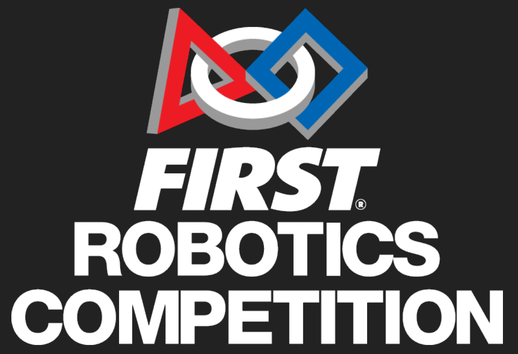


FRC JAVA PROGRAMMING

CREATED BY TAYLER UVA
FRC TEAM 3255 - THE SUPERNURDS

GITHUB.COM/TAYLERUVA
SUPERNURDS.COM

Updated for the 2019 Season

TABLE OF CONTENTS

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:

SETUP AND BASICS OF WPILIB AND PROGRAMMING

LEARNING THE BASICS

PART A:
THE BRAINS OF THE BOT

THE ROBORIO

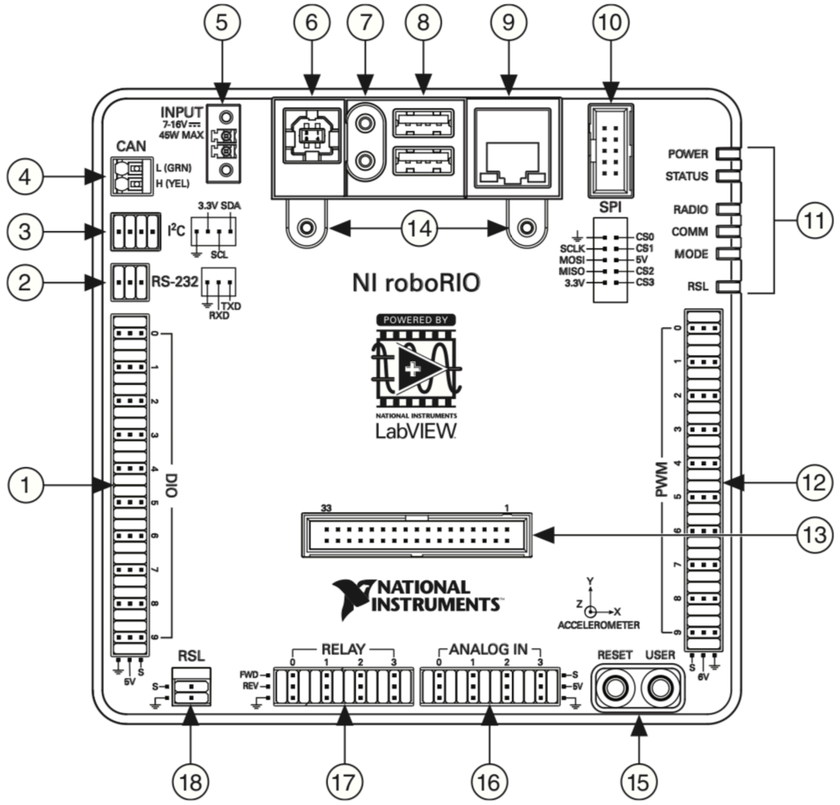
THE ROBORIO

- The roboRIO is the brain of 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

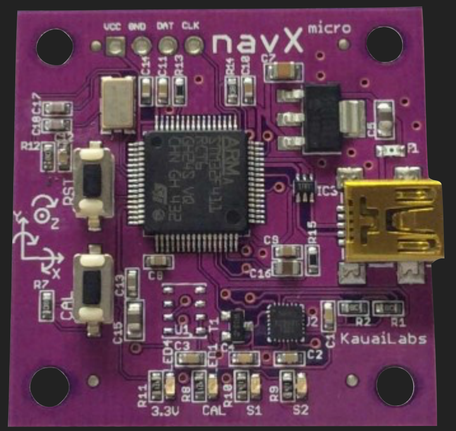
Table of Contents

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

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 at least one class but can be made 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.

CLASSES

- Classes can also have a **constructor** which is a special type of **method** having 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
  - They can be called again if the class is programmed to be unloaded (destroyed) and reloaded

VARIABLES

- Variables are objects that contain data, they are characterized by data types
- Variables are assigned names and data types on creation
  - Names be anything with the exception of pre-existing keywords such as `public` or `int`
- Data types as the name suggest define what type of data is being store 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 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. Copy 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 is 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.

- 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
    - 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

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

PART D:

MAKING FRC PROGRAMMING EASY

WPILIB PROGRAMMING BASICS

Table of Contents

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

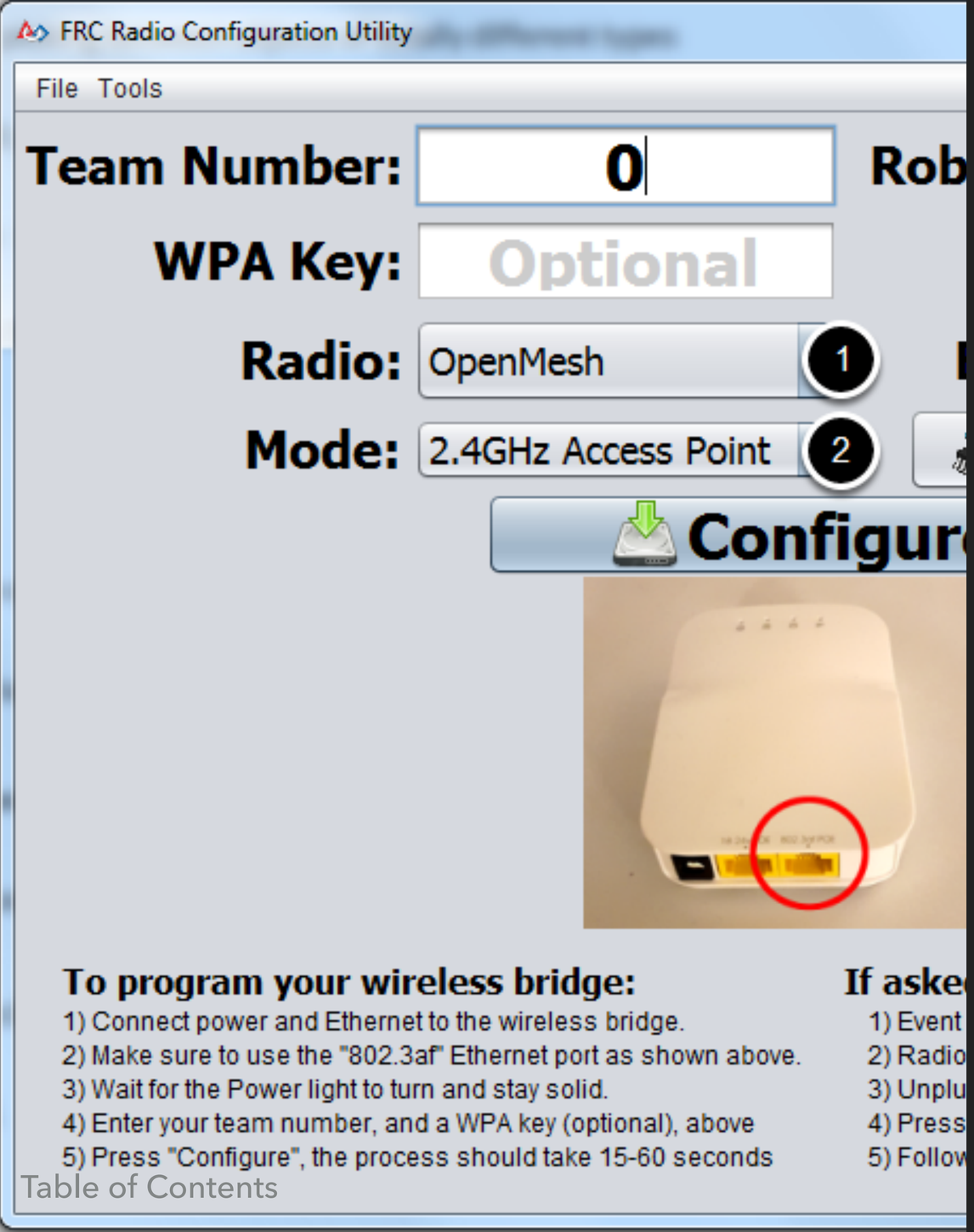
Visual Studio Code 

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



SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

THIS SECTION HAS NOT YET BEEN CREATED

PART F:
GOING WIRELESS!

SETTING-UP THE ROBOT RADIO

PARTS 1–7:
PROGRAMMING IN JAVA WITH WPILIB

PROGRAMMING FOR AN FRC ROBOT

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

PART 1:
LETS GET MOVING

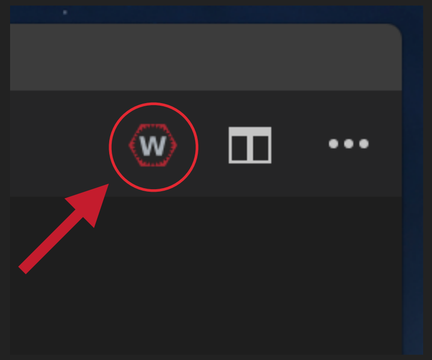
CREATING A BASIC DRIVING ROBOT

WHAT IS IN THIS ROBOT?

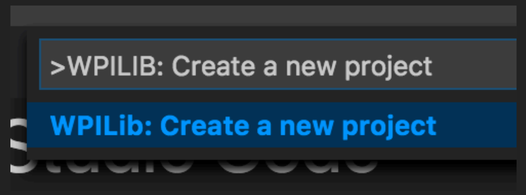
CREATING THE DRIVETRAIN SUBSYSTEM

CREATING A NEW PROJECT – VISUAL INSTRUCTIONS

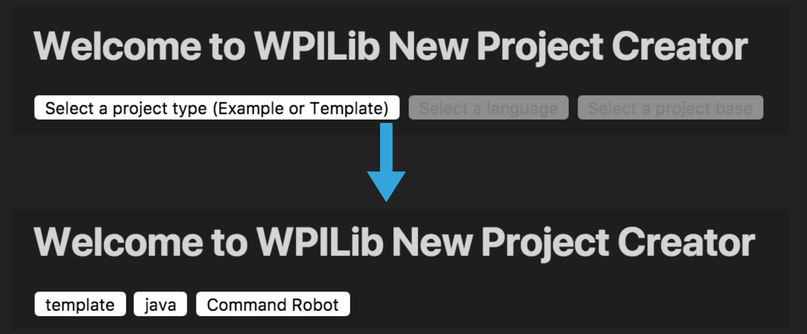
1.



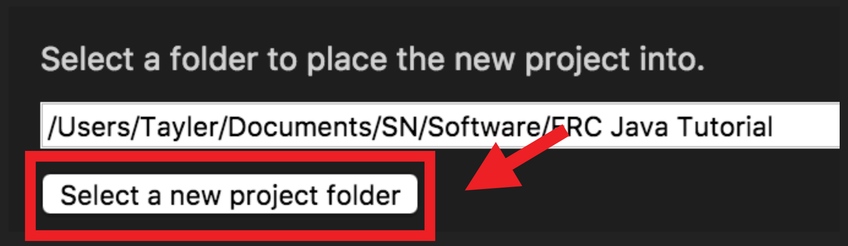
2.



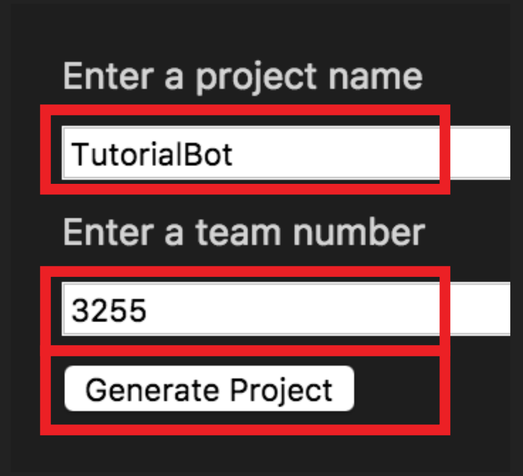
3.



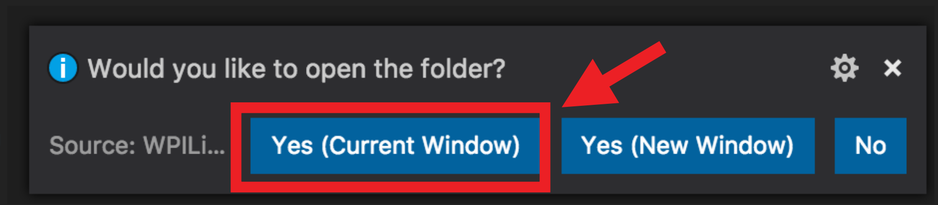
4.



5.



6.

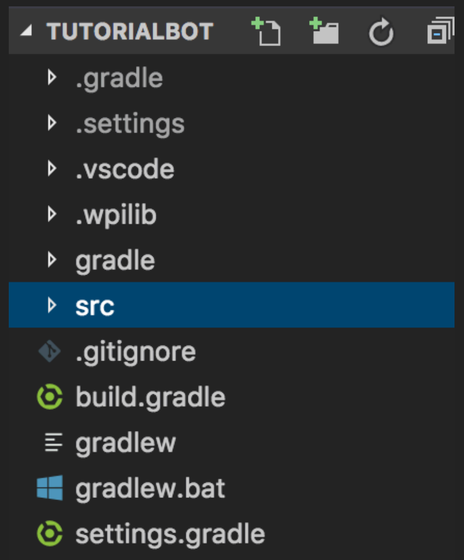


CREATING A NEW PROJECT – TEXT INSTRUCTIONS

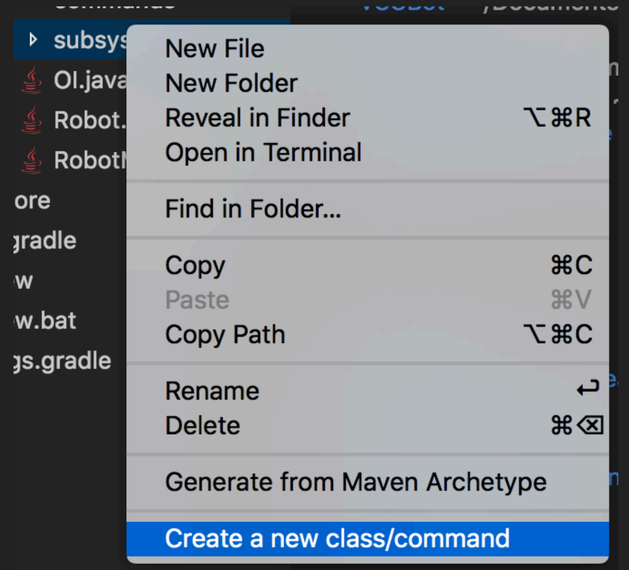
- Select the W icon from the tab bar or use the shortcut by holding down **Ctrl+Shift+P** at the same time. (Replace ctrl with cmd on macOS)
- Type and hit enter or select **WPILib: Create a new project**
- Click **Select a Project Type** and choose **Template**
- Click **Select a Language** and choose **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 THE DRIVETRAIN SUBSYSTEM – VISUAL INSTRUCTIONS

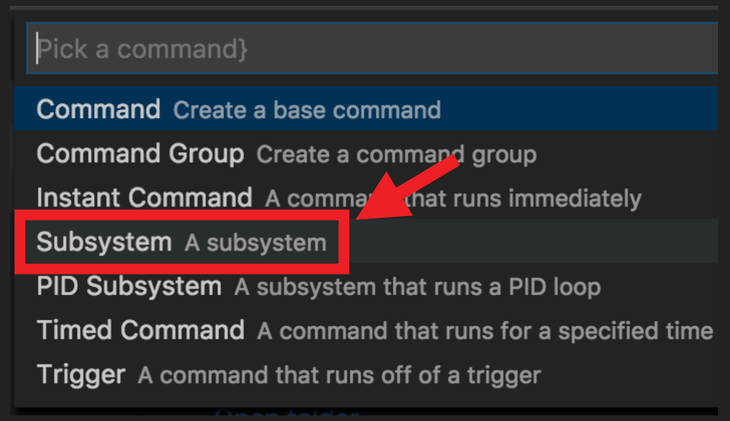
1.



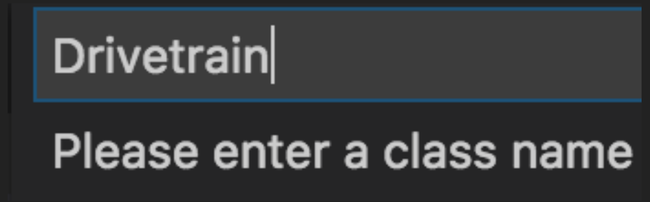
2.



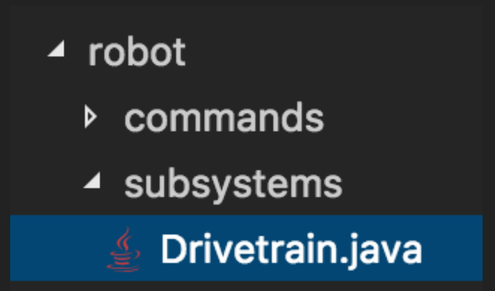
3.



4.



5.



CREATING A 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**

SLIDES BEYOND THIS POINT HAVE NOT BEEN UPDATED

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

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;
}
```

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

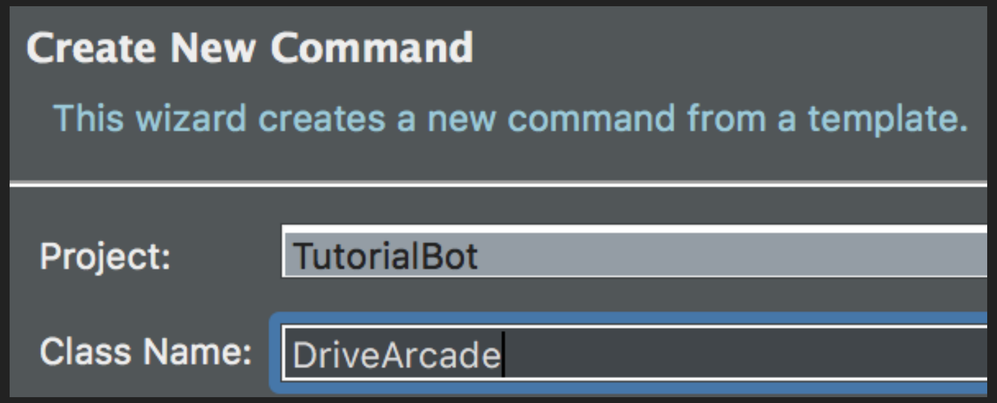
CREATING THE ARCADEDRIVE COMMAND

- 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.

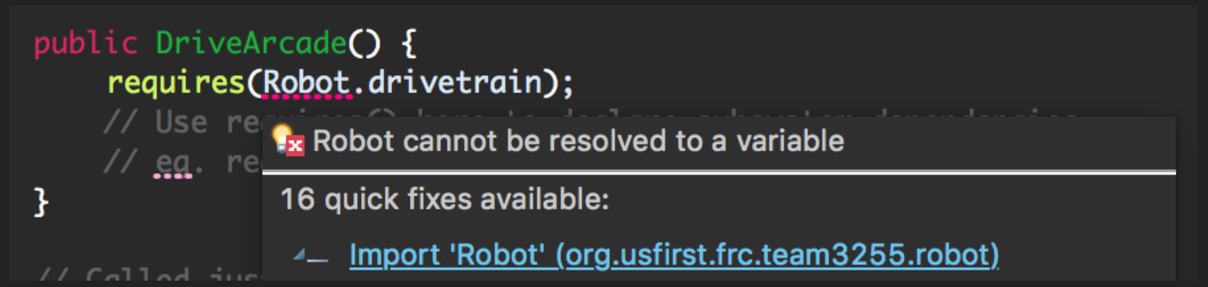
```
public class DriveArcade extends Command {

    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()

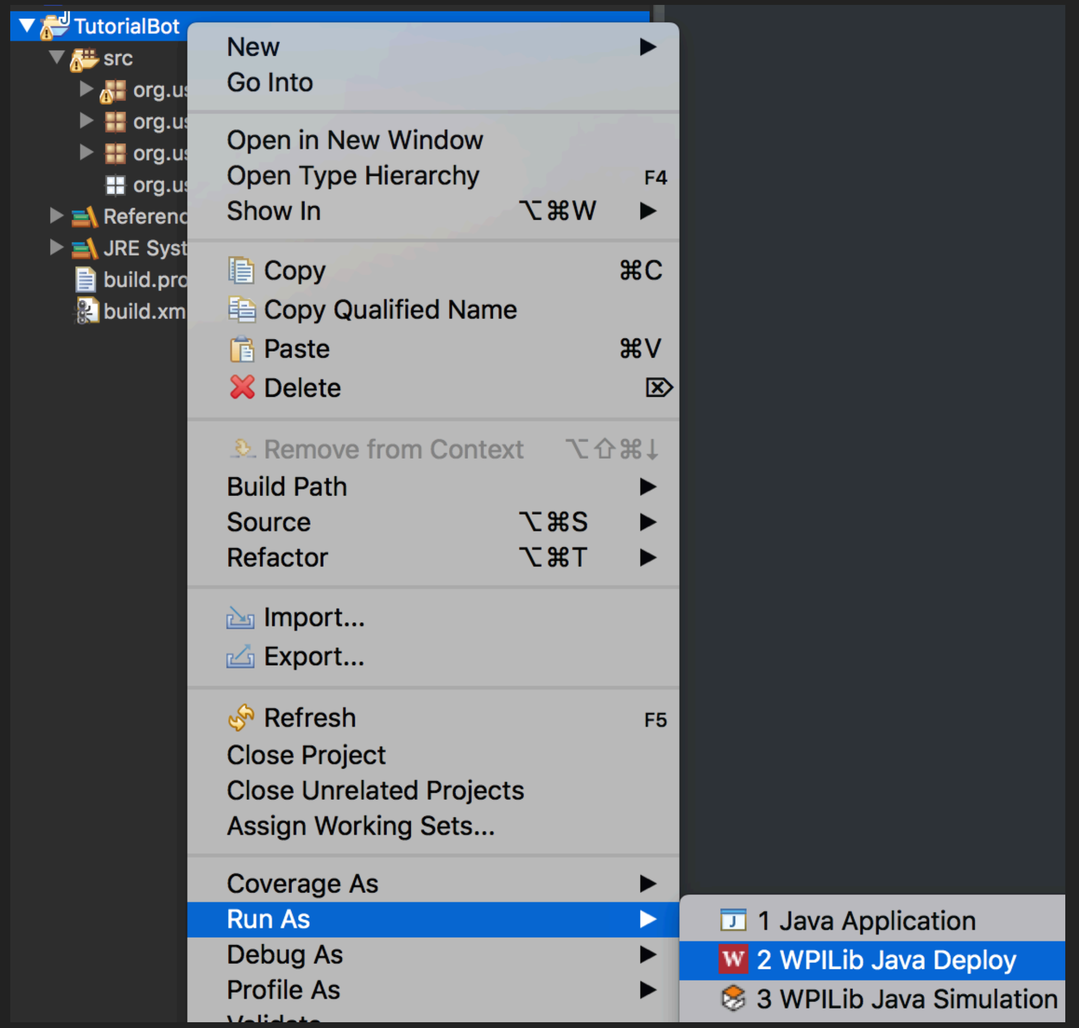
- Commands within this method run when the robot is enabled and run when no other commands require that subsystem.
  - This is why we write requires in the commands we create, it stops other commands using that subsystem when that command is ran.
- Back in Drivetrain in the initDefaultCommand() method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

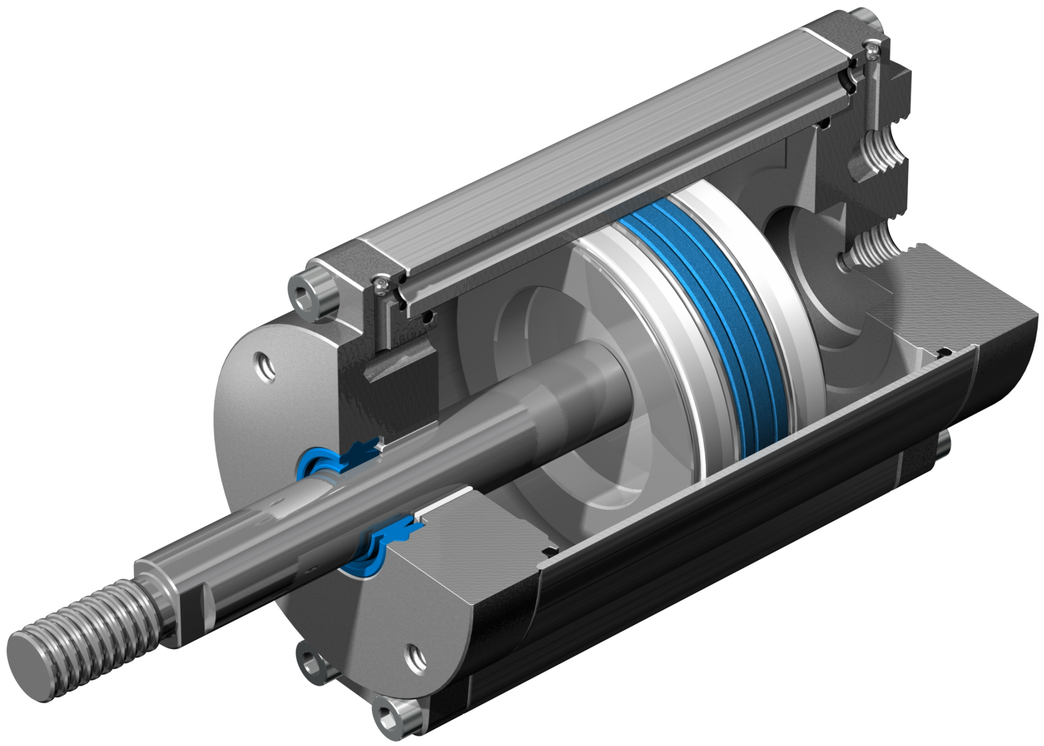
```
public void initDefaultCommand() {
    // Set the default command for a subsystem here.
    setDefaultCommand(new DriveArcade());
}
```

  - Quick fix import the command DriveArcade.

DEPLOYING CODE

- To deploy code, first make sure your computer either connect to the robot over USB, ethernet, or the radio.
- Make sure your team number in Eclipse is set to the same team number your roboRIO was programmed for.
  - You can check/change this in Eclipse preferences under WPILib Preferences.
- Select your project (TutorialBot) and right click it, then select **Run As** -> WPILib Java **Deploy**





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 to 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...

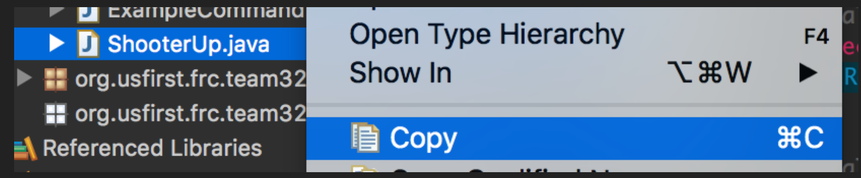
```java
public class ShooterUp extends Command {

    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;
    }
```

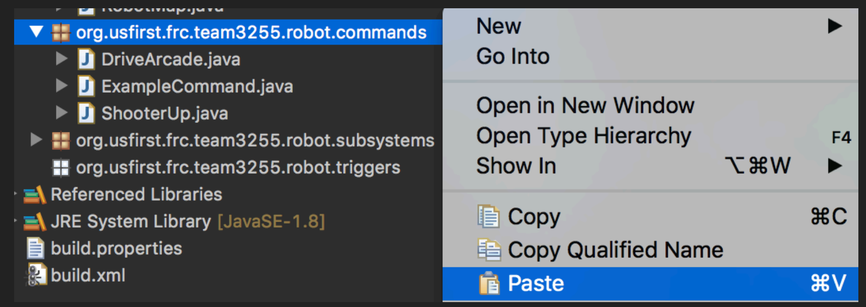
TIP: COPYING FILES

- 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();).

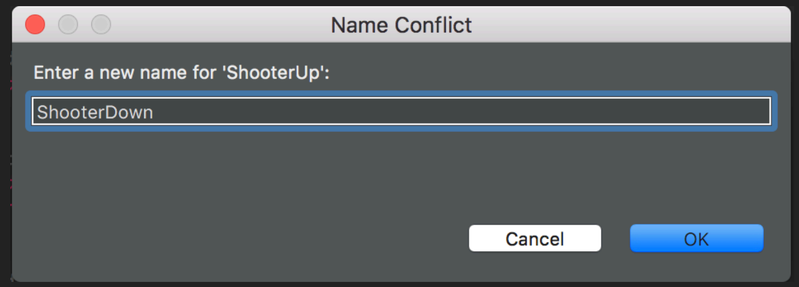
1)



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.

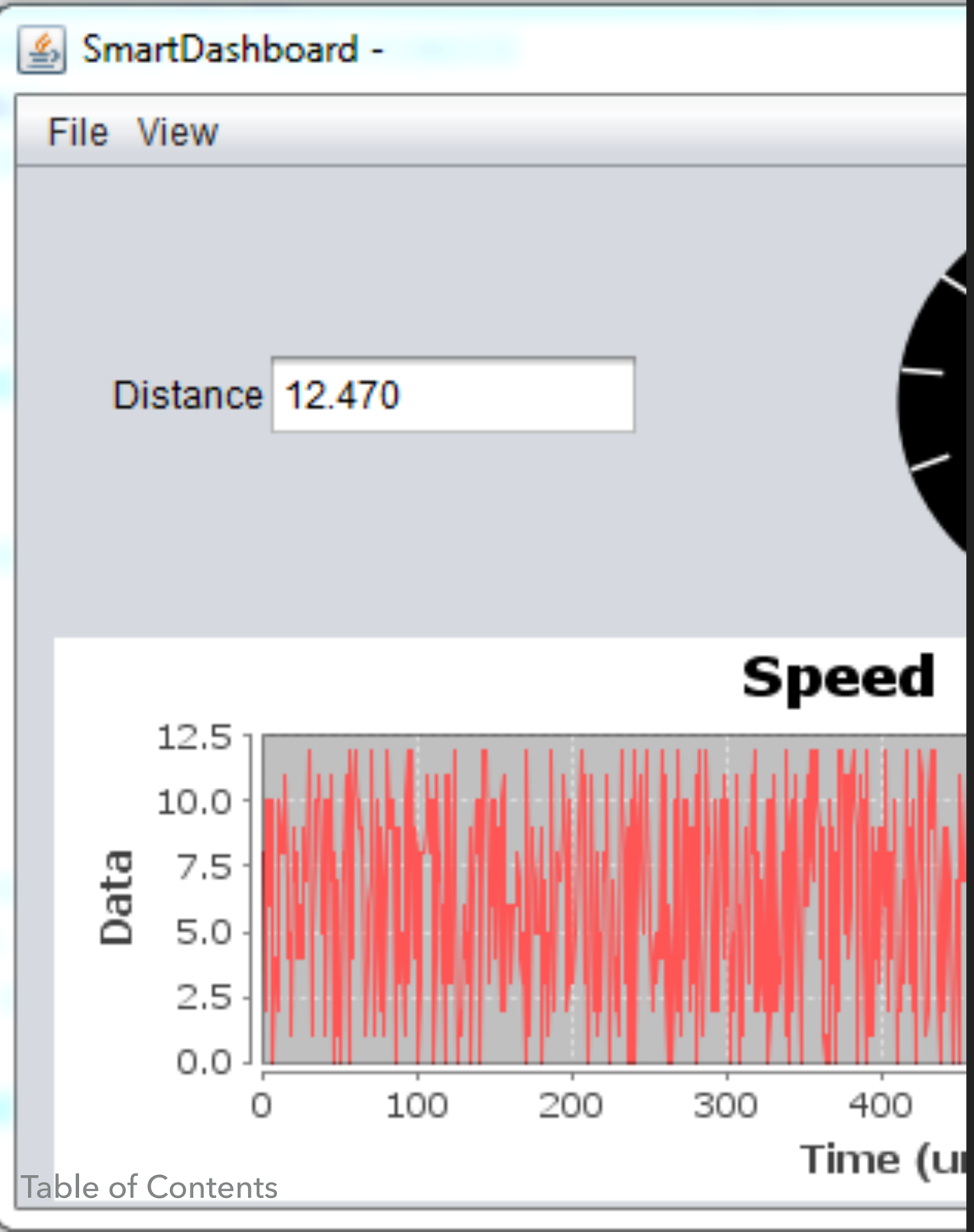
- **TIP:** You can read in detail what every method of the FIRST code does by reading the Java Docs

THIS SECTION HAS NOT YET BEEN CREATED

PART 3:
GIVING FEEDBACK

USING SENSORS AND SWITCHES

Table of Contents



THIS SECTION HAS NOT YET BEEN CREATED

PART 4:
GETTING FEEDBACK

USING THE SMARTDASHBOARD

Table of Contents

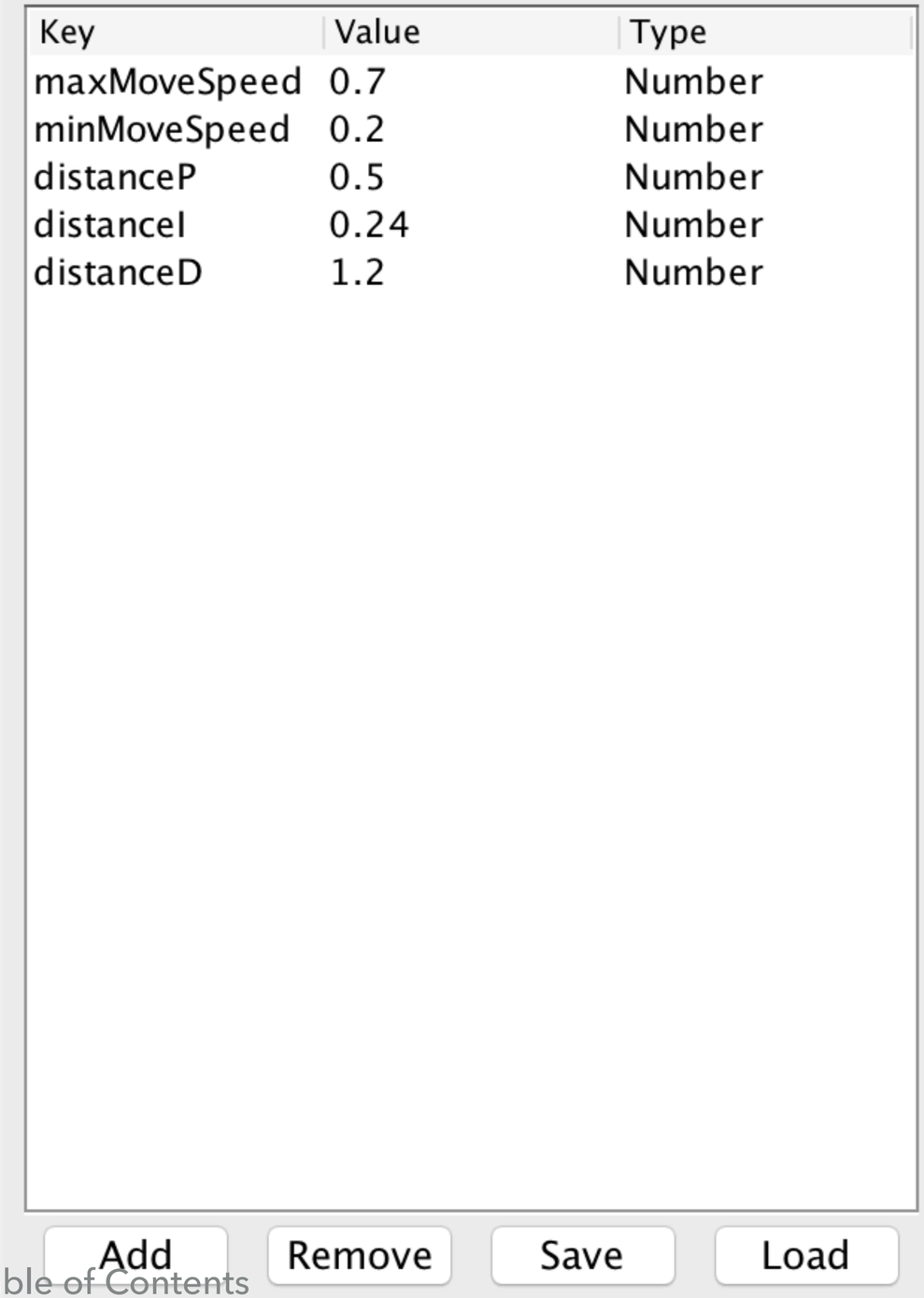


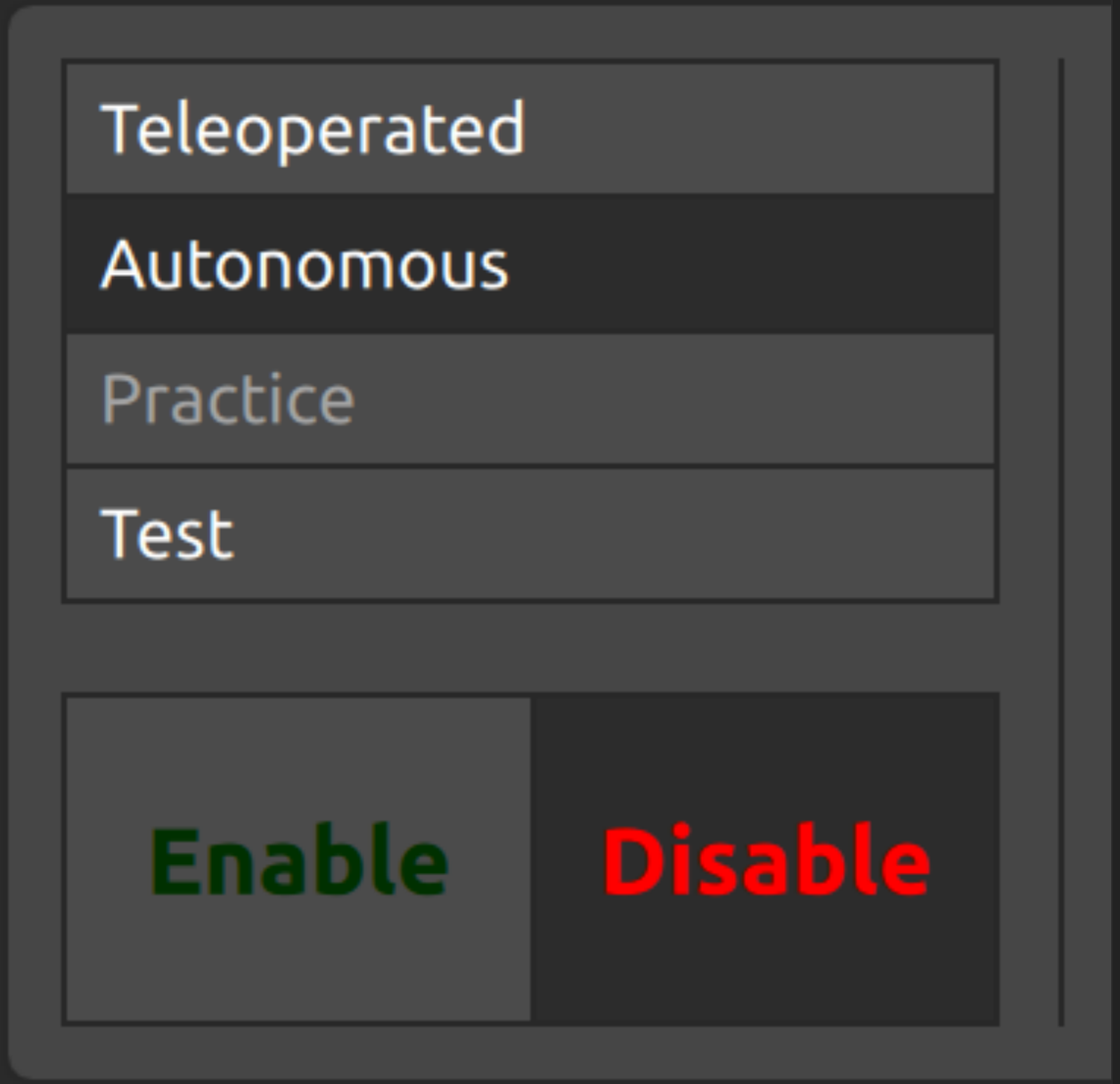
Table of Contents

SUPERNURDS
NEVER UNDERESTIMATE ROBOTIC DOMINATION

THIS SECTION HAS NOT YET BEEN CREATED

PART 5:
SETTING SETTINGS

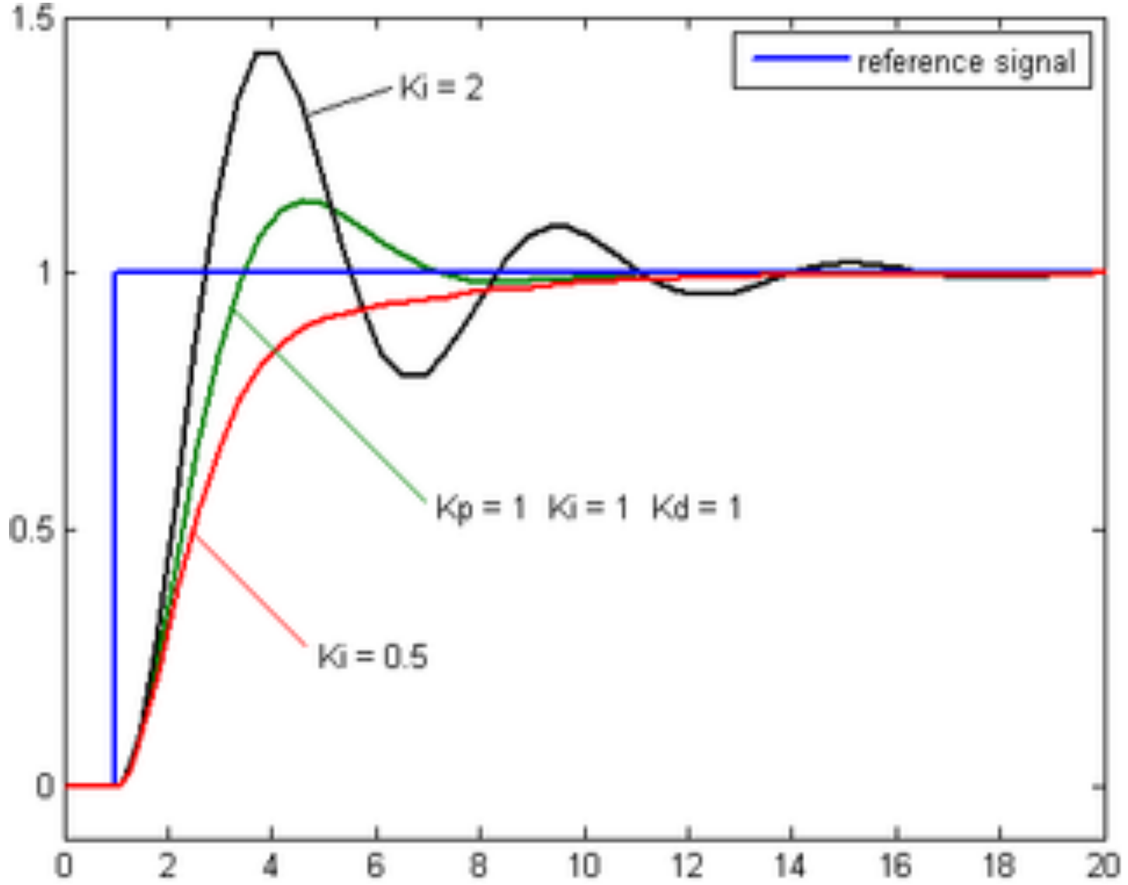
USING ROBOTPREFERENCES



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