2024 *FIRST*® Robotics Competition

KitBot Java Software Guide

1 Contents

2 Document Overview

This document will take you through how to get your 2024 KitBot up and running using the provided Java example code. To avoid content duplication this document frequently links to WPILib documentation for accomplishing specific steps along the way. In addition to getting you up and running with the provided code, this document will walk through the structure of that code so you can understand how it operates. Finally, we'll walk through some of the most likely changes you may wish to make to the code and provide concrete examples of how to make those modifications.

To get started with the example code, or to make some of the modifications described, minimal understanding of Java is required. The code and modification examples provided will likely provide enough of a pattern to get you going. To understand the walkthrough, or to make modifications not described in this document, a more thorough understanding of Java is likely required. The Intro to Programming module on Thinkscape is a great way to learn about Java using WPILib and the Romi or XRP robot platforms. For other options, check out the links on the Zero-to-Robot Introduction page.

This document, and the provided example code, assumes the use of the SPARK MAX controllers provided in the rookie Kickoff Kit.

3 Getting Started with your KitBot code

3.1 Wiring your robot

Use the WPILib Zero-to-Robot wiring document to help you get your robot wired up. Some notes specific to the 2024 KitBot:

- The 2024 KitBot does not utilize pneumatics. You can skip instructions regarding the Pneumatic Hub/Pneumatics Control Module unless you are adding pneumatics to the design.
- In order to use the same IDs for PWM and CAN operation, the 2024 KitBot code does not utilize PWM port 0. Either wire the PWM ports according to the IDs in Constants.java (Left = 1,2, Right = 3,4) or modify the constants to reflect your wiring.
- The 2024 KitBot contains two additional motors not included in the basic wiring document. Wire these in the same manner as the drivetrain motors. If using PWM, connect the Feeder motor (closer to the center of the robot) to PWM port 5 and the Launcher motor (the motor closer to the outside of the robot) to PWM port 6.

3.2 Configuring hardware and development environment

Before you are able to load code and test out your robot, you will need to configure your hardware (roboRIO, radio, etc.) and get your development environment set up. Follow the WPILib Zero-to-Robot guide steps 2 through 4 to get everything set up and ensure you can deploy a basic robot project.

If using PWM, make sure all 6 SPARK MAXs are in "Brushed" mode. When powered the LED should blink yellow or blue, not magenta or cyan. To change the mode, you can either hold the Mode button down for 3 seconds or use the USB connection and REV Hardware client. You may also wish to check the Idle modes, brake or coast. The Feeder and Launcher motors are recommended to be set to coast mode (blinking yellow). To change the Idle mode, press the Mode button briefly (less than 3 seconds) or use the USB connection and REV Hardware Client. There is no specific recommendation for the drivetrain motors, but you likely want all 4 drivetrain motors to match, you may wish to try driving around with each setting to decide what you prefer.

3.3 Opening the 2024 KitBot Example

The 2024 KitBot example code is provided in individual zip files for each language on the KitBot webpage. To open the Java code:

1. Download and unzip the Java example code. Make sure to unzip or copy to a permanent location, not in a temporary folder.
2. Open **WPILib VS Code** using the Start menu or desktop shortcuts.
3. In the top left click **File->Open Folder** and browse to the "Java" folder inside of the unzipped example code then click **Select Folder.**

3.4 Changing to CAN control

If you have wired your SPARK MAX motor controllers using CAN, you will need to do some further configuration and code modification before proceeding. If you are using PWM, skip down to Section 3.5 to deploy and test the code.

3.4.1 Configuring the SPARK MAXs

Before using the SPARK MAXs with CAN control, they each need to be assigned a unique ID. Because your SPARKs all start with the same ID you may wish to unplug the CAN bus from each device while you update and assign an ID.

1. Install the REV Hardware Client
2. With the robot powered off, connect a USB cable between the computer and the SPARK MAX USB port. Leaving the robot powered off ensures only the single SPARK MAX is powered and avoids changing the IDs on unintended devices.
3. Update the firmware on the SPARK MAX
4. Set the CAN ID and Motor Type (you can skip the current limit) and save the settings
    a. CAN IDs for each device can be found in Constants.java. You can either set the devices to match these IDs or set the IDs as desired (some teams set the CAN ID = the channel number the device is attached to on the PD) and then update these constants.
    b. Note: If you wish to “Spin the motor” as described on that webpage, make sure the robot is in a safe state to do so (wheels not touching the ground or table).
5. Repeat for all 6 devices on the robot.
6. While not required, if using the REV PDH you may wish to check that it has the latest firmware at this time as well. Do not change the ID of the PDH off of the default, each device type has a separate ID space and your PDH will not conflict with your SPARK MAX even if set to the same ID.

Now that all your devices are configured, you can do a preliminary check that your CAN bus is wired properly using the REV Hardware client. While plugged into any REV device on your CAN bus with a USB cable, power on the robot and you should see all the other devices listed in the left pane of the REV Hardware Client, under the CAN Bus heading. If you don’t see all of the devices, you likely have one or more issues with your CAN bus wiring:

1. Verify that your CAN bus starts with the roboRIO and ends with a 120 ohm resistor, or the built in terminator of a Power Distribution Hub or Power Distribution Panel (with the termination set to On using the appropriate jumper or switch.
2. Check that your CAN bus connections all match yellow-yellow and green-green.
3. Check that all CAN wire connections are secure to each other and that the connectors are securely installed in each SPARK Max
4. If you’re still having trouble, moving the USB connection around to different devices and seeing what each device can “see” on the bus can help pinpoint the location of an issue.

3.4.2 Installing REVLib

The software library for the SPARK MAX in CAN mode is provided by the vendor (REV Robotics). The 3rd party library configuration is already included in the project, but you will have to install the library itself. There are two ways you can do so:

1. **Recommended** Install the library offline – This will ensure that the library persists on your machine even if you don't build new code for a while (online installations can be cleaned up automatically by Gradle).
    a. Download the latest version of REVLib using the link from the REV documentation.
    b. Unzip into the *C:\Users\Public\wpilib\2024* directory on Windows and *~/wpilib/2024* directory on Unix-like systems.
2. Install Online - While the computer is connected to the Internet, click the WPILib icon in the top right of the VSCode window to bring up the WPILib extension prompt, then start typing "Build Robot Code" and select that option when it appears. This will automatically fetch the library online.

3.4.3 Updating the Code

The code is mostly in place in the project to switch from PWM to CAN control, you will just need to make a few edits to switch over.

1. In RobotContainer:

```
//import frc.robot.subsystems.PWMDrivetrain;
//import frc.robot.subsystems.PWMLauncher;
import frc.robot.subsystems.CANDrivetrain;
import frc.robot.subsystems.CANLauncher;
```

```
public class RobotContainer {
  // The robot's subsystems are defined here.
  //private final PWMDrivetrain m_drivetrain = new PWMDrivetrain();
  private final CANDrivetrain m_drivetrain = new CANDrivetrain();
  //private final PWMLauncher m_launcher = new PWMLauncher();
  private final CANLauncher m_launcher = new CANLauncher();
```

    a. Uncomment the import statements for the CANDrivetrain and CANLauncher. You can comment out or remove the statements for the PWM subsystems if you wish.
    b. Comment out the member variable declaration lines for the PWM subsystems and uncomment the ones for the CAN subsystems.

2. In LaunchNote and PrepareLaunch:

```
import edu.wpi.first.wpilibj2.command.CommandBase;
//import frc.robot.subsystems.PWMLauncher;
import frc.robot.subsystems.CANLauncher;
import static frc.robot.Constants.LauncherConstants.*;

public class PrepareLaunch extends CommandBase {
  //PWMLauncher m_launcher;
  CANLauncher m_launcher;

  /** Creates a new PrepareLaunch. */
  public PrepareLaunch(CANLauncher launcher) {
```

   a. Uncomment the import for CANLauncher and comment or remove the one for PWMLauncher
   b. Uncomment the member variable declaration for the CANLauncher and comment or remove the one for PWMLauncher.
   c. Change the parameter type in the constructor to CANLauncher

3. In Autos:

```
//import frc.robot.subsystems.PWMDrivetrain;
import frc.robot.subsystems.CANDrivetrain;
import edu.wpi.first.wpilibj2.command.CommandBase;
import edu.wpi.first.wpilibj2.command.RunCommand;

public final class Autos {
  /** Example static factory for an autonomous command. */
  public static CommandBase exampleAuto(CANDrivetrain drivetrain) {
```

   a. Uncomment the CANDrivetrain import and comment or remove the PWMDrivetrain import
   b. Change the method parameter type from PWMDrivetrain to CANDrivetrain

3.5 Deploying and testing the KitBot Example

To deploy the example to your robot, you will need to set the Team Number on the project. Click the **WPILib icon** in the top right corner of the VS Code window (W inside a gear) to open the WPILib prompt and start typing “Set Team Number” and select that option when it appears. Enter your team number (no leading 0s – e.g. 123 or 9996) and press Enter.

You are now ready to deploy the KitBot example just like you deployed the test project in Step 4 of the Zero-to-Robot guide.

Warning: Make sure you have space in all directions when operating a robot. Even with known code, the robot may move with unexpected speed or in unexpected directions. Be prepared to Disable (Enter) or E-stop (Spacebar) the robot if necessary. The 2024 KitBot code contains a very simple

autonomous routine that will move the robot backwards at ½ speed for 1 second when the robot is enabled in Autonomous mode.

3.6 Configuring Gamepads

The code is set up to use the Xbox controller class. The Logitech F310 gamepads provided in the Kit of Parts will appear like Xbox controllers to the WPILib software if they are configured in the correct mode. To set up the controllers, check that the switch on the back of the controller is set the 'X' setting. Then when using the controller, make sure the LED next to the Mode button is off, if it is on press the Mode button to toggle it. When the Mode button is on, the controller swaps the function of the left Analog stick and the D-pad.

3.7 What does the code do?

The provided code implements the following robot controls in Teleoperated:

- Driver controller is an Xbox Controller in Slot 0 of the Driver Station
    - Controls the robot drivetrain using Split-stick Arcade Drive
        - Y-axis (vertical) of left stick controls forward-back movement of drivetrain
        - X-axis (horizontal) of right stick controls rotation of drivetrain
- Operator controller is an Xbox Controller in Slot 1 of the Driver Station
    - Left Bumper – Runs both note launcher wheels inward at different speeds while the button is held. This lets the robot intake a Note
    - A button – Runs a short sequence to launch a Note while the button is held
        - Starts front wheel running to get up to speed
        - Waits 1 second
        - Runs back wheel to feed Note into spinning front wheel

4 Overall Code Structure

The provided code utilizes the Command-Based programming structure provided by WPILib. This structure breaks up the robot's actuators into "subsystems" which are controlled by "commands" or collections of commands (aptly name "command groups"). The Command-Based structure may be a bit overkill for a robot of this complexity, but it scales very well for teams looking to add additional functionality to their KitBot, as well as providing a lot of helpful tools for handling timed actions and sequences as will be seen when looking at the code for the Note Launcher. Additionally, this code structure was used by over 60% of teams in 2023, increasing the likelihood that teams around you may be able to provide assistance before or during the event.

To read more about the Command-Based structure, see the Command-Based Programming chapter of the WPILib documentation.

4.1 Ways of creating commands

There are multiple ways that you can define commands within the Command-Based structure. This project uses many of these different types in order to provide exposure to what they would look like in a full robot project. If one type feels odd to you or doesn't make a lot of sense, don't fret, you should be able to use the documentation of what the command does combined with the examples of other ways to create commands to re-create it in the form you prefer. The common ways of creating commands that are utilized in this project are:

- Defined as their own class in their own file (e.g. PrepareLaunch.java and LaunchNote.Java)
- Via a "Command Factory method" in the subsystem (e.g. getIntakeCommand() in *Launcher.java)
- Inline where the command is bound to a button or set as a default (e.g. line 53 of RobotContainer.java)

These same methods are also applicable to Command Groups. This project uses:

- Inline command group where binding occurs. This is done via "decorators", methods you can call on a command to turn it into a command group with specific properties, like "withTimeout" (e.g. line 62 of RobotContainer.java)
- Command Group Factory in its own file (Autos.java)

This project does not create any Command Groups in their own file, an example can be seen at the bottom of the Command Groups page linked above as well as on the Organizing Command-Based Robot Project page.

Most teams will not use all these different styles in their code, instead opting to select one or two types that feel best for them.

5 Code Walkthrough

5.1 Subsystems

As described in the What is Command-Based Programming article, "'subsystems' represent independently-controlled collections of robot hardware (such as motor controllers, sensors, pneumatic actuators, etc.) that operate together".

For the 2024 KitBot we have grouped the 6 motors into 2 groups, the Drivetrain, and the Launcher. For this robot, these choices were pretty easy, the 4 motors in the drivetrain always need to be working together to move the robot around the field and the 2 launcher motors must always be working together to manipulate Notes. Sometimes the boundaries between subsystems may not be so clear, if you have an arm with a shoulder and wrist joint and a set of motorized wheels on the end, is that all one subsystem or multiple? The general rule of thumb to follow is think about what actions, or commands, you might have to control the subsystems. Do you think you might want the two pieces to be controlled independent of each other (i.e. run the intake in or out while moving the arm or wrist?). If you're unsure, err towards more smaller subsystems; you can always make commands that require multiple subsystems but if you end up wanting separate commands to control a single subsystem at the same time, you'll have to refactor the subsystem to split it up.

5.1.1 PWMDrivetrain

This class is the subsystem for the drivetrain if you have wired your motor controller using PWM. If you have wired your motor controller signaling using CAN, see Section 3.4 for information on commenting out the usage of this class and replacing it with the CANDrivetrain class.

5.1.1.1 Package and Imports

```
package frc.robot.subsystems;

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

import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import edu.wpi.first.wpilibj.motorcontrol.MotorControllerGroup;
import edu.wpi.first.wpilibj.motorcontrol.PWMSparkMax;
import edu.wpi.first.wpilibj2.command.SubsystemBase;
```

This section declares what package our subsystem is part of (packages are a way of organizing Java classes) and what other classes we need to reference within this code (imports). A common practice is to add imports as you go; as you find yourself referencing a class you have not yet imported, you can use the lightbulb that VSCode will pop-up for you to add an import for that class. The middle line is a special type of import statement, called a static import, this allows us to reference the constants declared in that class without any class modifier (e.g. kLeftFrontID instead of DrivetrainConstants.kLeftFrontID) allowing the code to be a little more compact.

5.1.1.2 Class declaration, Member Variables and Constructor

```
public class PWMDrivetrain extends SubsystemBase {
  /*Class member variables. These variables represent things the class needs to keep track of and use between
  different method calls. */
  DifferentialDrive m_drivetrain;

  /*Constructor. This method is called when an instance of the class is created. This should generally be used to set up
   * member variables and perform any configuration or set up necessary on hardware.
   */
  public PWMDrivetrain() {
    /*Create MotorControllerGroups for each side of the drivetrain. These are declared here, and not at the class level
     * as we will not need to reference them directly anymore after we put them into a DifferentialDrive.
     */
    MotorControllerGroup leftMotors =
        new MotorControllerGroup(new PWMSparkMax(kLeftFrontID), new PWMSparkMax(kLeftRearID));
    MotorControllerGroup rightMotors =
        new MotorControllerGroup(new PWMSparkMax(kRightFrontID), new PWMSparkMax(kRightRearID));

    // Invert left side motors so both sides drive forward with positive output values
    leftMotors.setInverted(isInverted:true);
    rightMotors.setInverted(isInverted:false);

    // Put our controller groups into a DifferentialDrive object. This object represents all 4 motor
    // controllers in the drivetrain
    m_drivetrain = new DifferentialDrive(leftMotors, rightMotors);
  }
```

The first line of this image is the class declaration. This declares the name of our class and says that it's an extension of the SubsystemBase class. All subsystems should extend this class which provides some utility functions regarding setting the name of the subsystem, registering it with the scheduler, and sending information about it to the dashboard.

The next section is the member variables of the class. These are objects that we need to keep around between calls to the class methods. This typically includes the hardware associated with the subsystem and occasionally some state variables representing the state of the system. For our simple drivetrain, the DifferentialDrive object is all we need to store.

The last section is the constructor. Here we initialize any variables contained in the subsystem.

5.1.1.3 Methods

```
/*Method to control the drivetrain using arcade drive. Arcade drive takes a speed in the X (forward/back) direction
* and a rotation about the Z (turning the robot about it's center) and uses these to control the drivetrain motors */
public void arcadeDrive(double speed, double rotation) {
  m_drivetrain.arcadeDrive(speed, rotation);
}

@Override
public void periodic() {
  /*This method will be called once per scheduler run. It can be used for running tasks we know we want to update each
   * loop such as processing sensor data. Our drivetrain is simple so we don't have anything to put here */
}
```

The remainder of the subsystem class is methods. Here we define any methods that commands may need to call to get status from or perform actions on the subsystem. For our simple drivetrain, the only method we need is the arcadeDrive method which simply passes the parameters through to the same

method on the DifferentialDrive object. The last method in this class, the "periodic" method is a special method that is called each cycle by the Scheduler, regardless of what command is running. You can perform tasks here that you know you want to occur periodically such as updating sensor data. Our simple drivetrain doesn't have any tasks like this, you can choose to remove this method if you'd like.

5.1.2 CANDrivetrain

This class is the subsystem for the drivetrain if you have wired your motor controller using CAN. If you have wired your motor controller signaling using CAN, see Section 3.4 for information on commenting out the usage of PWMDrivetrain and replacing it with the CANDrivetrain class.

5.1.2.1 Package and Imports

```
package frc.robot.subsystems;

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

import com.revrobotics.CANSparkMax;
import com.revrobotics.CANSparkMaxLowLevel.MotorType;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import edu.wpi.first.wpilibj2.command.SubsystemBase;
```

This section declares what package our subsystem is part of (packages are a way of organizing Java classes) and what other classes we need to reference within this code (imports). A common practice is to add imports as you go; as you find yourself referencing a class you have not yet imported, you can use the lightbulb that VSCode will pop-up for you to add an import for that class. The middle line is a special type of import statement, called a static import, this allows us to reference the constants declared in that class without any class modifier (e.g. kLeftFrontID instead of DrivetrainConstants.kLeftFrontID) allowing the code to be a little more compact.

The first two lines of the last section declare the REVRobotics imports. While basic devices are supported directly by WPILib, more complex devices are supported by software provided by the vendor. The REV vendor library has already been added to this example project for you but you can learn more about managing vendor libraries on the WPILib 3rd Party Libraries page if you need to add others or need to create a new project that uses CANSparkMax.

5.1.2.2 Class declaration, Member Variables

```
public class CANDrivetrain extends SubsystemBase {
/*Class member variables. These variables represent things the class needs to keep track of and use between
different method calls. */
DifferentialDrive m_drivetrain;
```

The first line of this image is the class declaration. This declares the name of our class and says that it's an extension of the SubsystemBase class. All subsystems should extend this class which provides some utility functions regarding setting the name of the subsystem, registering it with the scheduler, and sending information about it to the dashboard.

The next section is the member variables of the class. These are objects that we need to keep around between calls to the class methods. This typically includes the hardware associated with the subsystem

and occasionally some state variables representing the state of the system. For our simple drivetrain, the DifferentialDrive object is all we need to store.

5.1.2.3 Constructor

```
public CANDrivetrain() {
  CANSparkMax leftFront = new CANSparkMax(kLeftFrontID, MotorType.kBrushed);
  CANSparkMax leftRear = new CANSparkMax(kLeftRearID, MotorType.kBrushed);
  CANSparkMax rightFront = new CANSparkMax(kRightFrontID, MotorType.kBrushed);
  CANSparkMax rightRear = new CANSparkMax(kRightRearID, MotorType.kBrushed);

  /*Sets current limits for the drivetrain motors. This helps reduce the likelihood of wheel spin, reduces motor heating
   *at stall (Drivetrain pushing against something) and helps maintain battery voltage under heavy demand */
  leftFront.setSmartCurrentLimit(kCurrentLimit);
  leftRear.setSmartCurrentLimit(kCurrentLimit);
  rightFront.setSmartCurrentLimit(kCurrentLimit);
  rightRear.setSmartCurrentLimit(kCurrentLimit);

  // Set the rear motors to follow the front motors.
  leftRear.follow(leftFront);
  rightRear.follow(rightFront);

  // Invert the left side so both side drive forward with positive motor outputs
  leftFront.setInverted(isInverted:true);
  rightFront.setInverted(isInverted:false);

  // Put the front motors into the differential drive object. This will control all 4 motors with
  // the rears set to follow the fronts
  m_drivetrain = new DifferentialDrive(leftFront, rightFront);
}
```

This section is the constructor, where we initialize and configure the hardware for the subsystem. The first section declares the motor controllers and indicates that they are connected to brushed motors (the CIM motors are brushed motors, you would change this to Brushless if using a NEO or NEO 500 motor that connects to all 3 wires of the Spark Max). These are used as local variables as we don't need to reference them directly after we use them in the constructor. If you had sensors attached to them or wanted to get other information out like current or temperature, you might choose to shift these to be class member variables.

The next section sets current limits on each of the motors. After that, the rear motors are set to follow the front motors on their respective side. This tells these controllers to listen for traffic to or from the front controllers and use that information to match the direction and output. This creates slightly less CAN traffic than putting the controllers on each side into a group like is done in the PWMDrivetrain (though that works fine as well!).

Then the left side is inverted. This is so that a positive command will result in the wheels moving that side of the robot forward.

The last line sets up the DifferentialDrive with the front controllers. The rear controllers will follow along so the DifferentialDrive object doesn't need to know about them.

5.1.2.4 Methods

```
/*Method to control the drivetrain using arcade drive. Arcade drive takes a speed in the X (forward/back) direction
* and a rotation about the Z (turning the robot about it's center) and uses these to control the drivetrain motors */
public void arcadeDrive(double speed, double rotation) {
  m_drivetrain.arcadeDrive(speed, rotation);
}

@Override
public void periodic() {
  /*This method will be called once per scheduler run. It can be used for running tasks we know we want to update each
  * loop such as processing sensor data. Our drivetrain is simple so we don't have anything to put here */
}
```

The remainder of the subsystem class is methods. Here we define any methods that commands may need to call to get status from or perform actions on the subsystem. For our simple drivetrain, the only method we need is the arcadeDrive method which simply passes the parameters through to the same method on the DifferentialDrive object. The last method in this class, the "periodic" method is a special method that is called each cycle by the Scheduler, regardless of what command is running. You can perform tasks here that you know you want to occur periodically such as updating sensor data. Our simple drivetrain doesn't have any tasks like this, you can choose to remove this method if you'd like.

5.1.3 PWMLauncher

This class is the subsystem for the launcher if you have wired your motor controller using PWM. If you have wired your motor controller signaling using CAN, see Section 3.4 for information on commenting out the usage of this class and replacing it with the CANLauncher class.

5.1.3.1 Package, Imports, Class Declaration, Member Variables, and Constructor

The first sections of this subsystem are very similar to the PWMDrivetrain subsystem. See section 5.1.1 for more detailed description of each of these parts of the code.

5.1.3.2 Methods – Command Factory

```
public CommandBase getIntakeCommand() {
    return this.startEnd(
        () -> {
          setFeedWheel(kIntakeFeederSpeed);
          setLaunchWheel(kIntakeLauncherSpeed);
        },
        () -> {
          stop();
        });
}
```

This method is what is called a "Command factory" because it creates instances of a command. We can call this method from wherever we are setting up buttons, creating command groups, etc. to get an instance of this command. The command itself is created using one of the inline command helpers (in

this case startEnd). To see the different options available, check the JavaDoc for the Subsystem class and look for the methods that return a CommandBase object.

In this case we use startEnd() which calls the first parameter when the command starts and the second parameter when the command is interrupted (e.g by a new command being scheduled or a button being released). When the command starts we want to start spinning both wheels inward at a specific speed to pull in a Note and when it ends we want to stop the wheels.

The programming technique used here is called a "lambda expression". To learn more about lambda expressions, check out this section of the WPILib docs about Lambda Expressions in Java.

5.1.3.3 Methods – Hardware Control

```
public void setLaunchWheel(double speed)
{
  m_launchWheel.set(speed);
}

public void setFeedWheel(double speed)
{
  m_feedWheel.set(speed);
}

public void stop()
{
  m_launchWheel.set(speed: 0);
  m_feedWheel.set(speed: 0);
}
```

The remainder of the subsystem class is hardware access methods. Here we define any methods that commands may need to call to get status from or perform actions on the subsystem. For our launcher this includes methods to set the speed of each wheel and a method to stop both wheels at once. The stop() method is a design choice, you absolutely can skip this method and just call the set speed methods of each of the two wheels anywhere you would want to stop them.

5.1.4 CANLauncher

This class is the subsystem for the launcher if you have wired your motor controller using CAN. If you have wired your motor controller signaling using CAN, see Section 3.4 for information on commenting out the usage of the PWMLauncher class and replacing it with the CANLauncher class.

5.1.4.1 Package, Imports, Class Declaration, Member Variables, and Constructor

The first sections of this subsystem are very similar to the CANDrivetrain subsystem. See section 5.1.1 for more detailed description of each of these parts of the code.

5.1.4.2 Methods – Command Factory

```
public CommandBase getIntakeCommand() {
    return this.startEnd(
        () -> {
          setFeedWheel(kIntakeFeederSpeed);
          setLaunchWheel(kIntakeLauncherSpeed);
        },
        () -> {
          stop();
        });
}
```

This method is what is called a "Command factory" because it creates instances of a command. We can call this method from wherever we are setting up buttons, creating command groups, etc. to get an instance of this command. The command itself is created using one of the inline command helpers (in this case startEnd). To see the different options available, check the JavaDoc for the Subsystem class and look for the methods that return a CommandBase object.

In this case we use startEnd() which calls the first parameter when the command starts and the second parameter when the command is interrupted (e.g by a new command being scheduled or a button being released). When the command starts we want to start spinning both wheels inward at a specific speed to pull in a Note and when it ends we want to stop the wheels.

The programming technique used here is called a "lambda expression". To learn more about lambda expressions, check out this section of the WPILib docs about Lambda Expressions in Java.

5.1.4.3 Methods – Hardware Control

```
public void setLaunchWheel(double speed)
{
  m_launchWheel.set(speed);
}

public void setFeedWheel(double speed)
{
  m_feedWheel.set(speed);
}

public void stop()
{
  m_launchWheel.set(speed: 0);
  m_feedWheel.set(speed: 0);
}
```

The remainder of the subsystem class is hardware access methods. Here we define any methods that commands may need to call to get status from or perform actions on the subsystem. For our launcher this includes methods to set the speed of each wheel and a method to stop both wheels at once. The stop() method is a design choice, you absolutely can skip this method and just call the set speed methods of each of the two wheels anywhere you would want to stop them.

5.2 Commands

This section will cover the 3 command files in the commands folder. Additional inline commands are defined in the RobotContainer file, they will be covered in the next section.

5.2.1 Autos

The Autos file is an example of a "Static Command Factory". Your program should never create an Autos object (as shown by the constructor simply printing an error message) instead you call class methods statically using Autos.exampleAuto() type syntax. This structure is one of the ways to define complex groups that involve multiple subsystems (though our example here is not complex and requires only a single subsystem).

```
public final class Autos {
  /** Example static factory for an autonomous command. */
  public static CommandBase exampleAuto(PWMDrivetrain drivetrain) {
    return new RunCommand(()-> drivetrain.arcadeDrive(-.5, rotation: 0)).withTimeout(seconds: 1)
    .andThen(new RunCommand(()-> drivetrain.arcadeDrive(speed: 0,rotation: 0)));
  }

  private Autos() {
    throw new UnsupportedOperationException("This is a utility class!");
  }
```

Our example file only has a single autonomous routine to get. You could easily extend this pattern by adding additional methods to define more autonomous routines and you could select between them using a SendableChooser on the dashboard.

This simple autonomous routine instructs the robot to drive backwards for 1 second at 50% power by using the withTimeout decorator to set a timeout of 1 second on the driving command. It uses the andThen decorator to tell the robot to stop moving after the first command is complete. The different types of command compositions that are built-in via decorators and factory methods are described on the Command Compositions page.

5.2.2 LaunchNote

The LaunchNote command is the first of two examples in this project of a command as a standalone class defined in its own file. This structure is generally quite useful for complex commands but can be used for simple commands as well, as shown here, depending on preference. The LaunchNote command runs both wheels of the Launcher to eject the Note from the robot.

5.2.2.1 Class Definition, Members, and Constructor

```
public class LaunchNote extends Command {
  PWMLauncher m_launcher;

  //CANLauncher m_launcher;

  /** Creates a new LaunchNote. */
  public LaunchNote(PWMLauncher launcher) {
    // save the launcher system internally
    m_launcher = launcher;

    // indicate that this command requires the launcher system
    addRequirements(m_launcher);
  }
```

The first line in this section defines the class as an extension of Command. This base class defines many of the helper methods used to manage getting and setting requirements, interacting with the Scheduler, and handling decorators that modify the command or collect it into a group.

Next, we define our class member variables, in this case the subsystem that the command operates on. We need to save this subsystem internally in order to call methods on it when the command is running.

Finally, we have the constructor. This constructor takes the subsystem as a parameter so it can be saved locally. The last line of the constructor indicates that this command requires this subsystem. This requirements declaration is the glue that holds the Command-Based architecture together. Any command that performs any output actions on a subsystem must “require” that subsystem to help the Scheduler maintain only a single command controlling a subsystem at once.

```
// Called when the command is initially scheduled.
@Override
public void initialize() {
  m_launcher.setLaunchWheel(kLauncherSpeed);
  m_launcher.setFeedWheel(kLaunchFeederSpeed);
}
```

Next are the “lifecycle” methods of the command. These are the methods that are called when the command is scheduled (initialize), while it is running (execute, isFinished) and when it is done or interrupted (end). These methods are stubbed out in the base command class and are generally overridden by child classes (this is indicated by using the @Override annotation above the method). For the LaunchNote class, we set the wheels to a speed when the command is initialized. We can do this because the desired speed doesn’t change while the command is running. If you wanted to change the speed while the command was running (based on a joystick input for example), setting these speeds in execute would be more appropriate.

```
// Called every time the scheduler runs while the command is scheduled.
@Override
public void execute() {

}
```

Because our LaunchNote command uses a single speed set during initialize(), we don't have anything to do here in the execute method. This method is called each scheduler run (generally every 20ms) while the command is running.

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

The isFinished() method is also called each run of the scheduler, after execute() to check if the command is finished running. In this case, we want the wheels to keep running as long as the operator is holding the button so we always return False to indicate the command is not finished. The Scheduler will handle canceling the command when the operator releases the button (covered more in the next section on RobotContainer).

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

The end() command is called when the command is being removed from the scheduler. You should perform and "clean-up" needed on your mechanism here. Often, but not always, you will want to stop the mechanism in this method, as is the case here. The scheduler passes in a boolean to indicate whether the command was interrupted or not (not interrupted = ended on it's own by returning True from isFinished()). In this case, we don't care whether the command was interrupted or not, we want to stop the wheels when the command ends.

5.2.3 PrepareLaunch

The PrepareLaunch command is the second of two examples in this project of a command as a standalone class defined in its own file. The PrepareLaunch command spins just the outside wheel of the launcher to allow it to get up to speed before launching. The PrepareLaunch command code is almost identical to LaunchNote with two main exceptions.

```
// Called when the command is initially scheduled.
@Override
public void initialize() {
  m_launcher.setLaunchWheel(kLauncherSpeed);
}
```

In the initialize() method, the speed of the launch wheel (outside wheel) is set but the feed wheel is left alone so the Note is not yet fed into the launch wheel.

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

In the end() method we do not stop the wheels because we need the launch wheel to keep running as part of the launching sequence. This does mean that we have to handle stopping the wheel if the sequence is interrupted while running this command which we will cover in the RobotContainer section 5.5 below.

5.3 Constants

This class contains named constants used elsewhere in the code. Subclasses are used to organize the constants into distinct groups, in this case by subsystem. The file contains comments to indicate what the constants represent.

5.4 Main and Robot

These two files are identical to the default Command-Based template. You can find a description of the elements in the Robot class in the Structuring a Command-Based Robot Project article. The Main class is generally not modified for FRC robot programming, regardless of template.

5.5 RobotContainer

The RobotContainer class is where instances of the robot subsystems and controllers are declared and where default commands and mappings of buttons to commands are defined.

```
public class RobotContainer {
  // The robot's subsystems are defined here.
  private final PWMDrivetrain m_drivetrain = new PWMDrivetrain();
  //private final CANDrivetrain m_drivetrain = new CANDrivetrain();
  private final PWMLauncher m_launcher = new PWMLauncher();
  //private final CANLauncher m_launcher = new CANLauncher();

  /*The gamepad provided in the KOP shows up like an XBox controller if the mode switch is set to X mode using the
  * switch on the top.*/
  private final CommandXboxController m_driverController =
      new CommandXboxController(OperatorConstants.kDriverControllerPort);
  private final CommandXboxController m_operatorController =
      new CommandXboxController(OperatorConstants.kOperatorControllerPort);
```

The first section defines the class member variables. For RobotContainer this generally includes all of your subsystems and control devices. This code uses the CommandXboxController to represent the gamepads as it contains a number of helper methods that make it much easier to connect commands to buttons.

```
/** The container for the robot. Contains subsystems, OI devices, and commands. */
public RobotContainer() {
  // Configure the trigger bindings
  configureBindings();
}
```

The constructor contains a single call to configureBindings() which we will cover below. This method is used to set up button bindings and default commands. This constructor rarely needs to be modified.

```
private void configureBindings() {
  // Set the default command for the drivetrain to drive using the joysticks
  m_drivetrain.setDefaultCommand(
      new RunCommand(
          () ->
              m_drivetrain.arcadeDrive(
                  -m_driverController.getLeftY(), -m_driverController.getRightX()),
          m_drivetrain));
```

The configureBindings() method is where we put all of the glue code that tells commands when to run. The first section here is for the drivetrain. We want a command to run on our drivetrain to allow us to drive the robot with joysticks whenever we don't have some other command using the drivetrain (like the exampleAuto command). To do this we use the setDefaultCommand() method of the subsystem. This sets the command that will run whenever the Scheduler sees nothing else running on that subsystem.

To set up the command, we use an inline command definition using the RunCommand class. The RunCommand class is used to turn a single method call into a command. The method we pass to the RunCommand is inserted into the execute() section of the command lifecycle discussed in the LaunchNote section, meaning it will be called repeatedly while the command is scheduled. The method to run is again captured using a Lambda expression.

In this code, the method we want to call is the arcadeDrive method of the drivetrain subsystem. For the forward/back movement we pass in the value from the Y-axis (vertical) of the left stick of the controller, but we negate it. This is because joysticks generally define pushing the stick away from you as negative and pulling the stick towards you as positive (a result of the original use being flight simulators). We want pushing the stick away from us to drive the robot forward, so we negate the value. Similar for the turning value where we negate the X-axis (horizontal) of the right stick of the controller. The joystick considers pushing this to the right as a positive value, but the WPILib classes consider clockwise rotation (what would be expected when pushing the joystick right) as negative.

```
/*Create an inline sequence to run when the operator presses and holds the A (green) button. Run the PrepareLaunch
* command for .25 seconds and the run the LaunchNote command */
m_operatorController.a().whileTrue(
  new PrepareLaunch(m_launcher)
  .withTimeout(seconds: .25)
  .andThen(new LaunchNote(m_launcher))
  .handleInterrupt(() -> m_launcher.stop())
);
```

Next, we set up an inline command group to run a launch sequence while the operator holds the A button on the controller. We first use the a() helper method on the controller to get a Trigger and then use the whileTrue() method to run the command while the button is held, canceling it when the button is released. You can see other options available for Trigger methods on the Trigger Javadoc page or on the WPILib doc page about Binding Commands to Triggers.

The first command in our sequence is PrepareLaunch(). We then use the withTimeout() decorator to have this command run for a fixed time before ending. The andThen() decorator is used to run the LaunchNote command after the timeout expires. Finally, the handleInterrupt() decorator is used to provide a method to run if the command is interrupted; this is needed to make sure the wheels stop if the operator lets go of the button while the PrepareLaunch command is running as the end() method of that command won't stop the wheels by itself.

6 Making Changes

This section details some common possible changes you may want to make to the KitBot code and provides some references for how to approach making those changes.

6.1 Changing buttons for actions

One of the easiest changes to make to Command-Based robot code is to switch what buttons or button behaviors control a command. The commands used in the 2024 KitBot do not end (isFinished always returns false) so they should generally only be used with the whileHeld() behavior, but changing which buttons they map to can be done very simply.

The button mappings in the example code are done near the end of the configureBindings() method inside the RobotContainer file. The bindings used for this project are made using the helper methods of the CommandXboxController class. These helper methods exist for each button on the controller and return a Trigger object which can then be used to specific a behavior for the binding.

As provided the code connects the **a** button to the launch sequence and the **leftBumper** to intaking a Note. To change these, simply change the a() or leftBumper() helper methods, to the method for any of the other buttons! You can see all of the available options by looking through the CommandXboxController Javadoc for methods which take no parameter and return a Trigger object.

For example, to change the intake command from the left bumper to the x button, simply replace the leftBumper() with x()

```
//Before
m_operatorController.leftBumper().whileTrue(m_launcher.getIntakeCommand());
//After
m_operatorController.x().whileTrue(m_launcher.getIntakeCommand());
```

6.2 Changing Drive Axis Behavior

Another easy change to make is to modify which axes of the controller are used as which part of the robot driving and how. The provided code does this mapping when setting up the drivetrain default command at the top of configureBindings() in RobotContainer.

The example code uses the Y-axis of the left stick to drive forward and back and the X-axis of the right stick to rotate. These can easily be swapped to the opposite sticks or move just one so they are on the same stick! To review the available options, look for methods that return a **double** in the CommandXboxController Javadoc. To make this type of modification, locate the method call you wish to change, such as getLeftY(), and replace it with the new desired method, such as getRightY()

Example: changing the forward-back driving to the right stick Y-axis and leaving the rotation on the right X-axis

```
private void configureBindings() {
  //Set the default command for the drivetrain to drive using the joysticks
  m_drivetrain.setDefaultCommand(new RunCommand(() -> m_drivetrain.arcadeDrive(
    -m_driverController.getRightY(), -m_driverController.getRightX()),
    m_drivetrain
  ));
```

You can also modify the axis values. One common modification is to cube the values. This preserves the sign of the value (positive stays postive, negative stays negative) and the maximum value (doesn't reduce the max speed of the robot) while providing less sensitivity at low inputs, potentially allowing for more precise control at low speeds. To make this type of modification, you can apply the modification to the axis where it's being captured.The Arcade Drive method from the Differential Drive class already squares the inputs by default (while preserving sign), you likely want to disable this if you are cubing them yourself by passing an additional parameter to the Arcade Drive method call in the drivetrain subsystem.

Example: changing only the rotation axis to be cubed:

```
private void configureBindings() {
  //Set the default command for the drivetrain to drive using the joysticks
  m_drivetrain.setDefaultCommand(new RunCommand(() -> m_drivetrain.arcadeDrive(
    -m_driverController.getLeftY(), -Math.pow(m_driverController.getRightX(), 3)),
    m_drivetrain
  ));
```

```
48  /*Method to control the drivetrain using arcade drive. Arcade drive takes a speed in the X (forward/back) direction
49   * and a rotation about the Z (turning the robot about it's center) and uses these to control the drivetrain motors */
50  public void arcadeDrive(double speed, double rotation) {
51    m_drivetrain.arcadeDrive(speed, rotation, squareInputs:false);
52  }
```

Another common modification is to scale the values down by default, but allow for the maximum value if a button is pressed (turbo mode). This type of modification can also be done at the point of capture, though as complexity grows, you may wish to shift from an inline command definition to a different type where you can define the command behavior more clearly.

Example: Scale the forward-back driving by 50% unless the right bumper is pressed

```
private void configureBindings() {
  //Set the default command for the drivetrain to drive using the joysticks
  m_drivetrain.setDefaultCommand(new RunCommand(() -> m_drivetrain.arcadeDrive(
    -m_driverController.getLeftY()*(m_driverController.getHID().getRightBumper()?1:.5), -Math.pow(m_driverController.getRightX(), 3)),
    m_drivetrain
  ));
```

This example uses two specific things that warrant explanation. The first one is that it uses the getHID() method on the driver controller. This returns the XboxController object that CommandXboxController is wrapping. This gives us access to some different methods than the CommandXboxController, in this case getRightBumper() which allows us to get the Boolean value of the button rather than a Trigger object associated with the button.

The other thing this example uses is the ? operator, called the ternary operator. This operator allows us to write a simple "if" statement in a very compact way, if the right bumper is pressed, we multiply by 1, if not, by 0.5.

6.3 Changing Drive Type

The last likely change we will cover is changing from Arcade Drive to Tank Drive. Unlike Arcade drive which maps one axis to rotation and one to forward/back, Tank drive maps one axis (generally the Y-axis) to each side of a differential drivetrain. To make this change, you'll have to reach beyond RobotContainer as the provided drivetrain subsystems don't expose a tank drive method. In the appropriate drivetrain subsystem (PWMDrivetrain or CANDrivetrain) make a new method called tankDrive(). This method should look a lot like the arcadeDrive method. Then, modify the default command mapping in RobotContainer to use this new method with the appropriate joystick axis.

Example:

```
public void tankDrive(double leftSpeed, double rightSpeed)
{
  m_drivetrain.tankDrive(leftSpeed, rightSpeed);
}
```

```
private void configureBindings() {
  //Set the default command for the drivetrain to drive using the joysticks
  m_drivetrain.setDefaultCommand(new RunCommand(() -> m_drivetrain.tankDrive(
    -m_driverController.getLeftY(), -m_driverController.getRightY()),
    m_drivetrain
  ));
```

6.4 Developing Autonomous Routines

The provided code contains a very basic autonomous mode that drives backwards at ½ power for 1 second. Additional autonomous modes can be developed, either by adding additional methods in the Autos file (see the Hatchbot Inlined example project for an example of this style with more complex autonomous) or by creating separate files for each autonomous routine (see the Hatchbot Traditional for an example of this approach).

It's common (but definitely not required!) to have multiple autonomous routines that you may wish to run based on different starting locations or strategies. If you pursue this, the most common way to choose between them for each match is to select between them using a SendableChooser on the dashboard.