FIRST ROBOTICS COMPETITION

REBUILT™ PRESENTED BY HAAS

FIRST® AGE™
presented by Qualcomm

firstinspires.org/robotics/frc

2026 *FIRST*® Robotics Competition

KitBot Java Software Guide

1 Contents

2 Document Overview

This document will take you through how to get your 2026 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.

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. The KitBot wiring and code is documented with the Control System components that have recently come in the Rookie Kit of Parts (i.e. REV PDH and Spark MAX controllers), the KitBot wiring and code can be adapted to other electronics, but that adaptation is not covered by these documents.

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.

3.3 Opening the 2026 KitBot Example

The 2026 KitBot example code is provided in a single zip file. The Java code contains two complete projects which illustrate different ways of creating Commands. Some description of the difference can be found in Ways of creating commands below. 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 open the desired one of the two example projects, and then click **Select Folder.**

3.4 Spark MAX firmware update and CAN IDs

Before using the SPARK MAXs with CAN control, they each need to be assigned a unique 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 page, make sure the robot is in a safe state to do so (wheels not touching the ground or table, all hands clear of the robot).
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.

3.5 Installing REVLib

The software library for the SPARK MAX in CAN mode is provided by the vendor (REV Robotics). 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. If the version does not start with 2026, the 2026 version may not be available yet, you will have to wait for it to be available to be able to build and test the KitBot code.
    b. Unzip into the *C:\Users\Public\wpilib\2026* directory on Windows and *~/wpilib/2026* directory on Unix-like systems.
    c. Click the WPILib icon in the top right (looks like red and grey '<<>>' symbol), and start typing "Manage Vendor Libraries" and select that option when it appears, then select "Install new libraries (offline)", finally select the installed version of REVLib.
2. Install Online - While the computer is connected to the Internet, you can use the WPILib Vendor Library manager inside VS Code to locate REVLib. If it does not appear, the 2026 version of REVLib may be available yet, you will have to wait for it to be available to be able to build and test the KitBot code.

3.6 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 (Logo that looks like red and grey '<<>>'

symbol) 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 2026 KitBot code contains a simple autonomous routine that will move the robot forwards at ½ speed for ¼ second (and then launch balls for 10 seconds) when the robot is enabled in Autonomous mode.

3.7 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 light is on, the controller swaps the function of the left Analog stick and the D-pad.

3.8 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
        - The “front” of the robot from this code’s perspective is the intake side.
- Operator controller is an Xbox controller in Slot 1 of the Driver Station
    - Controls the gamepiece rollers using the bumpers and buttons
        - Left Bumper – Intakes gamepieces
        - Right Bumper – Launches gamepieces by first running a spin-up sequence for 1 second and then launching for as long as the button is held.
        - A-Button – Ejects gamepieces back out the intake.

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. Additionally, this code structure was used by over 60% of teams in 2024, 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 two of these different types to provide exposure to what they would look like in a full robot project. The common ways of creating commands that are utilized in this project are:

- Classes: Command/group is defined as its own class in its own file.
- Inline: Command/group is defined via a "Command Factory method" in the subsystem or a separate static command class or is composed "inline" using lambda functions.

Traditional	Inline
+ Generally easier to understand	- Slightly high learning curve
- Modularity – Commands can be long depending on the complexity	+ Modularity – Commands are broken down into small pieces and strung together into groups
- Boilerplate – Command classes require subclassing and can be long	+ Boilerplate – Commands are written as methods in the subsystem, meaning less unnecessary code
- Organization – Having many Command classes can cause clutter and slow programmer's efficiency	+ Organization – Commands are grouped together based on the subsystems they require, leading to fewer/no dedicated Command classes
+ Debugging – Easier to debug due to more defined structure and traditional logic	- Debugging – More difficult to debug due to the lambda functions and command compositions

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 2026 KitBot we have 6 motors that make up 2 subsystems, the Drivetrain, and the Fuel system. The 4 motors in the drivetrain always need to be working together to move the robot around the field and the Fuel motors must spin in coordinated ways to manipulate Fuel.

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 CANDriveSubsystem

This class is the subsystem for the drivetrain.

5.1.1.1 Imports

```
import com.revrobotics.spark.SparkBase.PersistMode;
import com.revrobotics.spark.SparkBase.ResetMode;
import com.revrobotics.spark.SparkLowLevel.MotorType;
import com.revrobotics.spark.SparkMax;
import com.revrobotics.spark.config.SparkMaxConfig;

import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import edu.wpi.first.wpilibj2.command.SubsystemBase;
import static frc.robot.Constants.DriveConstants.*;
```

This section declares what other classes or packages 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 add an import for that class. The first group of imports in this subsystem is from the REV Robotics library for various items we need to reference for the Spark MAXs. The second group is WPILib classes (one for the type of drivetrain on the robot and one for subsystems) and the DriveConstants section of the Constants file from this project. The constants import is a special import called a static import that allows us to reference the constants using just their name, reducing clutter

by not having to use “DriveConstants.” for each of them. Because the names of all of the variables in the DriveConstants section are very unique, we can do this safely without likelihood of any confusion.

If you’re using the InlineCommands version of the project you’ll see a few more imports for some WPILib Command classes as well as one at the top for a Java class called “DoubleSupplier” which is how you’ll pass changing values into the commands.

5.1.1.2 Class declaration, and Member Variables

```
public class CANDriveSubsystem extends SubsystemBase {
  private final SparkMax leftLeader;
  private final SparkMax leftFollower;
  private final SparkMax rightLeader;
  private final SparkMax rightFollower;

  private final DifferentialDrive drive;
```

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.

5.1.1.3 Constructor

```
/** Class to drive the robot over CAN */
public CANDriveSubsystem() {
  // create brushed motors for drive
  leftLeader = new SparkMax(DriveConstants.LEFT_LEADER_ID, MotorType.kBrushed);
  leftFollower = new SparkMax(DriveConstants.LEFT_FOLLOWER_ID, MotorType.kBrushed);
  rightLeader = new SparkMax(DriveConstants.RIGHT_LEADER_ID, MotorType.kBrushed);
  rightFollower = new SparkMax(DriveConstants.RIGHT_FOLLOWER_ID, MotorType.kBrushed);

  // set up differential drive class
  drive = new DifferentialDrive(leftLeader, rightLeader);
```

The next section is the constructor. In the first part of the constructor we initialize any variables contained in the subsystem. In the case of our drivetrain, we initialize the 4 motor controllers and add one controller for each side into a WPILib DifferentialDrive object which describes the whole drivetrain.

The remainder of the constructor (not pictured) sets up the SparkMAXs for the drivetrain. One motor on each side is set as a follower and directed to follow the leader, then the leader motors are configured with some additional details. Review the comments for what settings are being configured and why.

5.1.1.4 Methods

```
38    public void driveArcade(double xSpeed, double zRotation) {
39      drive.arcadeDrive(xSpeed, zRotation);
40    }
```

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.

In the InlineCommands version of the project this method will look a little different, for more detail see the Inline Commands section.

5.1.2 CANFuelSubsystem

This class is the subsystem for the roller mechanism.

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

The first sections of this subsystem are very similar to the Drive subsystem. See that section for more detailed description of each of these parts of the code. For the launcher, the two single motors are created and controlled independently, no follower or drivetrain object is used.

5.1.2.2 SmartDashboard

```
SmartDashboard.putNumber(key:"Intaking feeder roller value", INTAKING_FEEDER_VOLTAGE);
SmartDashboard.putNumber(key:"Intaking intake roller value", INTAKING_INTAKE_VOLTAGE);
SmartDashboard.putNumber(key:"Launching feeder roller value", LAUNCHING_FEEDER_VOLTAGE);
SmartDashboard.putNumber(key:"Launching launcher roller value", LAUNCHING_LAUNCHER_VOLTAGE);
SmartDashboard.putNumber(key:"Spin-up feeder roller value", SPIN_UP_FEEDER_VOLTAGE);
```

The constructor for this subsystem has an extra section which puts a number of the fuel system constants onto the dashboard. This allows for tuning these values easily while testing with the robot. The “SmartDashboard” class is an easy way to access the “NetworkTables” protocol which is like a shared dictionary between the robot program, dashboard, and any other clients (like vision coprocessors”. Values are referenced using “keys” and we can use these same keys later to get the latest copy of each value. For more information about tuning the robot using these values, see Tuning Robot Values.

5.1.2.3 Methods – Hardware Control

```
public void setIntakeLauncherRoller(double voltage) {
  intakeLauncherRoller.setVoltage(voltage);
}
```

```
public void setFeederRoller(double voltage) {
  feederRoller.setVoltage(voltage);
}
```

```
public void stop() {
  feederRoller.set(0);
  intakeLauncherRoller.set(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 fuel system this includes methods to set the speed of each motor and a method to stop both motors.

In the InlineCommands version of the project this method will look a little different, for more detail see the Inline Commands section.

5.2 Commands

Commands are what tell the robot when to run the different components that are defined in their subsystems. Commands are "scheduled" for execution, performed, and then removed from the command scheduler. Each command has 4 distinct parts that are performed throughout its lifecycle:

- Initialize: performed when the command is initially scheduled. Anything put in the initialize section of a command will run right before the main body of the command.
- Execute: the main body of the command, which runs once every loop cycle (~20 ms)
- End: performed when the command is removed from the scheduler which will occur when the command indicates it's finished or if it's interrupted by a new command requiring one of the same subsystems.
- isFinished: called once per loop cycle after the execute method. isFinished returns true when the condition for exiting the command is met. When isFinished returns true, the end method is called.

As we discussed in section 4, the two ways of writing commands that we focus on in this tutorial are Class commands which subclass the WPILib Command class, and inline commands, which use command factories and command decorators to create commands. Regardless of the way your team chooses to make commands, this is the underlying structure that commands follow.

5.3 Class Commands

This subsection will focus on the class command-based framework. Class commands are defined in their own classes by subclassing the generic WPILib Command. This means that we can directly override the behavior of the command methods.

5.3.1 DriveCommand

5.3.1.1 Imports and Constructors

```java
package frc.robot.commands;

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

import edu.wpi.first.wpilibj2.command.Command;
import edu.wpi.first.wpilibj2.command.button.CommandXboxController;
import frc.robot.subsystems.CANDriveSubsystem;

/* You should consider using the more terse Command factories API instead https://docs.w
public class Drive extends Command {
  /** Creates a new Drive. */
  CANDriveSubsystem driveSubsystem;
  CommandXboxController controller;

  public Drive(CANDriveSubsystem driveSystem, CommandXboxController driverController) {
    // Use addRequirements() here to declare subsystem dependencies.
    addRequirements(driveSystem);
    driveSubsystem = driveSystem;
    controller = driverController;
  }
```

The constructor of the command simply stores some parameters into class variables for later use and declares the subsystem the command requires using “addRequirements”. All commands should have an “addRequirements” indicating any subsystems they will call methods on that control outputs (i.e. you may call methods to **get** values from a subsystem without requiring it but should not **set** values)

5.3.1.2 Methods – Command State Overrides

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

// Called every time the scheduler runs while the command is scheduled.
// The Y axis of the controller is inverted so that pushing the
// stick away from you (a negative value) drives the robot forwards (a positive
// value). The X axis is scaled down so the rotation is more easily
// controllable.
@Override
public void execute() {
  driveSubsystem.driveArcade(-controller.getLeftY() * DRIVE_SCALING, -controller.getRightX() * ROTATION_SCALING);
}

// Called once the command ends or is interrupted.
@Override
public void end(boolean interrupted) {
  driveSubsystem.driveArcade(xSpeed:0, zRotation:0);
}

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

In the execute method, we call the ArcadeDrive method from the drive subsystem which tells the motors to drive such that the robot moves according to the joystick inputs. If you don't have any code in a particular command method (such as "initialize" and "end" here) it is permitted to delete them, the base Command class includes a blank implementation that will be used if your class doesn't have an Override.

5.3.2 Eject/Intake/Launch/Spinup

These commands are nearly identical to each other, but set the rollers to different values. This is an example of how the class approach can be more verbose with more boilerplate. Though you avoid this by choosing to make a single command class for setting these values and pass the values to be set in as parameters. The first section of these commands are very similar to the Drive command. See section **Error! Reference source not found.**1 for more detailed description of each of the parts of the code.

5.3.2.1 Imports and Constructor

```java
package frc.robot.commands;

import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;
import edu.wpi.first.wpilibj2.command.Command;
import frc.robot.subsystems.CANFuelSubsystem;
import static frc.robot.Constants.FuelConstants.*;

/* You should consider using the more terse Command factori
public class Eject extends Command {
  /** Creates a new Intake. */

  CANFuelSubsystem fuelSubsystem;

  public Eject(CANFuelSubsystem fuelSystem) {
    addRequirements(fuelSystem);
    this.fuelSubsystem = fuelSystem;
  }
```

5.3.2.2 Methods – Command State Overrides

```java
// Called when the command is initially scheduled. Set the rollers to the
// appropriate values for ejecting
@Override
public void initialize() {
  fuelSubsystem
      .setIntakeLauncherRoller(
          -1 * SmartDashboard.getNumber(key:"Intaking intake roller value", INTAKING_INTAKE_VOLTAGE));
  fuelSubsystem
      .setFeederRoller(-1 * SmartDashboard.getNumber(key:"Intaking feeder roller value", INTAKING_FEEDER_VOLTAGE));
}

// Called every time the scheduler runs while the command is scheduled. This
// command doesn't require updating any values while running
@Override
public void execute() {
}

// Called once the command ends or is interrupted. Stop the rollers
@Override
public void end(boolean interrupted) {
  fuelSubsystem.setIntakeLauncherRoller(voltage:0);
  fuelSubsystem.setFeederRoller(voltage:0);
}

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

In these commands, the roller values are set in the initialize() method. The values are retrieved from the SmartDashboard keys (the constants specified are the "default" values if the key couldn't be found) to allow for easy tuning. The execute() method is empty because we don't need to update the roller values after we've set them. The end() method sets the rollers to 0 except for the Spinup command where we want the rollers to keep running so we can transition directly to launching.

5.3.3 ExampleAuto and Launch Sequence

```java
package frc.robot.commands;

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

// NOTE:  Consider using this command inline, rather than writing a subclass.  For more
// information, see:
// https://docs.wpilib.org/en/stable/docs/software/commandbased/convenience-features.html
public class ExampleAuto extends SequentialCommandGroup {
  /** Creates a new ExampleAuto. */
  public ExampleAuto(CANDriveSubsystem driveSubsystem, CANFuelSubsystem ballSubsystem) {
    // Add your commands in the addCommands() call, e.g.
    // addCommands(new FooCommand(), new BarCommand());
    addCommands(
    // Drive backwards for .25 seconds. The driveArcadeAuto command factory
    // intentionally creates a command which does not end which allows us to control
    // the timing using the withTimeout decorator
    new AutoDrive(driveSubsystem,xSpeed:0.5,  zRotation:0.0).withTimeout(seconds:.25),
    // Spin up the launcher for 1 second and then launch balls for 9 seconds, for a
    // total of 10 seconds
    new Launch(ballSubsystem).withTimeout(seconds:10));
  }
```

These commands are both Sequential Command Groups. A Sequential Command Group is a sequence of commands that execute in order. These commands generally contain a single method in the constructor called addCommands() where you specify the commands in the group as a comma separated list. For the LaunchSequence we spin up the launcher for 1 second before running the launch command. For the ExampleAuto, we drive the robot forward at half-speed for .25 seconds before launching for 10 seconds.

5.4 Inline Commands

This subsection will focus on the factory command-based framework. Commands are defined using WPILib inline commands and decorators.

5.4.1 Drive Command Factory

The drive command factory is a method in the Drive subsystem which creates a command.

```java
77    public Command driveArcade(DoubleSupplier xSpeed, DoubleSupplier zRotation) {
78      return this.run(
79          () -> drive.arcadeDrive(xSpeed.getAsDouble(), zRotation.getAsDouble()));
80    }
```

The “this.run()” notation means we are using the run() helper function from the SubsystemBase class which turns a Runnable into a command. In this case, we use lambda notation, the () ->, to turn a single method (the arcadeDrive method of our DifferentialDrive object) into a Runnable. By using the helper method on the subsystem, it automatically sets up the command to require this subsystem. You can also see that the parameters passed into the command are a special type called a DoubleSupplier rather than just a double. This is because this command factory is only called once, when setting up the command bindings, but we need the speed and rotation values to continually update while the command is running. Using a DoubleSupplier allows the arcadeDrive function to retrieve new values each iteration.

5.4.2 Fuel Command Factories

The fuel subsystem also has a couple of command factories that turn methods in the subsystem into commands.

```
93    public Command spinUpCommand() {
94      return this.run(() -> spinUp());
95    }
96
97    // A command factory to turn the launch method into a command that requires this
98    // subsystem
99    public Command launchCommand() {
100     return this.run(() -> launch());
101   }
```

5.4.3 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.

```
12  public final class Autos {
13    // Example autonomous command which drives forward for 1 second.
14    public static final Command exampleAuto(CANDriveSubsystem driveSubsystem, CANFuelSubsystem ballSubsystem) {
15      return new SequentialCommandGroup(
16          // Drive backwards for .25 seconds. The driveArcadeAuto command factory
17          // creates a command which does not end which allows us to control
18          // the timing using the withTimeout decorator
19          driveSubsystem.driveArcade(() -> 0.5, () -> 0).withTimeout(seconds:.25),
20          // Stop driving. This line uses the regular driveArcade command factory so it
21          // ends immediately after commanding the motors to stop
22          driveSubsystem.driveArcade(() -> 0, () -> 0),
23          // Spin up the launcher for 1 second and then launch balls for 9 seconds, for a
24          // total of 10 seconds
25          ballSubsystem.spinUpCommand().withTimeout(seconds:1),
26          ballSubsystem.launchCommand().withTimeout(seconds:9),
27          // Stop running the launcher
28          ballSubsystem.runOnce(() -> ballSubsystem.stop()));
29    }
```

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 forwards for 0.25 seconds at 50% power by using the WithTimeout decorator to set a timeout of 0.25 seconds on the driving command. The different types of command compositions that are built-in via decorators and factory methods are described on the Command Compositions page. It then stops driving, spins up for 1 second, launches for 9 seconds, then stops the fuel rollers.

5.5 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 provided constant names should pretty clearly describe what each is for.

5.6 Robot

This file is 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.

5.7 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.

5.7.1 Imports

```
5   package frc.robot;
6
7   import edu.wpi.first.wpilibj.smartdashboard.SendableChooser;
8   import edu.wpi.first.wpilibj2.command.Command;
9   import edu.wpi.first.wpilibj2.command.button.CommandXboxController;
10  import edu.wpi.first.wpilibj2.command.button.Trigger;
11
12  import static frc.robot.Constants.OperatorConstants.*;
13  import static frc.robot.Constants.FuelConstants.*;
14  import frc.robot.commands.Autos;
15  import frc.robot.subsystems.CANDriveSubsystem;
16  import frc.robot.subsystems.CANFuelSubsystem;
```

The first section of code is the imports. In this case we need to import the “SendableChooser” for selecting auto modes, some elements from the commands module, some sections of the constants file from our project, and then all of our commands and subsystems.

5.7.2 Class definition and Constructor

```
25  public class RobotContainer {
26    // The robot's subsystems
27    private final CANDriveSubsystem driveSubsystem = new CANDriveSubsystem();
28    private final CANFuelSubsystem ballSubsystem = new CANFuelSubsystem();
29
30    // The driver's controller
31    private final CommandXboxController driverController = new CommandXboxController(
32        DRIVER_CONTROLLER_PORT);
33
34    // The operator's controller
35    private final CommandXboxController operatorController = new CommandXboxController(
36        OPERATOR_CONTROLLER_PORT);
37
38    // The autonomous chooser
39    private final SendableChooser<Command> autoChooser = new SendableChooser<>();
```

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

```
public RobotContainer() {
  configureBindings();

  // Set the options to show up in the Dashboard for selecting auto modes. If you
  // add additional auto modes you can add additional lines here with
  // autoChooser.addOption
  autoChooser.setDefaultOption(name:"Autonomous", Autos.exampleAuto(driveSubsystem, ballSubsystem));
}
```

Next is the constructor which contains a call to configureBindings() which we will cover below. This method is used to set up button bindings and default commands. You could put all this code directly in the constructor, but as your robot and controls become more complex, it's often helpful to split things up for clarity.

The other thing the constructor does is add the one autonomous mode to the dashboard chooser. Additional autonomous mode options can be added to this chooser using the adoption() method and then you could select which one to run from SmartDashboard, Shuffleboard, Elastic or other 3rd party dashboards.

5.7.3 configureBindings()

This method sets up the relationships between our controls and commands. In this section all examples are shown with the inline project as the bindings for that project are a bit more complex. The bindings in the class project are in the same order.

```
67      operatorController.leftBumper()
68          .whileTrue(fuelSubsystem.runEnd(() -> fuelSubsystem.intake(), () -> fuelSubsystem.stop()));
```

The first section sets up a binding for the 'LeftBumper' button on the operator controller to intake fuel. The CommandXboxController class contains methods for each button which return "Trigger" objects. These "Trigger" objects then have further methods that narrow down the behavior we want to control the command such as toggles, initiating on change, or running only while the Trigger is true or false. In this instance we use "whileTrue()" to have our command run while the button is being held and stop when it is released. For the inline project, the command to run is created inline using the runEnd() method from the subsystem. As noted earlier, using the subsystem built-in factories makes sure the commands already require that subsystem. The runEnd() factory takes two Runnables, one to run continuously while the command is running and one to run when the command ends. In order to turn the methods we want to execute into Runnables, we need to use a lambda function.

```
71      operatorController.rightBumper()
72          .whileTrue(fuelSubsystem.spinUpCommand().withTimeout(SPIN_UP_SECONDS)
73              .andThen(fuelSubsystem.launchCommand())
74              .finallyDo(() -> fuelSubsystem.stop()));
```

Next is the binding for the 'RightBumper' button to launch fuel. In the class project, this binding looks pretty similar, with the additional logic tucked away in the command classes. In the inline project, we need to build up in the additional logic in the binding using command decorators (or compositions). We start with the first command we want to run when the button is pressed, spinUpCommand(). Then we use the withTimeout decorator to end that command after a certain amount of time. We modify the result with the andThen() decorator to specify that after the spinUpCommand ends we want to run the launchCommand(). Lastly, we use the finallyDo() decorator to specify a Runnable to run no matter what when the command ends (such as when the button is let go) to stop the fuel rollers.

```
87      driveSubsystem.setDefaultCommand(
88          driveSubsystem.driveArcade(
89              () -> -driverController.getLeftY() * DRIVE_SCALING,
90              () -> -driverController.getRightX() * ROTATION_SCALING));
```

The provided code also sets default commands in the configureBindings method. The choice of where to do this is personal preference. As your robot and controls get more complex, you may prefer to put the default command mapping in the constructor or even split it into it's own method called from the constructor.

Next, we set up the default command 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.

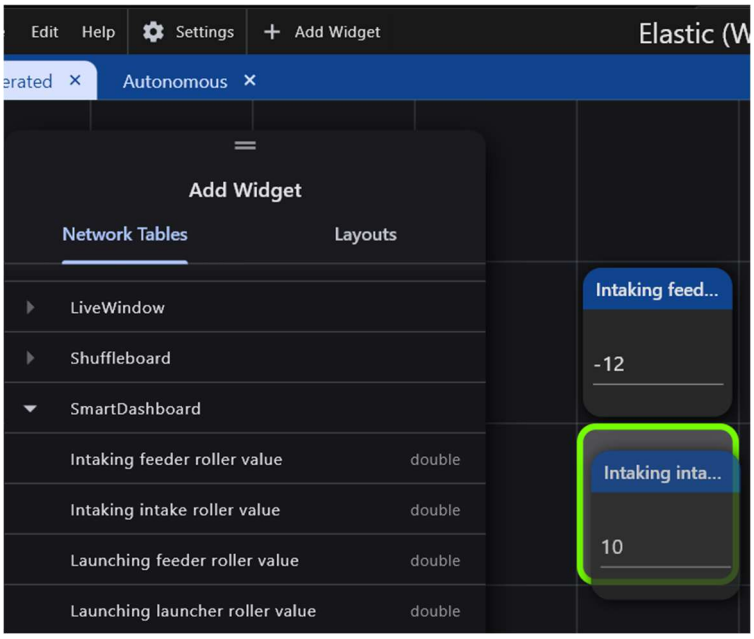
For the forward/back movement we pass in the value from the Y-axis (vertical) of the left stick of the controller, but we negate it. This is because joysticks generally define pushing the stick away from you

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.

6 Both values are scaled down by different factors to make the robot more controllable. As your drivers get more comfortable, you can experiment with increasing these constants, or try some of the alternative options covered in the Tuning Robot Values

As mentioned in the code walkthrough, many of the constants for the fuel subsystem have default values written to Network Tables and then read the value from there during execution. This allows you to easily tweak the value currently being used by the robot to try to tune robot behavior.

To access these values, open the Driver Station, then use the Dashboard Type dropdown on the Settings (gear) tab to select SmartDashboard, Shuffleboard, or Elastic (these are all installed by the WPILib installer). SmartDashboard and Shuffleboard are a little easier to use immediately as the values will appear automatically, but both are expected to be removed in future seasons due to lack of maintainers. To see the values in Elastic, click the Add Widget button at the top of the screen, expand the SmartDashboard section, then click and drag each value out onto the main display.



There are 5 constants used by the fuel system that are published to the dashboard, all of which are values in volts provided to the specified roller:

- Intaking intake roller value: Corresponds to the speed of the intake roller when intaking. A larger value spins the roller faster which will intake the fuel more rapidly, but may increase the chance of the fuel 'popcorning' directly up and out of the launcher.
- Intaking feeder roller value: Corresponds to the speed of the feeder roller when intaking. The value is negative so that the feeder spins the opposite way from when the robot is launching. In our testing the minimum value of -12V worked best but you may wish to test on your own, especially if you are modifying the intake roller value.
- Launching feeder roller value: Corresponds to the speed of the feeder roller during launching. A larger value will feed fuel closer together and impart more speed before reaching the launcher, but may not give the launcher enough time to get back up to speed. If you are seeing inconsistent distance when launching multiple fuel, you may be feeding too quickly.
- Launching launch roller value: Corresponds to the speed of the launcher wheels during launching. A larger value will launch the fuel farther, a smaller value will launch the fuel shorter.
- Spin-up feeder roller value: Corresponds to the speed of the feeder roller while the launcher wheel is spinning up. The feeder runs in reverse during the spin-up to ensure Fuel do not accidentally enter the launcher. In our testing the exact value was not particularly important, but you may wish to experiment for yourself to see if it has any effect.

To tune these values while running, simply click the box to select it, type in a new value, then use the tab key to move out of the box and send the value to the robot (using Enter will disable the robot if it is enabled). If you find a new value you like, make sure to go to the Constants file and update the default so the code will use the new value the next time you restart the robot.

Making Changes section below.

The inline project doesn't need a default command for the fuel rollers as each command binding stops the rollers when it ends. For the classes project we use a default command to stop the rollers as the spinUp command doesn't stop the rollers (so it can transition right into launch) and specifying the command group in a class file doesn't provide as simple of a way to specify interrupted behavior like the finallyDo decorator.

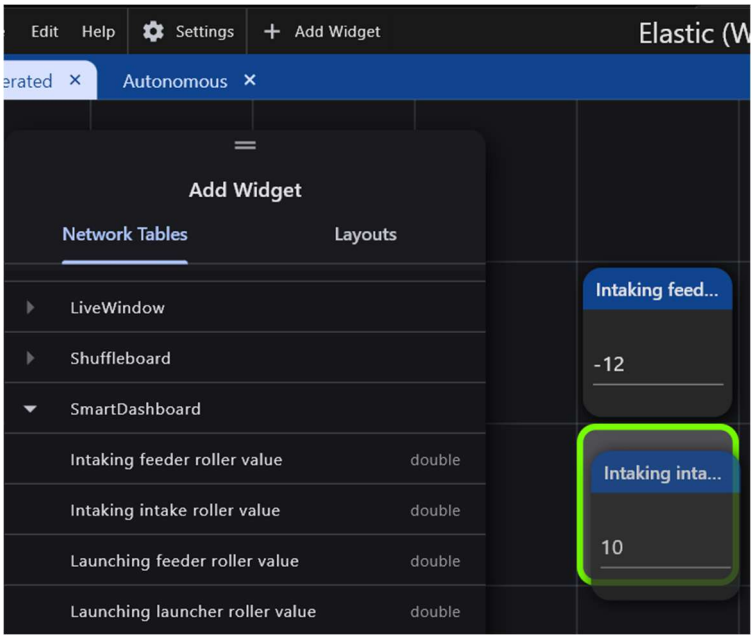
```
84      fuelSubsystem.setDefaultCommand(fuelSubsystem.run(() -> fuelSubsystem.stop()));
```

7 Tuning Robot Values

As mentioned in the code walkthrough, many of the constants for the fuel subsystem have default values written to Network Tables and then read the value from there during execution. This allows you to easily tweak the value currently being used by the robot to try to tune robot behavior.

To access these values, open the Driver Station, then use the Dashboard Type dropdown on the Settings (gear) tab to select SmartDashboard, Shuffleboard, or Elastic (these are all installed by the WPILib installer). SmartDashboard and Shuffleboard are a little easier to use immediately as the values

will appear automatically, but both are expected to be removed in future seasons due to lack of maintainers. To see the values in Elastic, click the Add Widget button at the top of the screen, expand the SmartDashboard section, then click and drag each value out onto the main display.



There are 5 constants used by the fuel system that are published to the dashboard, all of which are values in volts provided to the specified roller:

- Intaking intake roller value: Corresponds to the speed of the intake roller when intaking. A larger value spins the roller faster which will intake the fuel more rapidly, but may increase the chance of the fuel 'popcorning' directly up and out of the launcher.
- Intaking feeder roller value: Corresponds to the speed of the feeder roller when intaking. The value is negative so that the feeder spins the opposite way from when the robot is launching. In our testing the minimum value of -12V worked best but you may wish to test on your own, especially if you are modifying the intake roller value.
- Launching feeder roller value: Corresponds to the speed of the feeder roller during launching. A larger value will feed fuel closer together and impart more speed before reaching the launcher, but may not give the launcher enough time to get back up to speed. If you are seeing inconsistent distance when launching multiple fuel, you may be feeding too quickly.
- Launching launch roller value: Corresponds to the speed of the launcher wheels during launching. A larger value will launch the fuel farther, a smaller value will launch the fuel shorter.
- Spin-up feeder roller value: Corresponds to the speed of the feeder roller while the launcher wheel is spinning up. The feeder runs in reverse during the spin-up to ensure Fuel do not accidentally enter the launcher. In our testing the exact value was not particularly important, but you may wish to experiment for yourself to see if it has any effect.

To tune these values while running, simply click the box to select it, type in a new value, then use the tab key to move out of the box and send the value to the robot (using Enter will disable the robot if it is

enabled). If you find a new value you like, make sure to go to the Constants file and update the default so the code will use the new value the next time you restart the robot.

8 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. Each item is shown with an example image from one of the two projects, but the same concept should be portable to the other project type.

8.1 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 2026 KitBot do not end (isFinished always returns false) so they should generally only be used with the whileTrue() behavior, but changing which buttons they map to can be done very simply.

The button mappings in the example code are done in 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 has further methods that can be used to specify a behavior for the binding.

As provided the code connects the **left bumper** button to the intake action. To change this, simply change the leftBumper() helper method 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()

```
69      //before
70      operatorController.leftBumper().whileTrue(new Intake(fuelSubsystem));
71      //after
72      operatorController.x().whileTrue(new Intake(fuelSubsystem));
```

The lines for the inline command project will look a little different, but the premise is the same.

8.2 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 in two different places depending on the project:

- For the class project: In the execute() method of the Drive command class

- For the inline project: when setting up the drivetrain default command at the end of the 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 **float** in the XboxController API Doc. 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 in the inline project

```
87      driveSubsystem.setDefaultCommand(
88          driveSubsystem.driveArcade(
89              () -> -driverController.getRightY() * DRIVE_SCALING,
90              () -> -driverController.getRightX() * ROTATION_SCALING));
```

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 both axes to be cubed in the classes project

```
37    public void execute() {
38      driveSubsystem.driveArcade(Math.pow(-controller.getLeftY() * DRIVE_SCALING, b:3),
39      Math.pow(-controller.getRightX() * ROTATION_SCALING, b:3));
40    }
```

```
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 in the inline project

```
87      driveSubsystem.setDefaultCommand(
88          driveSubsystem.driveArcade(
89              () -> -driverController.getLeftY() *
90              (driverController.getHID().getRightBumperButton() ? 1: 0.5),
91              () -> -driverController.getRightX() * ROTATION_SCALING));
```

This example uses a shorthand if-else construction 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 pass the full value, if not we multiply it by .5. The code also uses a method called "getHID()" on the CommandXboxController; this method gives use the underlying XboxController object which we use to get the direct Boolean value of the button instead of the Trigger objects that come from the CommandXboxController class.

8.3 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. For some drivers this control is more natural, especially for precisely controlling how the robot moves when driving arcs. To make this change, you'll have to reach beyond RobotContainer as the provided drivetrain subsystems don't expose a tank drive method or command. In the drivetrain subsystem make a new method (for classes) or command factory (for inline) called tankDrive(). This method should look a lot like the arcadeDrive method. Then, modify the default command mapping in RobotContainer (for inline) or the Drive command (for classes) to use this new method with the appropriate joystick axis (note that this is likely different axes than arcade).

Example: changing to tank drive in the class project by creating a method in the subsystem and modifying the Drive command

```
77    public void driveTank (double leftSpeed, double rightSpeed) {
78      drive.tankDrive(leftSpeed, rightSpeed);
79    }
```

```
37  public void execute() {
38    driveSubsystem.driveTank(-controller.getLeftY() * DRIVE_SCALING, -controller.getRightY() * DRIVE_SCALING);
39  }
```

8.4 Developing Autonomous Routines

The provided code contains a simple autonomous mode that drives forward at ½ power for 0.25 seconds to space the robot away from the Hub, then launches for 10 seconds to attempt to launch the preloaded Fuel. 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. These projects already include the SendableChooser with the single option, ready for you to extend with more choices.