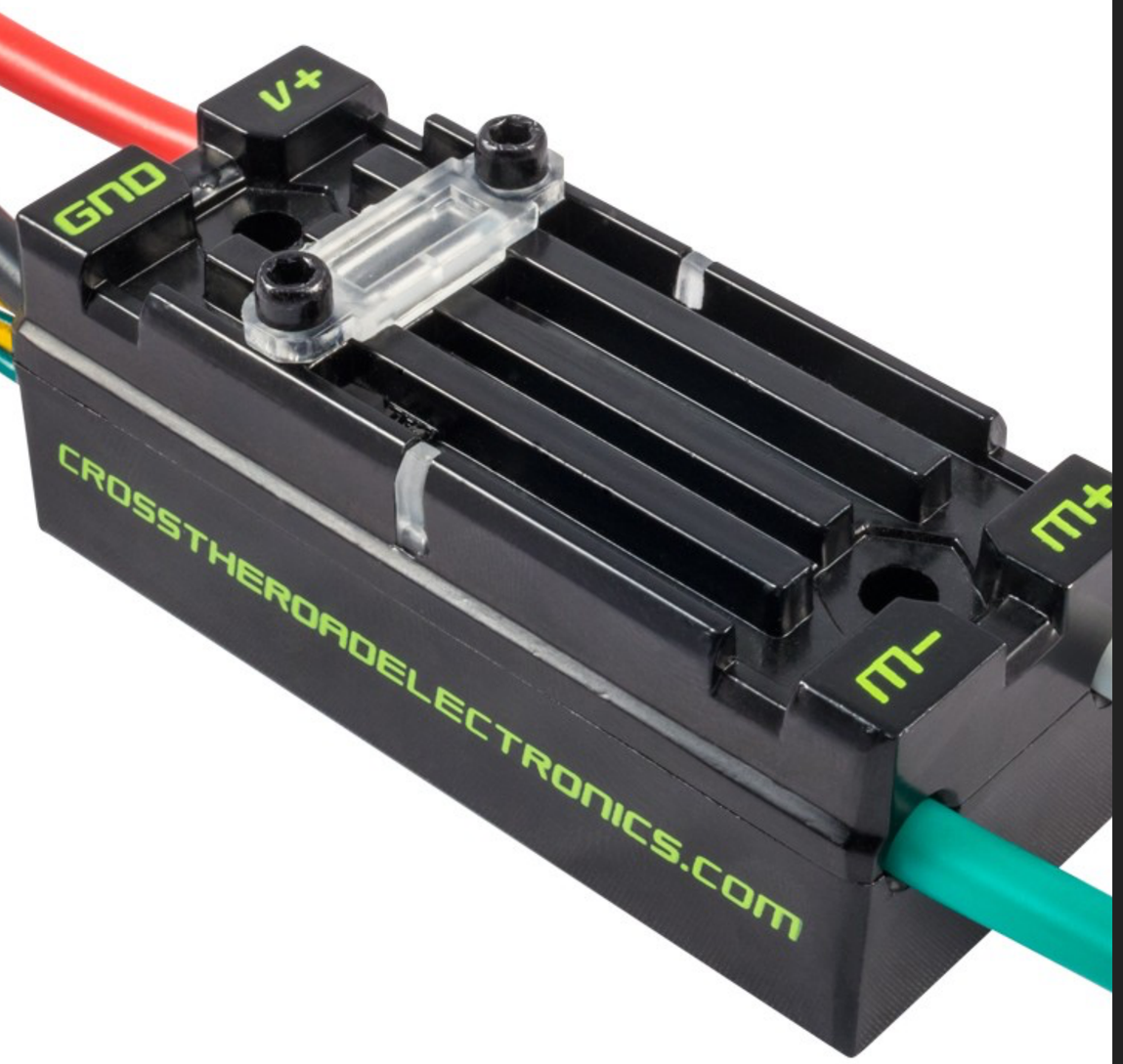
PLEASE FOLLOW OFFICIAL FIRST INSTALLATION GUIDES

All these steps must be **COMPLETED** and in **THIS ORDER**!

Official FIRST Installation Guides:

- Installing Eclipse (C++/Java)
- Installing the FRC Update Suite (All Languages)
- Imaging your roboRIO
- Installing Java 8 on the roboRIO using the FRC roboRIO Java Installer

These links should automatically update for the 2018 season.

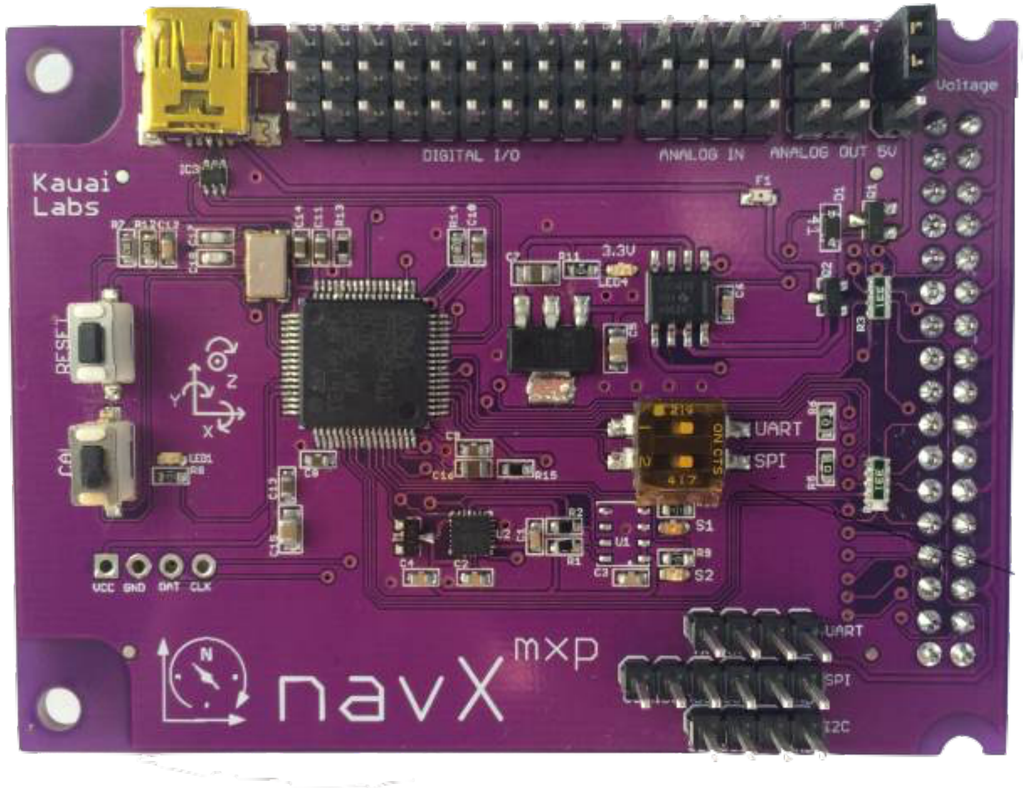


THIS SECTION HAS NOT YET BEEN CREATED

PART E:
SETTING UP A TALON SRX

INSTALLING CTRE CANTALON LIBRARIES

Table of Contents



THIS SECTION HAS NOT YET BEEN CREATED

PART F:
SETTING UP THE NAVX

INSTALLING KAUAILABS NAVX LIBRARIES

PARTS 1–7:
PROGRAMMING IN JAVA WITH WPILIB

PROGRAMMING FOR AN FRC ROBOT

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

PART 1:
LETS GET MOVING

CREATING A BASIC DRIVING ROBOT

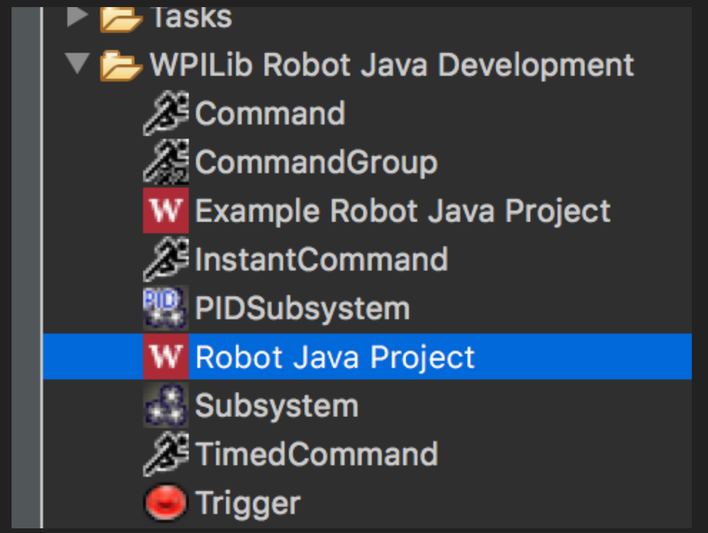
WHAT IS IN THIS ROBOT?

CREATING THE DRIVETRAIN SUBSYSTEM

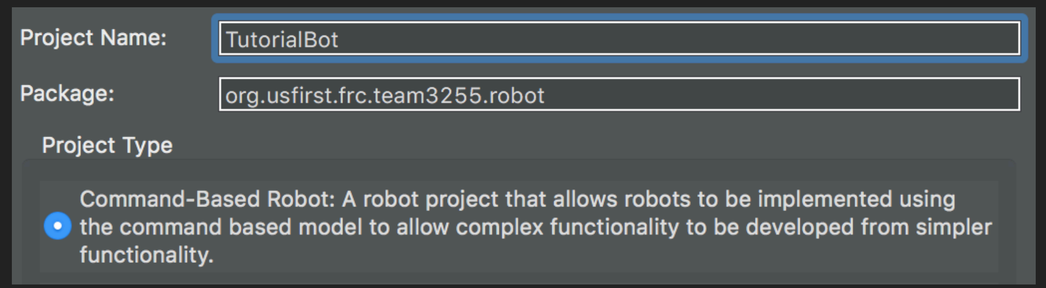
CREATING A NEW PROJECT

- Select File -> New -> Other -> WPILIB Robot Java Development -> Robot Java Project -> Command-Based Robot
  - Control (Mac OS: Command) N will bring up new -> other dialogue box - we will use this method from now on.
- If prompted, input your team number (this can be changed later in Preferences -> WPILib Preferences)
- Choose a name for your project (i.e. Robot2018)
  - You can also change the robot portion of the package to more easily identify different robot programs you have. (This is completely optional)
- Select Finish

1.



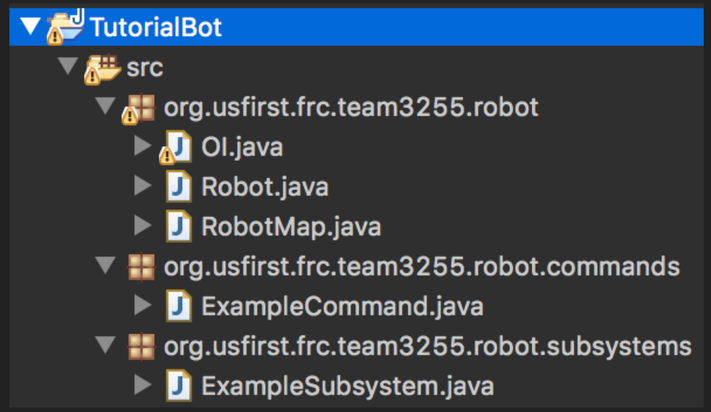
2.



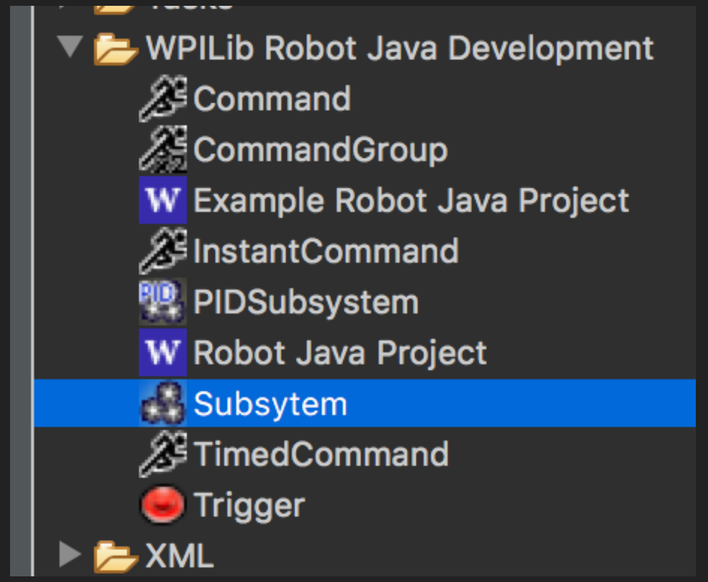
CREATING A NEW SUBSYSTEM

- Double click on your project to expand it. Do the same for the src, .robot, .commands, and .subsystems.
- Select .subsystems create a new subsystem (ctrl N -> WPILib -> Subsystem)
- Under Class Name type: Drivetrain
- Open Drivetrain.java

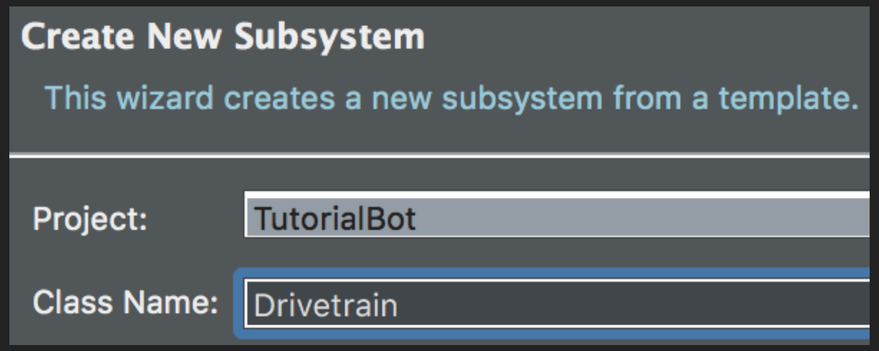
1.



2.



3.



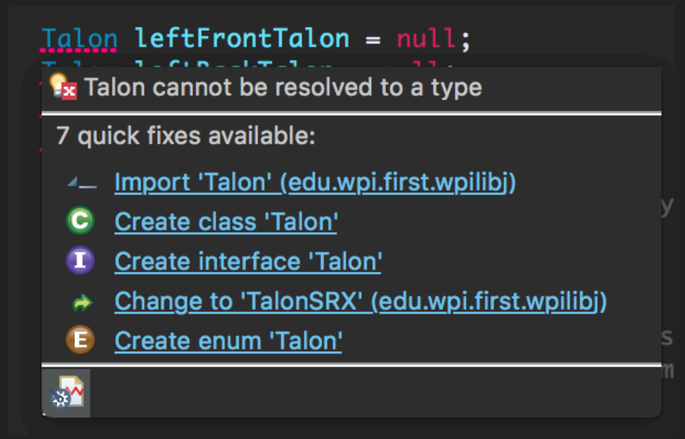
WHAT'S IN THE SUBSYSTEM?

- In the file of the Drivetrain 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 (brand of controller).
  - You could use other motor controllers such as Victor SPs or Talon SRXs but we will be using Talon SRs
  - You could alternatively only use 2 motors (left and right) however for this tutorial we will use 4.
- Lets create the talons and set their value to null.
  - null means they have a value of nothing and are empty.
    - We explicitly do this to make sure its empty at this point.
  - We will set their value later.
- We will name them leftFrontTalon, rightFrontTalon, leftBackTalon, rightBackTalon
  - These are variables that hold values
  - Class wide variables are normally created at the top level.
    - This is so they can be seen in all methods.

HOW IT SHOULD BE LOOKING...

```
8  public class Drivetrain extends Subsystem {
9
10     Talon leftFrontTalon = null;
11     Talon leftBackTalon = null;
12     Talon rightFrontTalon = null;
13     Talon rightBackTalon = null;
```

- In Eclipse (our programming software) red dotted lines means there is an error in your code. You can hover over them to see what the error is and **quick fixes**.



- In this case the Talons class needs to be imported. The code doesn't know what type “Talon” is until we import the class.
  - From this point on the tutorial will assume you do import quick fixes
- Click Import ‘Talon’ and the errors should clear. (Don't forget to save!)

CREATING A CONSTRUCTOR

```
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);
    }
```

- Lets also create a constructor for the class (similar to a method). (See image)
  - A constructor is a special method that uses the name of the class itself. (i.e. Drivetrain class has Drivetrain() constructor). Constructors are used when trying to access the class in other files.
  - Methods within this constructor will run as soon as the drivetrain class is called (loaded) in the code.
- Now that we have created the Talons we must initialize them and tell them what port on the roboRIO the they are on.
- Lets initialize (set value of) leftFrontTalon to 'new Talon(0)' this initialized a new talon, leftFrontTalon, in a new piece of memory and states it is on port 0.
  - This is 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.

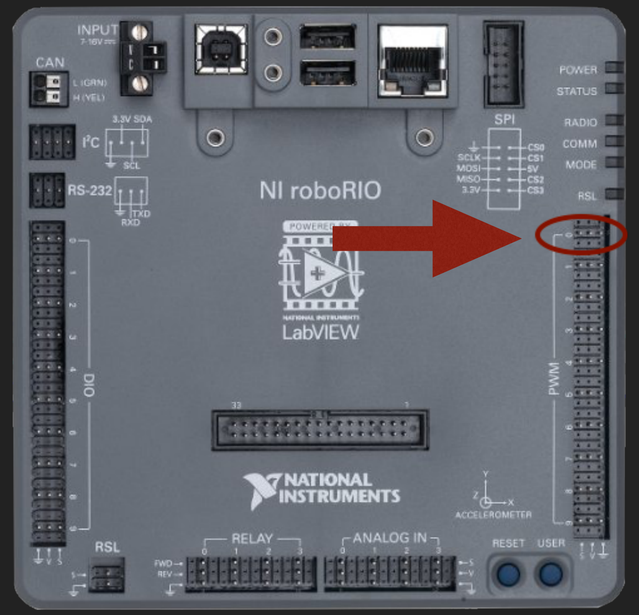


Table of Contents

ROBOT MAP

- 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 a **RobotMap**
- 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 final (can only be changed by the user) global (accessible by all classes) variable. This is naming convention for finals (constants).
- Use the quick fix method to import the class.
- Use the quick fix method to create a constant.
- This will open the RobotMap file.
- Change the 0 to the actual port on your roboRIO **(MAKE SURE THIS IS CORRECT OR YOU COULD BRAKE YOUR ROBOT!)**
- Remember to save! (Ctrl S)
- Repeat these steps for the remaining Talons.

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

2.
```
public Drivetrain() {
    // Talons
    leftFrontTalon = new Talon(RobotMap.DRIVETRAIN_LEFT_FRONT_TALON);
}

// Put methods for controlling
// here. Call these from Comman
```
RobotMap cannot be resolved to a variable

15 quick fixes available:

Import 'RobotMap' (org.usfirst.frc.team3255.robot)

3.
```
Talon(RobotMap.DRIVETRAIN_LEFT_FRONT_TALON);
```
DRIVETRAIN_LEFT_FRONT_TALON cannot be resolved or is not a field

2 quick fixes available:

Create constant 'DRIVETRAIN_LEFT_FRONT_TALON' in type 'RobotMap'

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

Table of Contents

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

```java
public class Drivetrain extends Subsystem {

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

    // Robot Drive
    RobotDrive robotDrive = 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 the
    // following variables to use with your drivetrain subsystem.
    // public static int leftMotor = 1;
    // public static int rightMotor = 2;

    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 CREATE OUR DRIVETRAIN, LETS MAKE A WAY TO CONTROL IT!

WE ARE MISSING A CRUCIAL STEP!

ADDING THE SUBSYSTEM TO ROBOT.JAVA

ADDING THE SUBSYSTEM TO ROBOT.JAVA

- 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 drivetrain and set is value to null.
  - Quick fix -> import Drivetrain
- In the robotInit() method add: drivetrain = new Drivetrain();
  - **IMPORTANT**: This must always be done above OI and Telemetry/ SmartDashboard (if present).
- Now when use this subsystem in commands we must call Robot.drivetrain to get access to it and its methods.

1.
```
public class Robot extends IterativeRobot {

    public static final ExampleSubsystem exampleSubsystem
    public static Drivetrain drivetrain = null;
    public static OI oi;
```

2.
```
public void robotInit() {
    drivetrain = new Drivetrain();

    oi = new OI(); // MUST ALWAYS BE THE LAST SUBSYSTEM

    chooser.addDefault("Default Auto", new ExampleCommand());
    // chooser.addObject("My Auto", new MyAutoCommand());
    SmartDashboard.putData("Auto mode", chooser);
}
```

Table of Contents

BACK TO CONTROLLING IT

SETTING UP JOYSTICKS

CREATING A JOYSTICK

- 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
  - Quick fix 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) (Go back to OI.java when finished)

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

```java
public class OI {

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

```java
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;

    // Joysticks
    public static final int OI_DRIVER_CONTROLLER = 0;
}
```

TIME FOR ROBOTDRIVE

LETS PROGRAM AN ARCADE DRIVE

WHAT IS ROBOTDRIVE?

- The FIRST RobotDrive class has many preconfigured methods available to us including TankDrive, ArcadeDrive, and many alterations of MecanumDrive.
- For our tutorial we will be creating an ArcadeDrive
- An ArcadeDrive drive is a special preconfigured method created in the RobotDrive class by FIRST available to teams.
  - 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 code and adding to it.

SETTING UP A ROBOT DRIVE

- In the same place we created our talons (outside of the constructor) we will create a new RobotDrive
  - Type: RobotDrive robotDrive = null;
- We must also initialize the robotDrive like we did the talons. In the constructor…
  - Type: robotDrive = new RobotDrive(leftFrontTalon, leftBackTalon, rightFrontTalon,rightBackTalon).
  - The RobotDrive constructor we are using requires parameters of 4 motors in the order we used above.
    - If you are only using 2 motors, just use the left and right (in that order).

CREATING THE ARCADEDRIVE METHOD

- Now its time to make an arcadeDrive from our robotDrive.
  - Lets 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: robotDrive.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.
  - It should look something like this (see image):

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

- At this point you could instead create a tank or mecanum drive, however implementation differs slightly.
- To do so do robotDrive.tankDrive
- Or MecanumDrive

Table of Contents

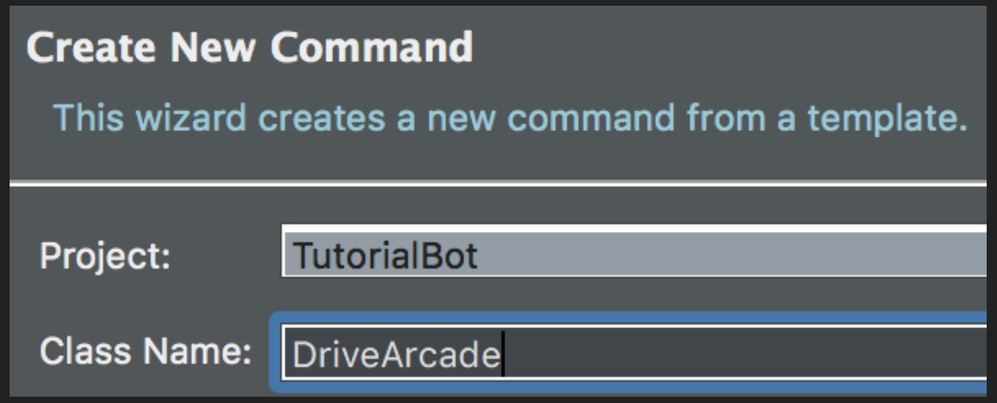
CREATING THE ARCADEDRIVE COMMAND

- Methods tell the robot what it can to 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.
- Lets create a command called DriveArcade that calls arcadeDrive

CREATING A NEW COMMAND

- Select .command create a new command (ctrl N -> WPILib -> Command)
- Under Class Name type: DriveArcade
- Open DriveArcade

1.



2.

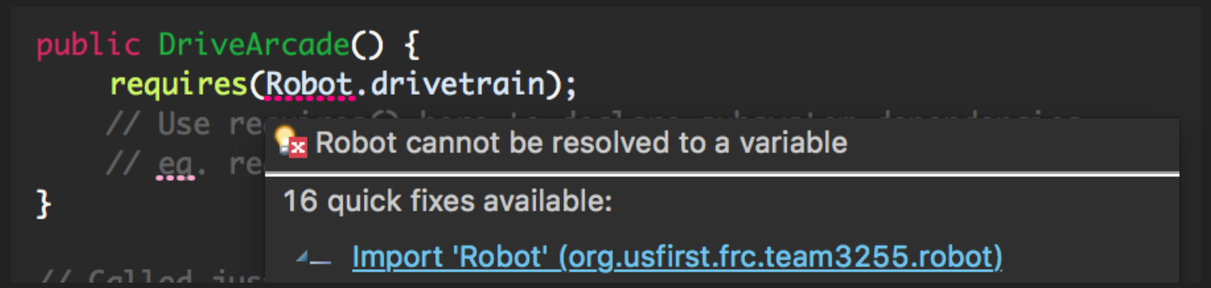
```java
public class DriveArcade extends Command {

    public DriveArcade() {
        // Use requires() here to declare subsystem dependencies
        // eg. requires(chassis);
    }

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

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

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

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

    // Called when another command which requires one or more of the same
    // subsystems is scheduled to run
    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

DRIVEARCADE COMMAND

- In the constructor DriveArcade() type: requires(Robot.drivetrain).
  - This means that this command will stop all other commands from accessing drivetrain while this command is using it.
  - Quick fix -> import ‘Robot’. (Only if you have an error)
    - **IMPORTANT:** Be sure to import the one with org.usfirst.frc.team



IN EXECUTE()...

- We will create 2 variables of type double called moveSpeed and rotateSpeed.
- We want these variables to be the value of the axises of the controller we are using to drive the robot. So we will set them equal to that.
- We use negative in front since on our personal controllers since up reads -1 (this may differ.)
- The joystick we created has a special FIRST created method called getRawAxis. This will get the position value of the axis as you move it. The method takes parameter: axis number. (This can be found in the driver station 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.

```
protected void execute() {
    double moveSpeed = -Robot.oi.joystick.getRawAxis(RobotMap.JOYSTICK_MOVE_AXIS);
    double rotateSpeed = Robot.oi.joystick.getRawAxis(RobotMap.JOYSTICK_ROTATE_AXIS);

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

IN END()...

- 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.
- In interrupted() we will make it call end.
  - This makes the end method get called if the command gets interrupted.

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

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

YOUR CODE SHOULD LOOK SIMILAR TO THIS IN DRIVEARCADE.JAVA

```
public class DriveArcade extends Command {

    public DriveArcade() {
        // Use requires() here to declare subsystem dependencies
        // eg. requires(chassis);
        requires(Robot.drivetrain);
    }

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

    // Called repeatedly when this Command is scheduled to run
    protected void execute() {
        double moveSpeed = -Robot.oi.joystick.getRawAxis(RobotMap.JOYSTICK_MOVE_AXIS);
        double rotateSpeed = Robot.oi.joystick.getRawAxis(RobotMap.JOYSTICK_ROTATE_AXIS);

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

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

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

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

INITDEFAULTCOMMAND()

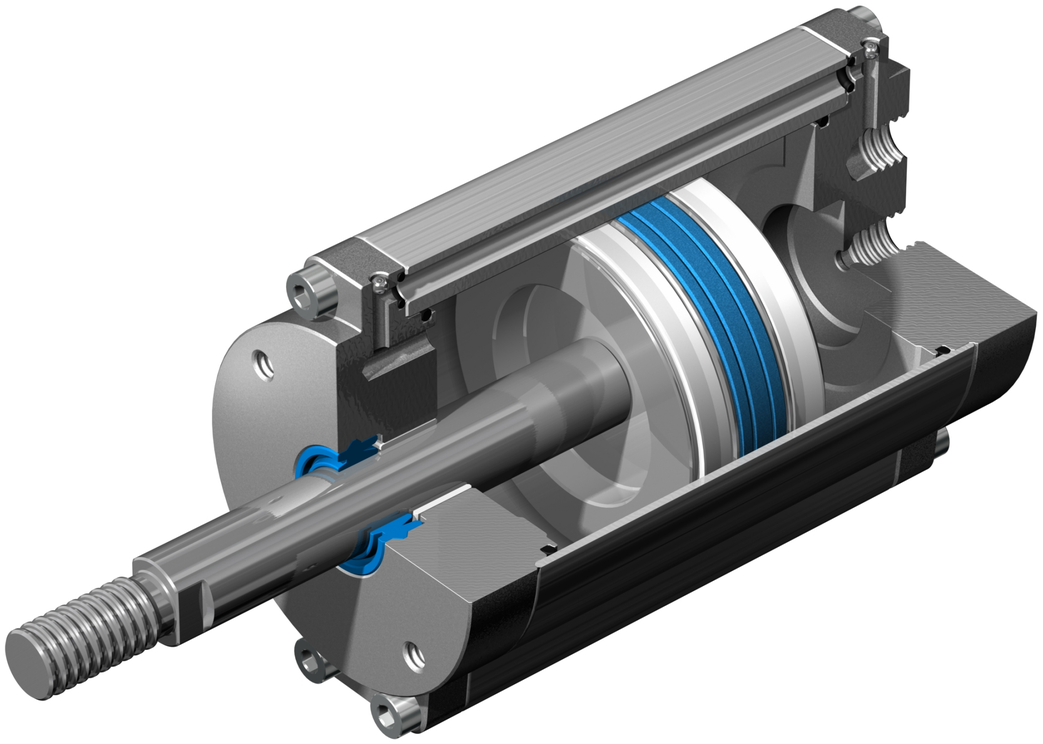
- Commands within this method run when the robot is enabled and run when no other commands require that subsystem.
  - This is why we write requires in the commands we create, it stops other commands using that subsystem when that command is ran.
- Back in Drivetrain in the initDefaultCommand() method we will put our newly created command.

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

  - Quick fix import the command DriveArcade.

DEPLOY!

YOUR CODE IS NOW READY TO RUN. DEPLOY CODE AND TRY MOVING THE JOYSTICKS ON YOUR CONTROLLER WHEN ENABLED.



PART 2:
CHECK THE AIR PRESSURE

USING PNEUMATICS

STATE CHANGE!

CREATING THE METHODS TO USE 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 to run these methods.

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

IN N OUT

CREATING THE COMMANDS TO USE PNEUMATICS

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...

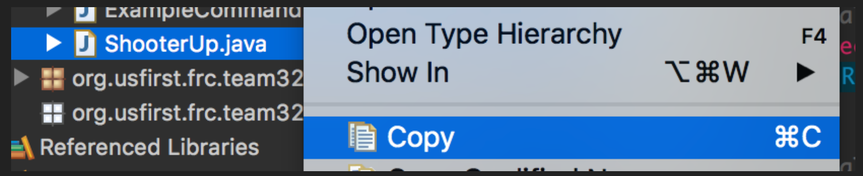
```java
public class ShooterUp extends Command {

    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;
    }
```

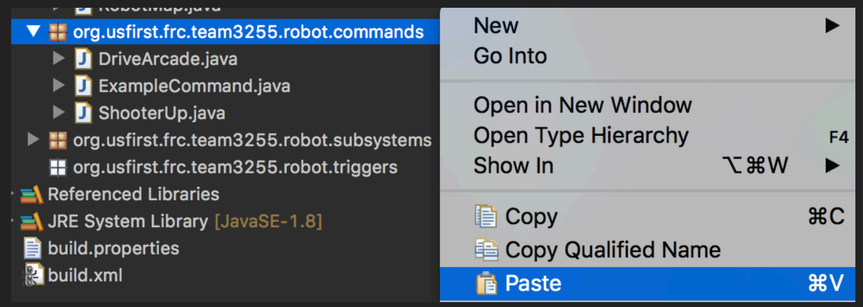
TIP: COPYING FILES

- 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();).

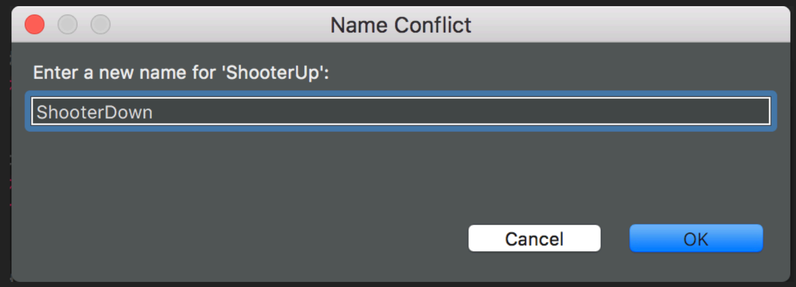
1)



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();
    }
```

Table of Contents

PUSHING BUTTONS

MAPPING COMMANDS TO BUTTONS

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

DEPLOY!

YOUR CODE IS NOW READY TO RUN. DEPLOY CODE AND TRY PRESSING THE BUTTONS ON YOUR CONTROLLER TO DEPLOY AND RETRACT THE PNEUMATIC CYLINDER.

THIS SECTION HAS NOT YET BEEN CREATED

PART 3:
GIVING FEEDBACK

USING SENSORS AND SWITCHES

Table of Contents

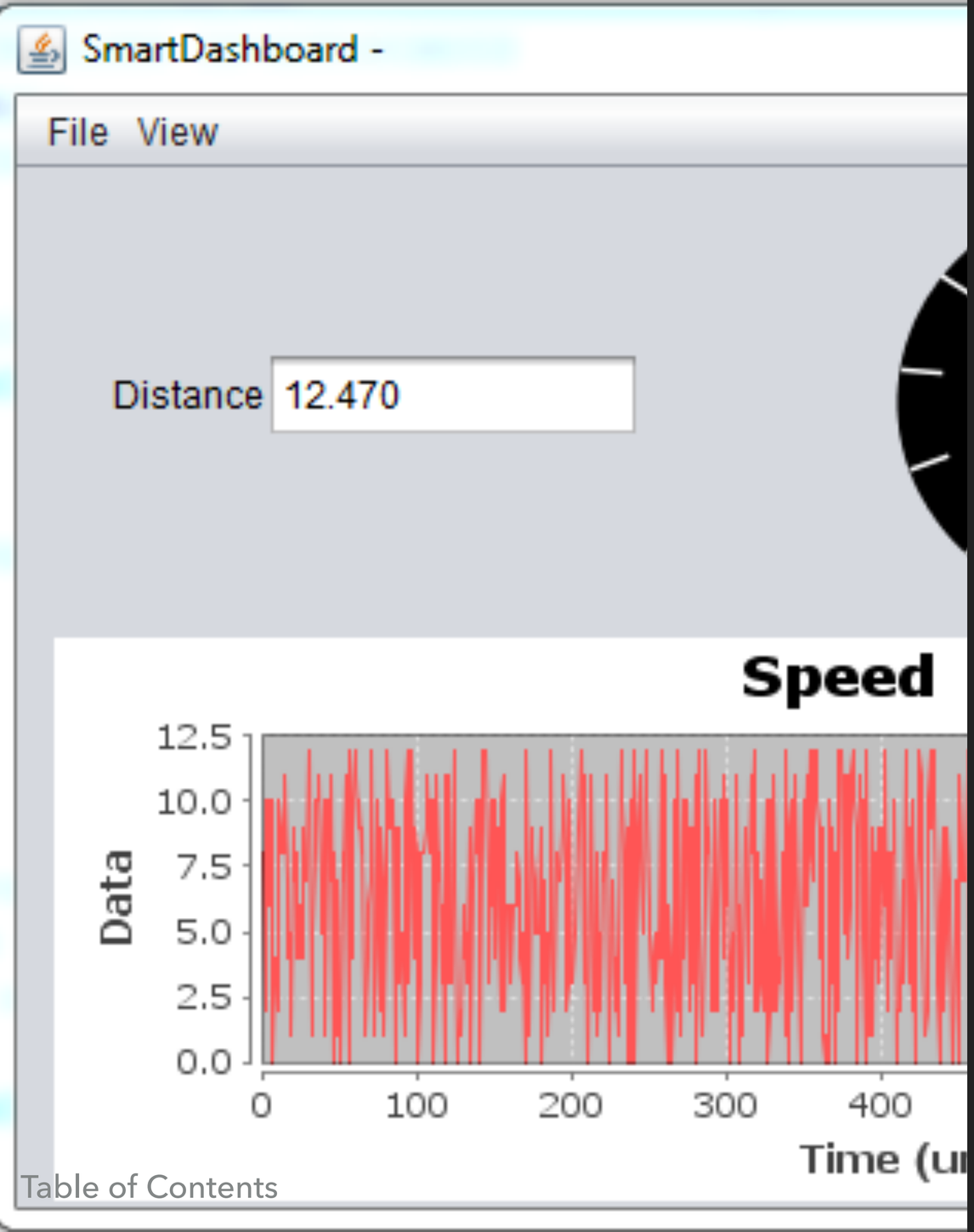


Table of Contents

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

THIS SECTION HAS NOT YET BEEN CREATED

PART 4:
GETTING FEEDBACK

USING THE SMARTDASHBOARD

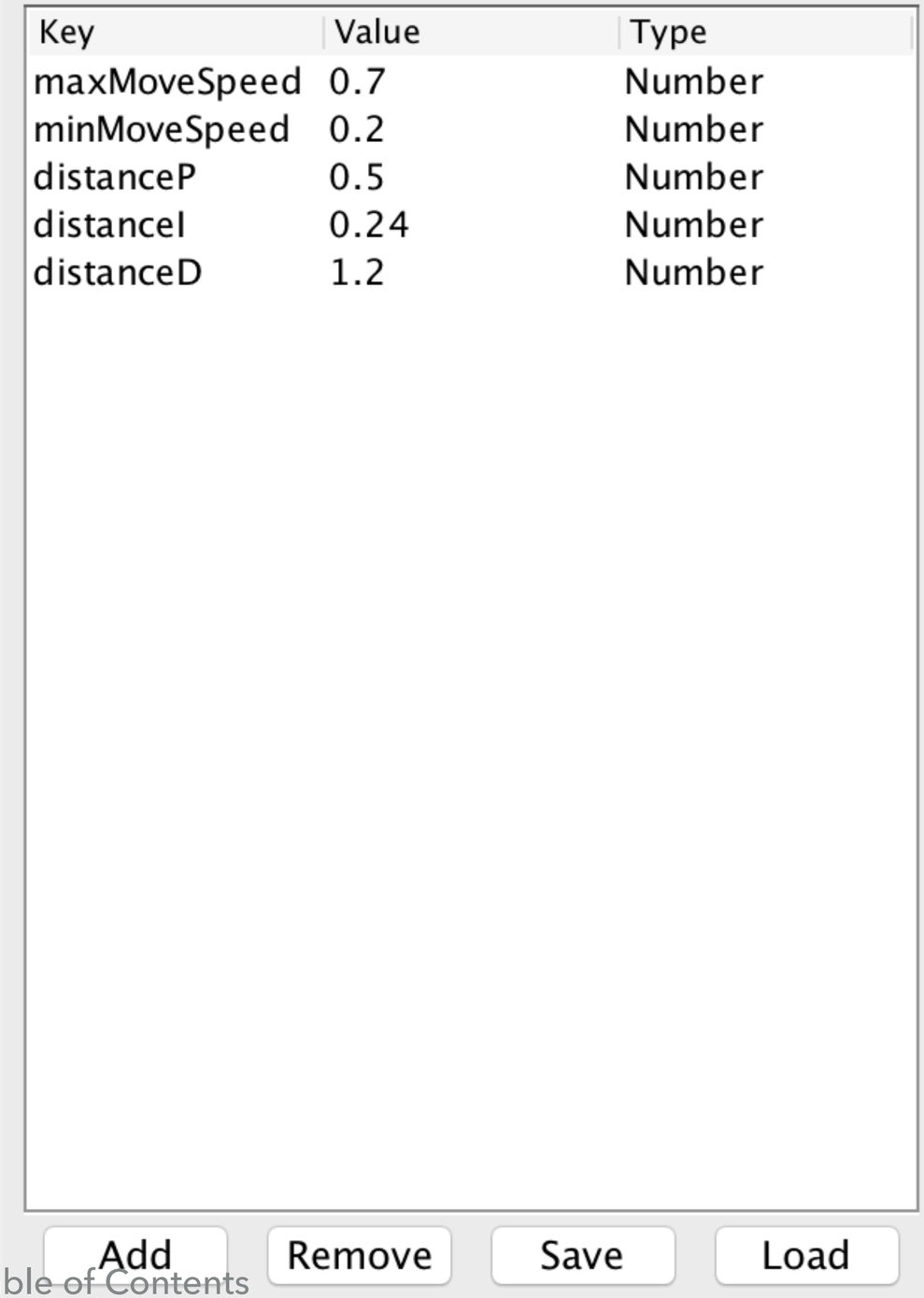
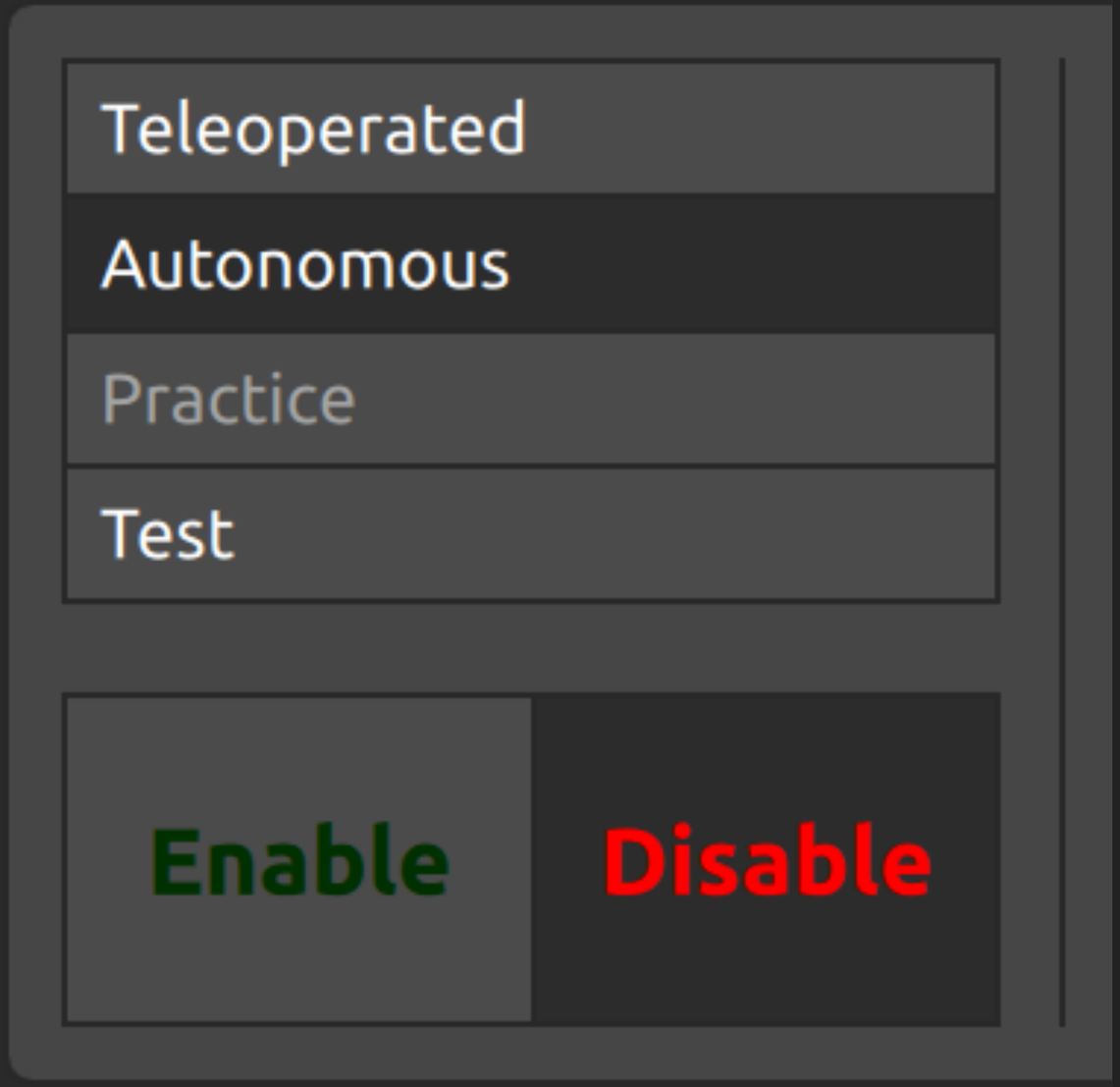


Table of Contents

THIS SECTION HAS NOT YET BEEN CREATED

PART 5:
SETTING SETTINGS

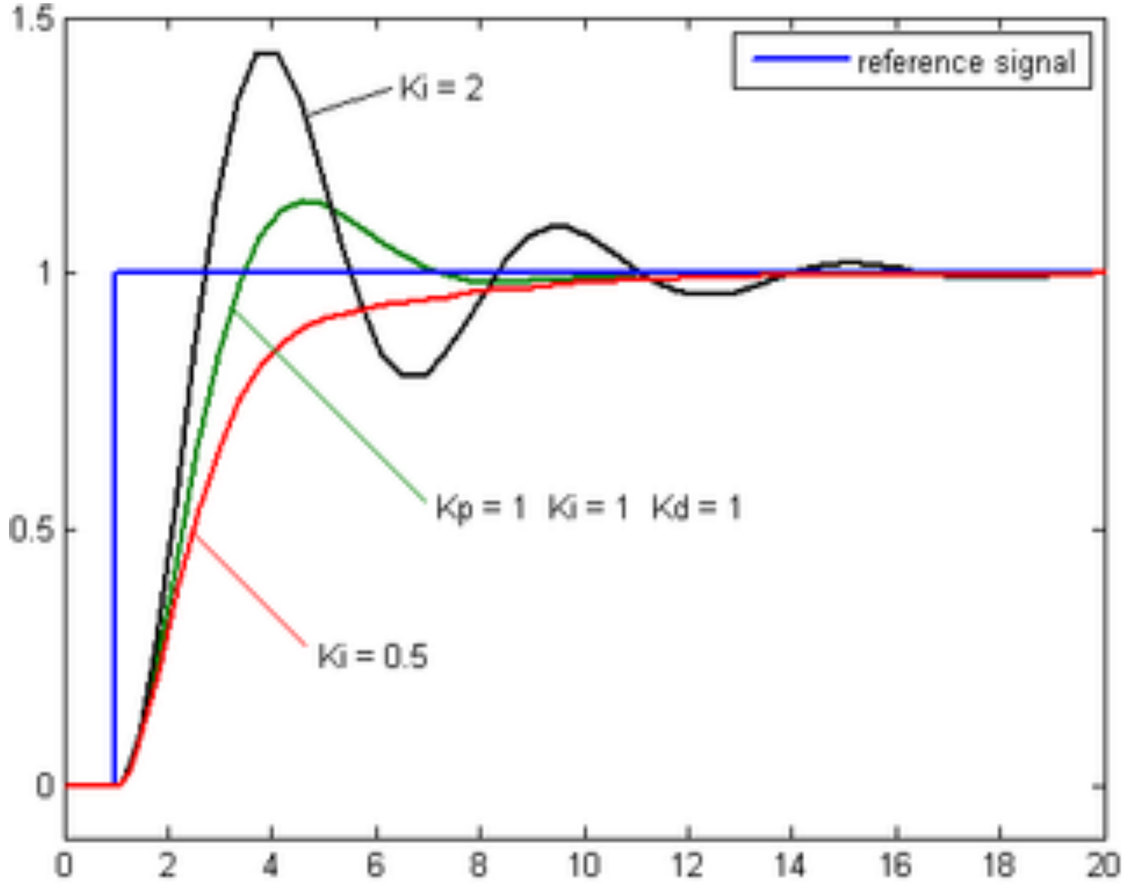
USING ROBOTPREFERENCES



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