BTHS Robotics – Essential Software and Hardware Introduction for Programmers

Table of Contents

BTHS Robotics – Essential Software and Hardware Introduction for Programmers 1
Essential Software / Libraries 2
Essential Hardware 3
CAN, Motor Controllers, and Encoders (Hardware Essentials CONT.) 4
Encoders 5
- Incremental Encoder example 5
- Absolute Encoder example 5

WPILib Intro 7
- Robot.java 7
- MINI EXAMPLE 8
- Constants.java 9
- RobotContainer.java 10

Subsystems and Commands 11
- Subsystems 11
- Commands 15
  - Command example using the claw 15
- OpenClaw Command 16
  - About *addRequirements()* and interruption 17
- Scheduling and using OpenClawCommand 18

Binding Commands to PS4 Buttons 20
- PS4 Controller in RobotContainer 20
- Default Commands 21
- A visual example for the Command-based system 22

Essential Software / Libraries

Driver Station: Primary interface between the robot and the computer. Lets the user enable / disable the robot in its different modes (autonomous -> robot moves in preprogrammed paths without user inputs, teleoperated -> robot moves from user input), and allows access to joystick values in the robot's code, from joysticks connected to the computer. https://docs.wpilib.org/en/stable/docs/software/driverstation/dri ver-station.html#operation-tab (please view).

WpiLib: Standard software library used by all FRC teams to program their robots. Is supported for both Java and C++ *(we use Java)*, and contains all the tools, classes, and subroutines for interfacing with various parts of the FRC control system (sensors, motor controllers, and **Driver Station**) https://docs.wpilib.org/en/stable/docs/zero-torobot/introduction.html (WPILIB docs (we use this all the time)).

WpiLib VSCode Extension: All our code is written in vscode, and the WpiLib extension is a part of the WpiLib library that makes is super easy to deploy (to the robot)/test code from vscode. It also makes creating new FRC projects super simple. https://docs.wpilib.org/en/stable/docs/software/vscodeoverview/vscode-basics.html

Github: All our code is stored online in this Github organization: https://github.com/orgs/Team334/repositories (new people will be added).

Essential Hardware

Battery: 12 Volt battery that supplies the robot with power.

RoboRIO: The "brain" of the robot. All our written Java code gets deployed and runs on the RoboRIO.

Robot Signal Light: A simple light connected to the RoboRIO to indicate whether the robot has been enabled / disabled in **Driver Station**.

PDP (Power Distribution Panel): Piece of hardware by CTRE (Cross The Road Electronics - builds a lot of the FRC hardware teams use), that is used to distribute power from the robot battery to devices on the robot (sensors, motors, etc.)

VRM (Voltage Regulator Module): Acts like a power distribution panel, but for devices that need specific regulated voltage (camera, some sensors)

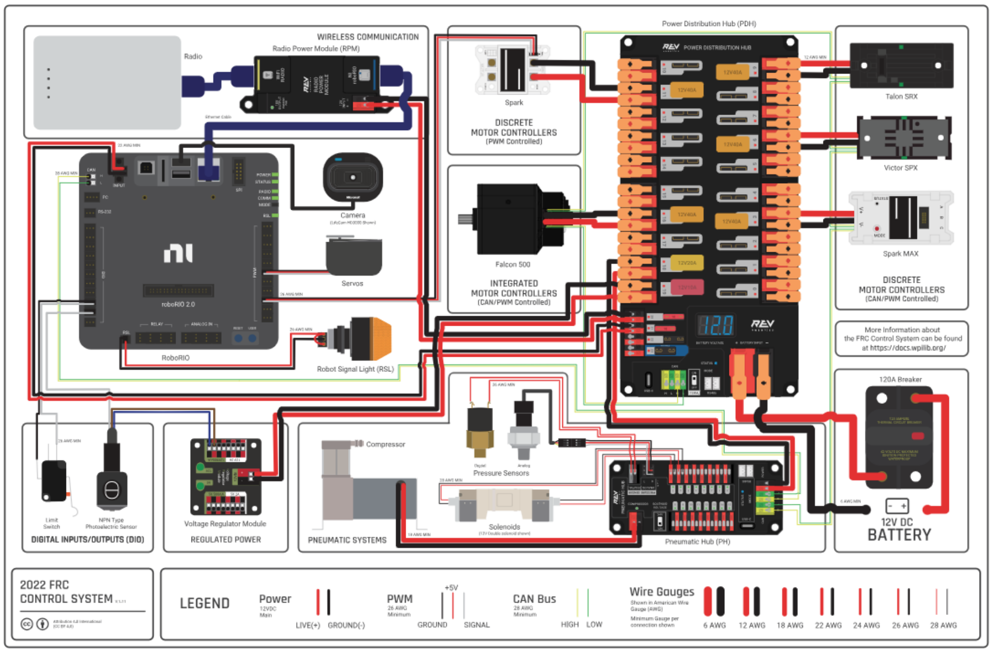
Circuit Breaker: The robot's power on / off switch, connects battery to the **PDP**.

Radio: Radio / router that allows Driver Station to connect wirelessly to the robot and upload code to the RoboRIO.

Limelight: Vision camera that connects to the Radio. (more on vision later)

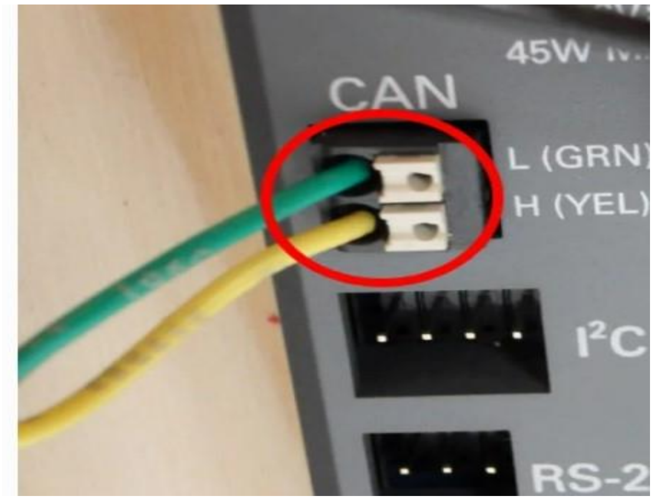
(we haven't used the pneumatic hub / system yet that's in the image below, but we might end up using it depending on what the robot will be this season)

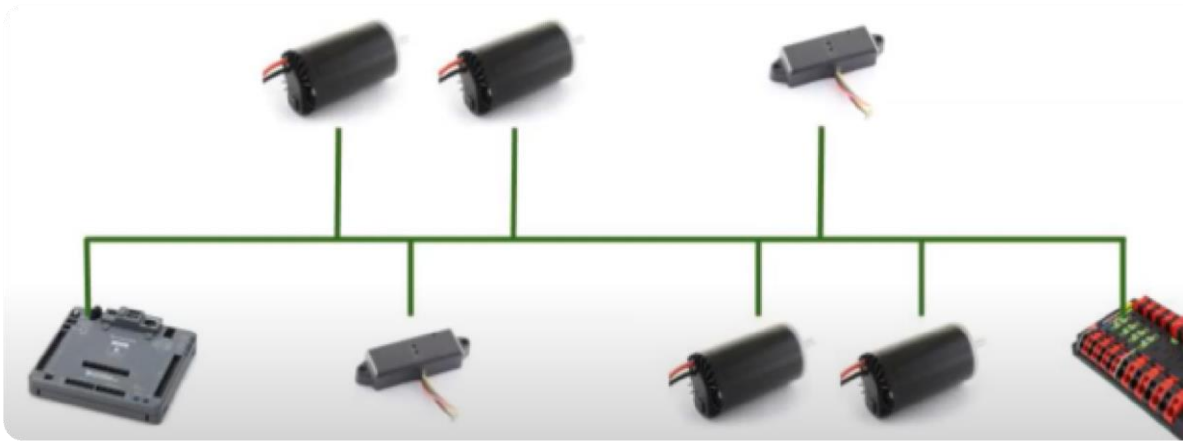
Hardware overview with pictures: https://docs.wpilib.org/en/stable/docs/controls-overviews/controlsystem-hardware.html
Limelight: https://limelightvision.io/



CAN, Motor Controllers, and Encoders (Hardware Essentials CONT.)

CAN Bus (!Really Important!): A two-wire network used for communication between the RoboRIO and peripherals. The network starts at the RoboRIO, and CAN devices (**TalonFX** motors/controllers, **CANCoder** encoders, etc.) can be added onto the two-wire network in a "daisy-chain" manner. Typically, a yellow and green wire are used for the CAN Bus *(as shown in the diagram above)*.




(CAN Bus network)

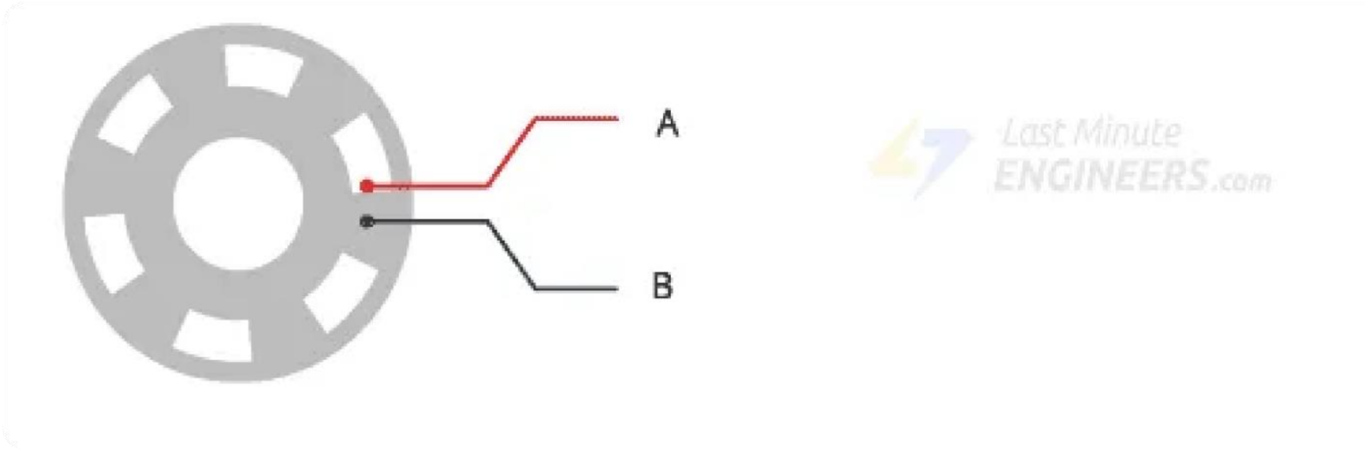
*Since all CAN devices are on the same network, they need to be distinguished when they are accessed in the Java code. Each CAN device has a unique **CAN ID**, and in the code, each device is accessed through its unique ID. (more on that later)*

Encoders

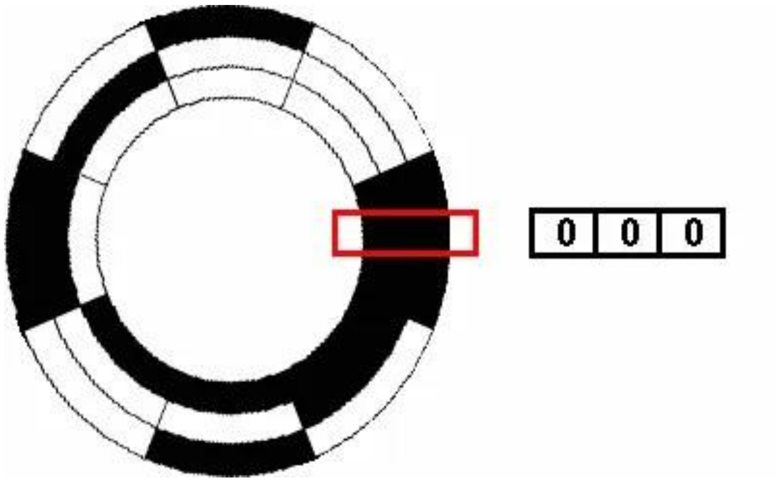
Encoder (in general): An electrical device that measures the rotation / angular position of a shaft.
Two kinds of encoders: Incremental and Absolute.
Incremental Encoder -> Measures the rotation of a shaft.
Rotation is broken into "ticks", and a revolution (full rotation) of the encoder is a certain number of ticks (i.e. 2048 ticks per revolution in the TalonFX built-in encoders). The ticks increase as the encoder spins in one direction and decrease as it spins in the other. **Absolute Encoder** -> Measures the angular position of a shaft (ex: arm shaft is at *90 degrees*). Every position has a unique value on the encoder, and absolute encoders don't lose their position if the robot is turned off and back on.

https://docs.wpilib.org/en/stable/docs/hardware/sensors/encoder s-hardware.html - (wpilib explanation)

Incremental Encoder example
(in this case a full revolution has 6 "ticks")



Absolute Encoder example
(every position has a unique value)



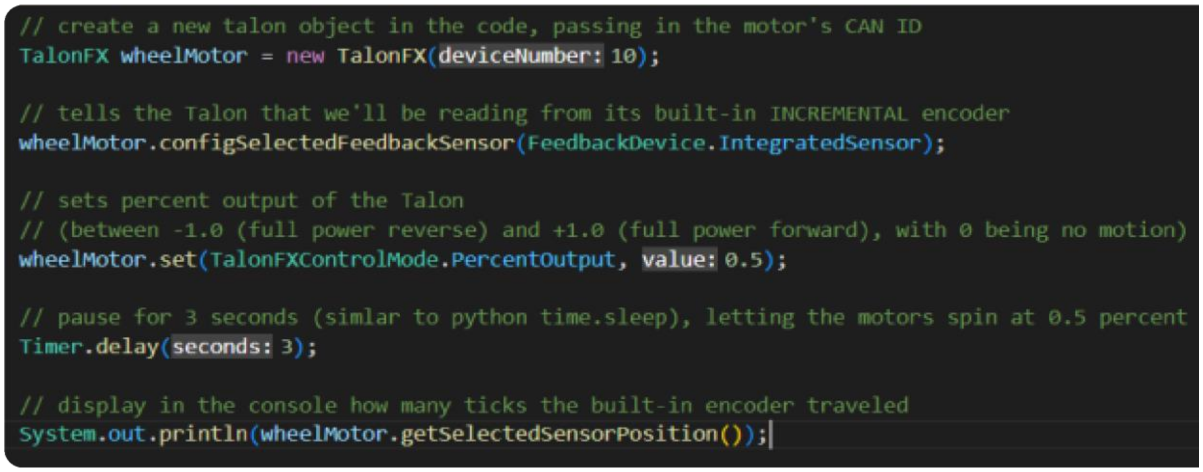
TalonFX *(also called Falcon 500)*: An electric motor, also built by CTRE, that has a built-in **encoder** and CAN *motor controller* (controls the motor based on received data from the RIO on the CAN Bus, and sends data about the motor / builtin encoder back to the RIO on the CAN Bus). The Talon is a "smart motor", and has a built-in braking system as well as many settings that can be configured (i.e., reverse the motor). To control the motor in our code, CTRE has a special Java library for their devices, and a class (called TalonFX) exists for the motor.

https://store.ctr-electronics.com/falcon-500-powered-by-talon-fx/ - page on the CTRE store

CANCoder: An encoder built by CTRE that is operated through the CAN Bus (like the TalonFX). The encoder can be set to either *absolute* or *incremental* through configuration (also has many settings like the talon). Like the TalonFXs, the CTRE Java library has a class for the device.

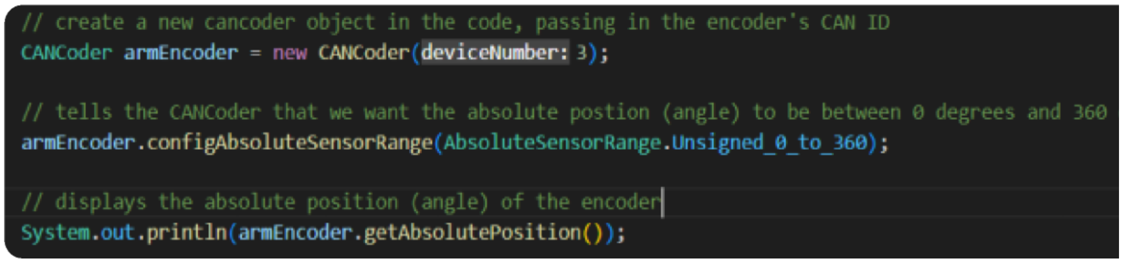
https://store.ctr-electronics.com/cancoder - page on the CTRE store

Example code that spins and reads from a TalonFX *(you will use these functions during season)*



```java
// create a new talon object in the code, passing in the motor's CAN ID
TalonFX wheelMotor = new TalonFX(deviceNumber: 10);

// tells the Talon that we'll be reading from its built-in INCREMENTAL encoder
wheelMotor.configSelectedFeedbackSensor(FeedbackDevice.IntegratedSensor);

// sets percent output of the Talon
// (between -1.0 (full power reverse) and +1.0 (full power forward), with 0 being no motion)
wheelMotor.set(TalonFXControlMode.PercentOutput, value: 0.5);

// pause for 3 seconds (simlar to python time.sleep), letting the motors spin at 0.5 percent
Timer.delay(seconds: 3);

// display in the console how many ticks the built-in encoder traveled
System.out.println(wheelMotor.getSelectedSensorPosition());
```

Example code that reads from a CANCoder *(you will use these functions during season)*

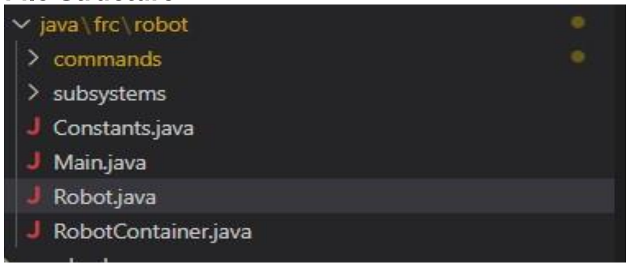


```java
// create a new cancoder object in the code, passing in the encoder's CAN ID
CANCoder armEncoder = new CANCoder(deviceNumber: 3);

// tells the CANCoder that we want the absolute postion (angle) to be between 0 degrees and 360
armEncoder.configAbsoluteSensorRange(AbsoluteSensorRange.Unsigned_0_to_360);

// displays the absolute position (angle) of the encoder
System.out.println(armEncoder.getAbsolutePosition());
```

WPILib Intro

We select the Command Based Robot template when creating a new coding project through the WpiLib VSCode Extension, with empty classes / methods for the robot that we fill with code. (A lot of the methods created by the extension have pre-written comments explaining what they do)

File Structure



```
java\frc\robot
  > commands
  > subsystems
  Constants.java
  Main.java
  Robot.java
  RobotContainer.java
```

For a detailed explanation, see this page on the wpilib docs about the structure: https://docs.wpilib.org/en/stable/docs/software/commandbased/s tructuring-command-based-project.html?present=#/4

Robot.java

Is the class / file that's responsible for the program's control flow on the robot, containing methods called periodically or during robot initialization. Important methods

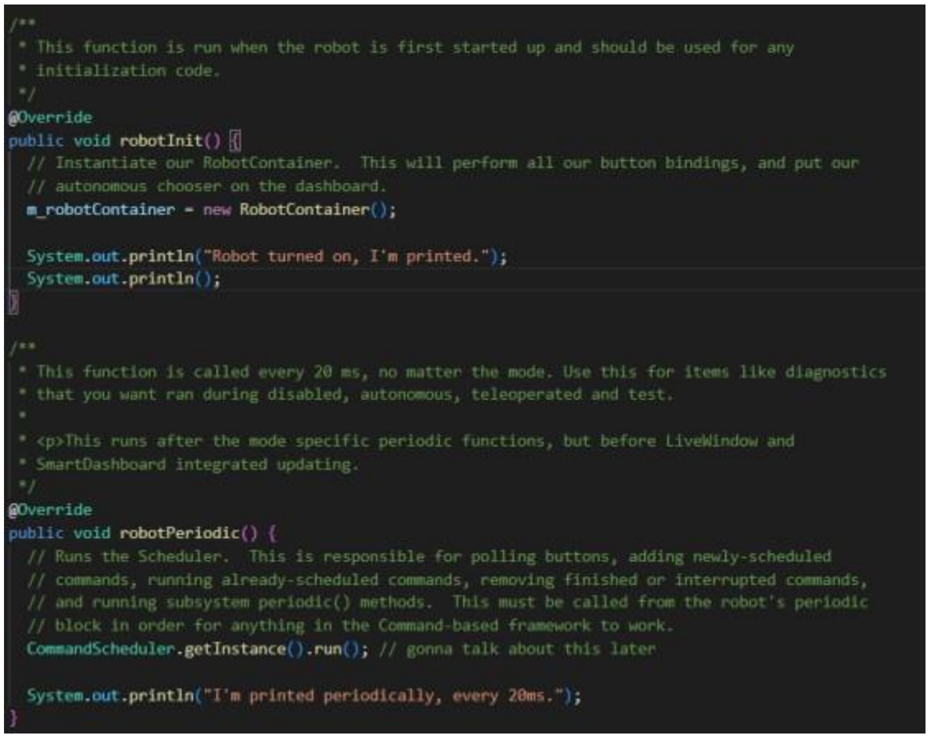
robotInit: Called ONCE when the robot is first started up (powered on).

robotPeriodic: Called PERIODICALLY every 20ms, no matter the mode (the method will still be called even if the robot is disabled in Driver Station).

autonomousInit: Called ONCE when autonomous mode is enabled in Driver Station. Autonomous *Command*(s) (more on that later) is typically scheduled in this method.

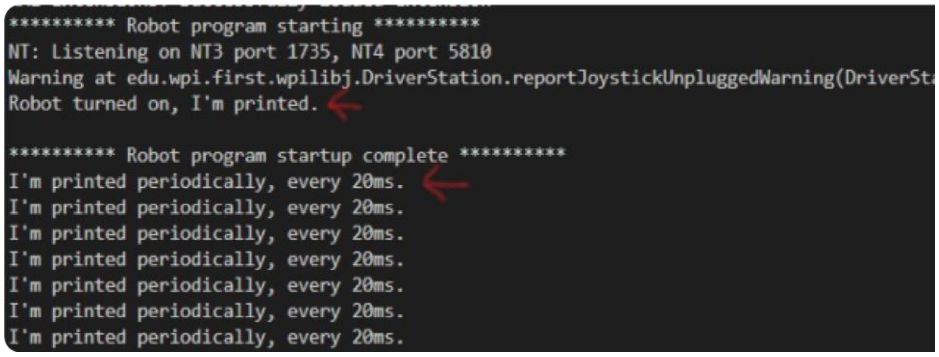
teleopInit: Called ONCE when teleop mode is enabled in Driver Station. Previously running Autonomous Command(s) is canceled typically.

MINI EXAMPLE



```java
/**
 * This function is run when the robot is first started up and should be used for any
 * initialization code.
 */
@Override
public void robotInit() {
  // Instantiate our RobotContainer.  This will perform all our button bindings, and put our
  // autonomous chooser on the dashboard.
  m_robotContainer = new RobotContainer();

  System.out.println("Robot turned on, I'm printed.");
  System.out.println();
}

/**
 * This function is called every 20 ms, no matter the mode. Use this for items like diagnostics
 * that you want ran during disabled, autonomous, teleoperated and test.
 *
 * <p>This runs after the mode specific periodic functions, but before LiveWindow and
 * SmartDashboard integrated updating.
 */
@Override
public void robotPeriodic() {
  // Runs the Scheduler.  This is responsible for polling buttons, adding newly-scheduled
  // commands, running already-scheduled commands, removing finished or interrupted commands,
  // and running subsystem periodic() methods.  This must be called from the robot's periodic
  // block in order for anything in the Command-based framework to work.
  CommandScheduler.getInstance().run(); // gonna talk about this later

  System.out.println("I'm printed periodically, every 20ms.");
}
```

Robot console output (contains messages about the robot from Wpilib, as well as anything we print in the code)



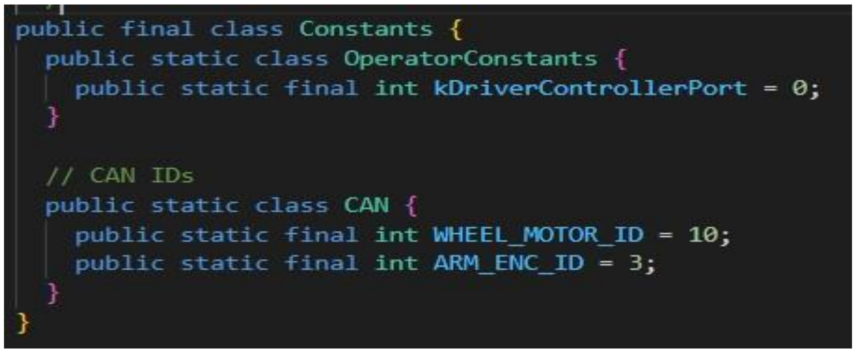
```
********** Robot program starting **********
NT: Listening on NT3 port 1735, NT4 port 5810
Warning at edu.wpi.first.wpilibj.DriverStation.reportJoystickUnpluggedWarning(DriverSt
Robot turned on, I'm printed.

********** Robot program startup complete **********
I'm printed periodically, every 20ms.
I'm printed periodically, every 20ms.
I'm printed periodically, every 20ms.
I'm printed periodically, every 20ms.
I'm printed periodically, every 20ms.
I'm printed periodically, every 20ms.
I'm printed periodically, every 20ms.
```

Constants.java

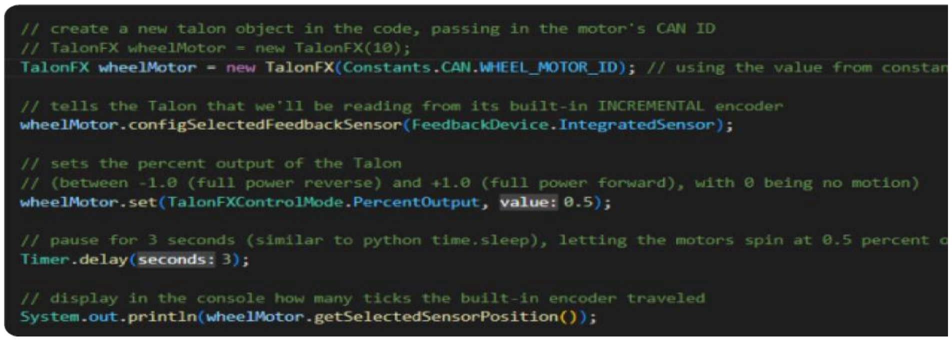
Class / file containing globally accessible constants about the robot (speeds, CAN IDs of motors, encoder offsets, etc.).

All the variables in Constants are constant -> they can't be modified after created, and they are public to all classes in the project. (created through java **public static final** keywords)

MINI EXAMPLE *(continuation of TalonFX / CANCoder example codes)*

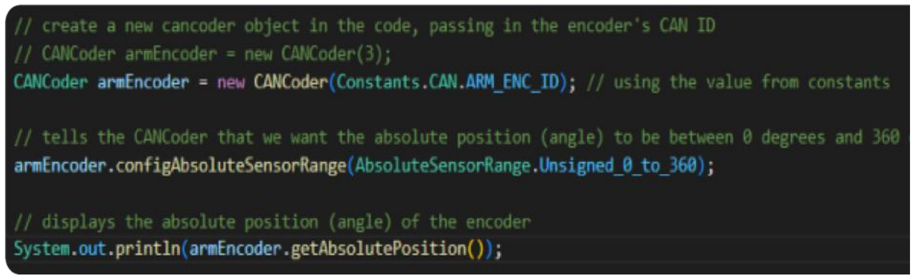


```java
public final class Constants {
  public static class OperatorConstants {
    public static final int kDriverControllerPort = 0;
  }

  // CAN IDs
  public static class CAN {
    public static final int WHEEL_MOTOR_ID = 10;
    public static final int ARM_ENC_ID = 3;
  }
}
```



```java
// create a new talon object in the code, passing in the motor's CAN ID
// TalonFX wheelMotor = new TalonFX(10);
TalonFX wheelMotor = new TalonFX(Constants.CAN.WHEEL_MOTOR_ID); // using the value from constan

// tells the Talon that we'll be reading from its built-in INCREMENTAL encoder
wheelMotor.configSelectedFeedbackSensor(FeedbackDevice.IntegratedSensor);

// sets the percent output of the Talon
// (between -1.0 (full power reverse) and +1.0 (full power forward), with 0 being no motion)
wheelMotor.set(TalonFXControlMode.PercentOutput, value: 0.5);

// pause for 3 seconds (similar to python time.sleep), letting the motors spin at 0.5 percent o
Timer.delay(seconds: 3);

// display in the console how many ticks the built-in encoder traveled
System.out.println(wheelMotor.getSelectedSensorPosition());
```

(TalonFX example code from last time using CAN ID defined in Constants)



```java
// create a new cancoder object in the code, passing in the encoder's CAN ID
// CANCoder armEncoder = new CANCoder(3);
CANCoder armEncoder = new CANCoder(Constants.CAN.ARM_ENC_ID); // using the value from constants

// tells the CANCoder that we want the absolute position (angle) to be between 0 degrees and 360
armEncoder.configAbsoluteSensorRange(AbsoluteSensorRange.Unsigned_0_to_360);

// displays the absolute position (angle) of the encoder
System.out.println(armEncoder.getAbsolutePosition());
```

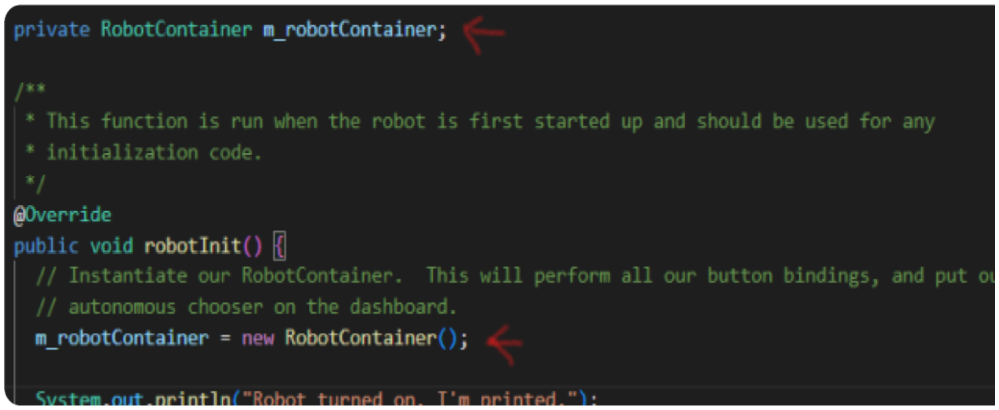
(CANCoder example code from last time using CAN ID defined in Constants)

RobotContainer.java

Is the class / file that contains the setup code for the command-based robot.

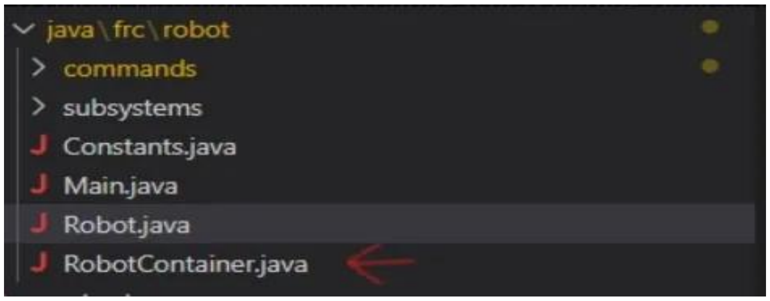
Commands, *Subsystem*s (more on that later), Autonomous Commands / routines, and bindings to controller (ps4) buttons / joysticks are created in this class *(more on that later)*.

Is instanced / used by the **Robot** class



```java
private RobotContainer m_robotContainer;

/**
 * This function is run when the robot is first started up and should be used for any
 * initialization code.
 */
@Override
public void robotInit() {
  // Instantiate our RobotContainer.  This will perform all our button bindings, and put ou
  // autonomous chooser on the dashboard.
  m_robotContainer = new RobotContainer();

  System.out.println("Robot turned on, I'm printed.");
```

*(located in the **RobotContainer.java** file of the project)*



```
java\frc\robot
  commands
  subsystems
  Constants.java
  Main.java
  Robot.java
  RobotContainer.java
```

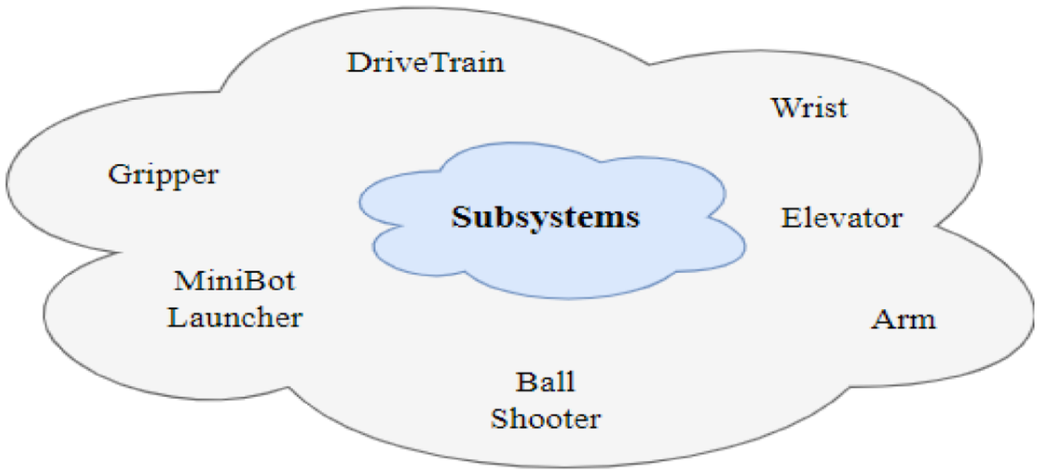
Subsystems and Commands

Subsystems and Commands are the basis of wpilib robot programming, and everyone will end up using them it no matter what part of the robot they work on.

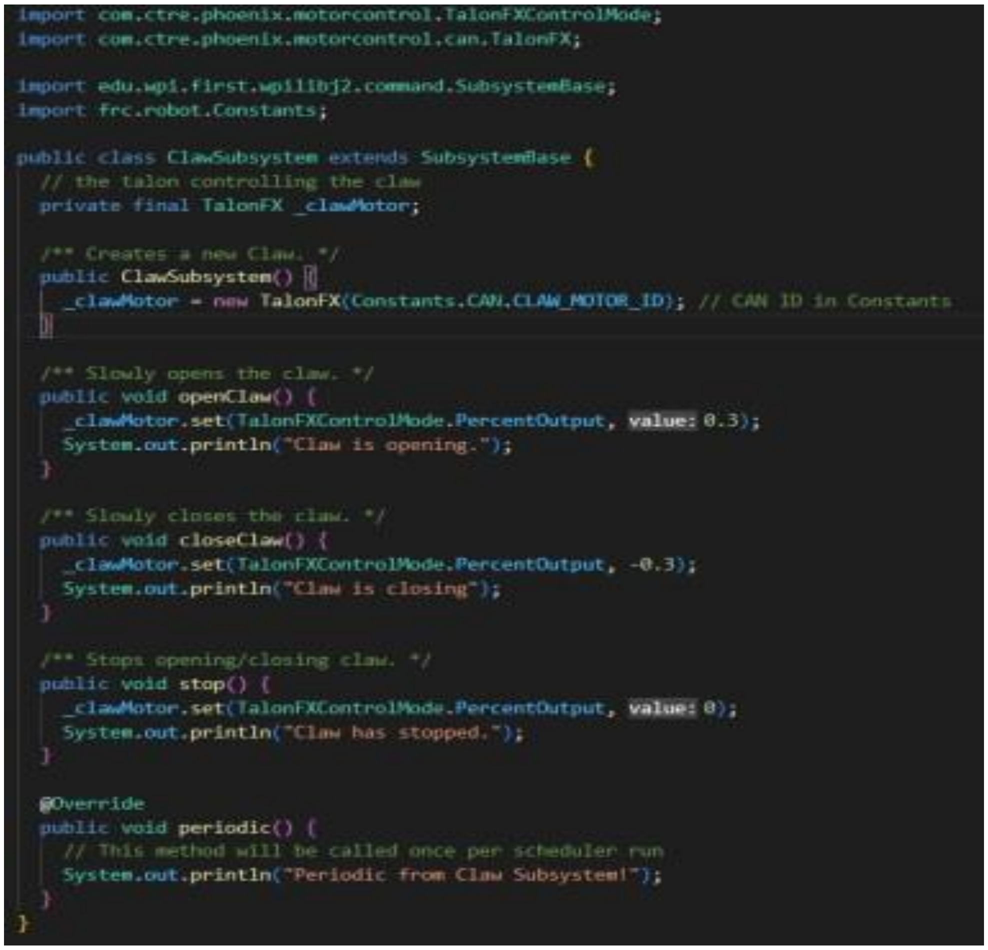
Subsystems

A subsystem represents an individual part / unit of the whole robot (ex: drivetrain, shooter, arm, etc). In code, a Subsystem is a class which is a collection of robot hardware which operates together as a unit. The robot hardware is hidden *(more on that later)* and the Subsystem is controlled through it's public methods (or functions). All Subsystems created by us in the code *extend* from the wpilib *SubsystemBase* class (don't worry too much about that).

Wpilib docs explanation:
https://docs.wpilib.org/en/stable/docs/software/commandbased/s ubsystems.html
General Subsystem examples from wpilib docs.



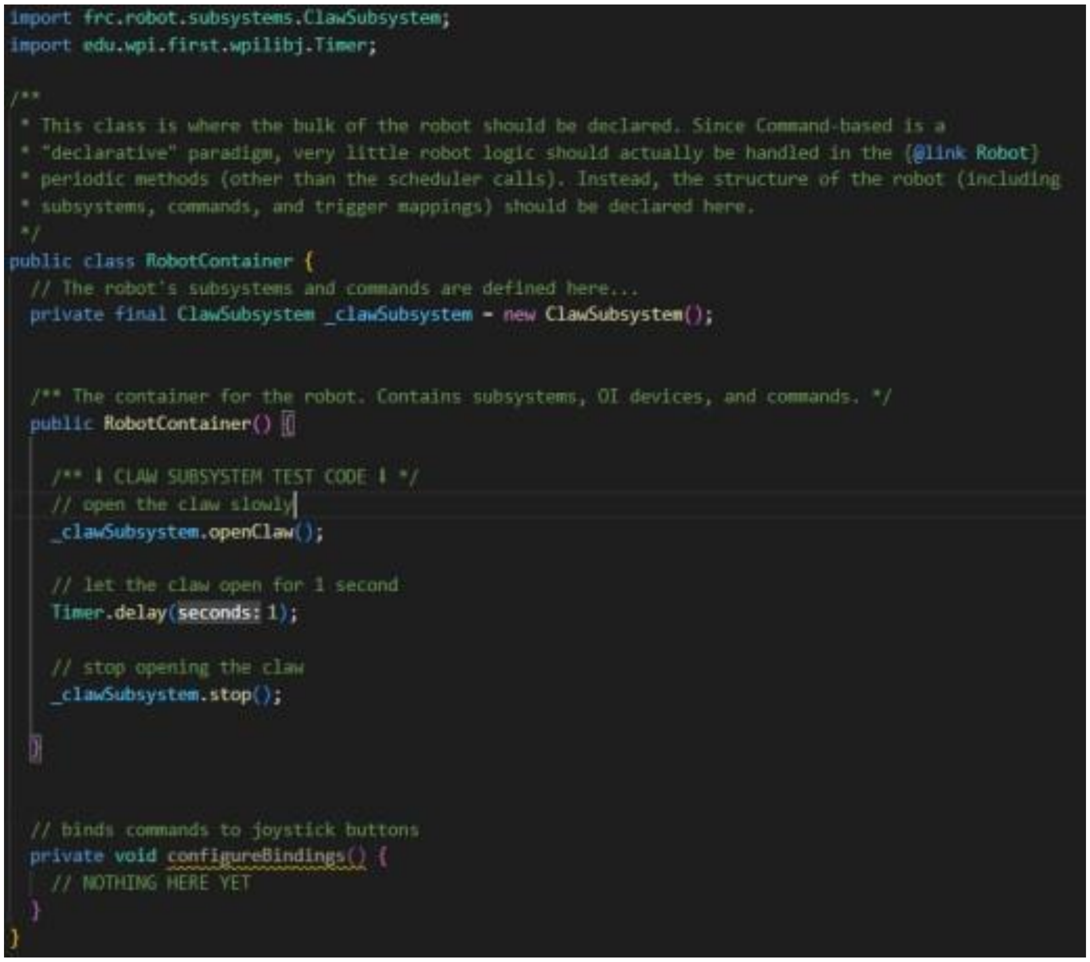
Example claw Subsystem: A subsystem for a claw where the whole mechanism is controlled with one TalonFX motor, called _*clawMotor*, which is hidden in the class through the *private* keyword (Java related topic). The subsystem has 3 accessible functions, *openClaw*, *closeClaw*, and *stop*, for spinning the claw talon, using percentage output *(talonfx example code is pinned in this channel for reference)*.



```java
import com.ctre.phoenix.motorcontrol.TalonFXControlMode;
import com.ctre.phoenix.motorcontrol.can.TalonFX;

import edu.wpi.first.wpilibj2.command.SubsystemBase;
import frc.robot.Constants;

public class ClawSubsystem extends SubsystemBase {
  // the talon controlling the claw
  private final TalonFX _clawMotor;

  /** Creates a new Claw. */
  public ClawSubsystem() {
    _clawMotor = new TalonFX(Constants.CAN.CLAW_MOTOR_ID); // CAN ID in Constants
  }

  /** Slowly opens the claw. */
  public void openClaw() {
    _clawMotor.set(TalonFXControlMode.PercentOutput, value: 0.3);
    System.out.println("Claw is opening.");
  }

  /** Slowly closes the claw. */
  public void closeClaw() {
    _clawMotor.set(TalonFXControlMode.PercentOutput, -0.3);
    System.out.println("Claw is closing");
  }

  /** Stops opening/closing claw. */
  public void stop() {
    _clawMotor.set(TalonFXControlMode.PercentOutput, value: 0);
    System.out.println("Claw has stopped.");
  }

  @Override
  public void periodic() {
    // This method will be called once per scheduler run
    System.out.println("Periodic from Claw Subsystem!");
  }
}
```

About "*periodic()*" - All Subsystem classes have a function called *periodic()* (inherited from SubsystemBase) which acts like *robotPeriodic()* from **Robot.java**. Like *robotPeriodic()*, *periodic()* will be called every 20ms no matter the selected mode (disabled, autonomous, or teleoperated) in Driver Station. Instancing the Claw subsystem and testing the subsystem's methods back in RobotContainer.

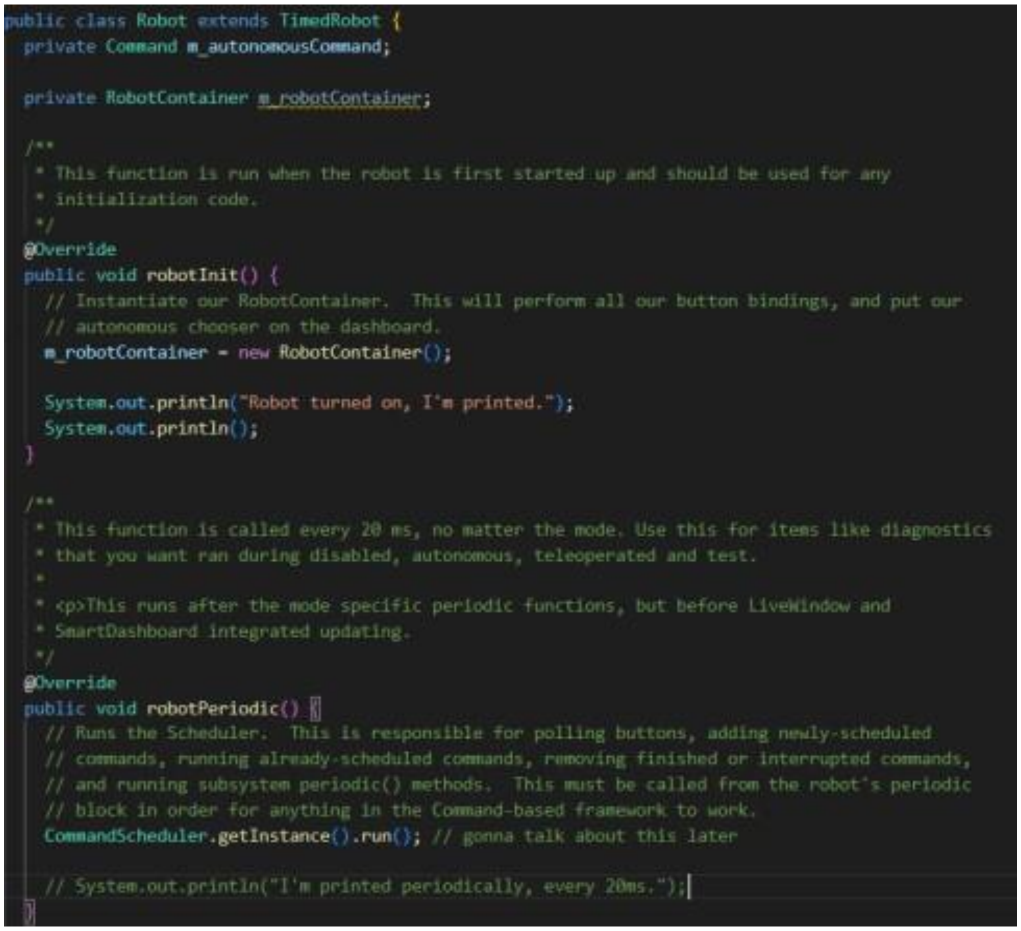
ClawSubsystem test code in RobotContainer



```java
import frc.robot.subsystems.ClawSubsystem;
import edu.wpi.first.wpilibj.Timer;

/**
 * This class is where the bulk of the robot should be declared. Since Command-based is a
 * "declarative" paradigm, very little robot logic should actually be handled in the {@link Robot}
 * periodic methods (other than the scheduler calls). Instead, the structure of the robot (including
 * subsystems, commands, and trigger mappings) should be declared here.
 */
public class RobotContainer {
  // The robot's subsystems and commands are defined here...
  private final ClawSubsystem _clawSubsystem = new ClawSubsystem();


  /** The container for the robot. Contains subsystems, OI devices, and commands. */
  public RobotContainer() {

    /** ↓ CLAW SUBSYSTEM TEST CODE ↓ */
    // open the claw slowly
    _clawSubsystem.openClaw();

    // let the claw open for 1 second
    Timer.delay(seconds: 1);

    // stop opening the claw
    _clawSubsystem.stop();

  }


  // binds commands to joystick buttons
  private void configureBindings() {
    // NOTHING HERE YET
  }
}
```

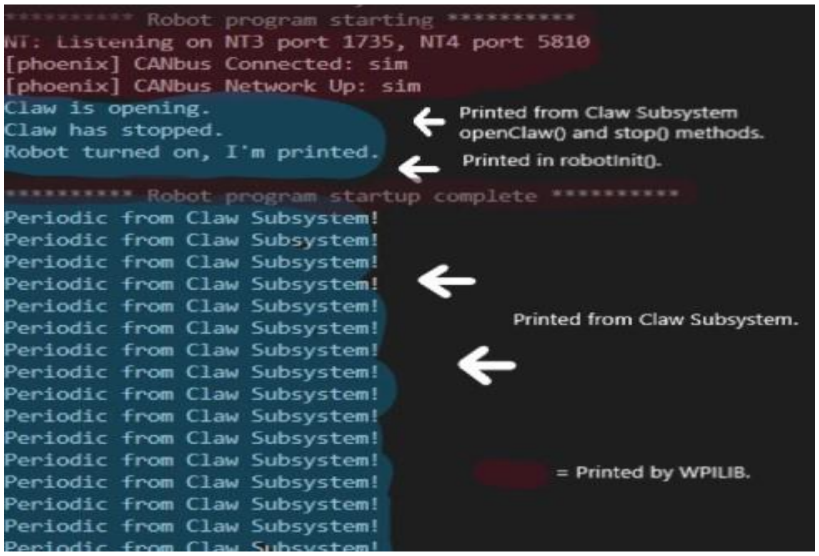
(Robot.java instancing the RobotContainer *(same code as last time just commented the print statement in robotPeriodic())*)

Robot (updated)



```java
public class Robot extends TimedRobot {
  private Command m_autonomousCommand;

  private RobotContainer m_robotContainer;

  /**
   * This function is run when the robot is first started up and should be used for any
   * initialization code.
   */
  @Override
  public void robotInit() {
    // Instantiate our RobotContainer.  This will perform all our button bindings, and put our
    // autonomous chooser on the dashboard.
    m_robotContainer = new RobotContainer();

    System.out.println("Robot turned on, I'm printed.");
    System.out.println();
  }

  /**
   * This function is called every 20 ms, no matter the mode. Use this for items like diagnostics
   * that you want ran during disabled, autonomous, teleoperated and test.
   *
   * <p>This runs after the mode specific periodic functions, but before LiveWindow and
   * SmartDashboard integrated updating.
   */
  @Override
  public void robotPeriodic() {
    // Runs the Scheduler.  This is responsible for polling buttons, adding newly-scheduled
    // commands, running already-scheduled commands, removing finished or interrupted commands,
    // and running subsystem periodic() methods.  This must be called from the robot's periodic
    // block in order for anything in the Command-based framework to work.
    CommandScheduler.getInstance().run(); // gonna talk about this later

    // System.out.println("I'm printed periodically, every 20ms.");
  }
```

Console output after deploying the code:



```
********** Robot program starting **********
NT: Listening on NT3 port 1735, NT4 port 5810
[phoenix] CANbus Connected: sim
[phoenix] CANbus Network Up: sim
Claw is opening.
Claw has stopped.
Robot turned on, I'm printed.
********** Robot program startup complete **********
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
Periodic from Claw Subsystem!
```

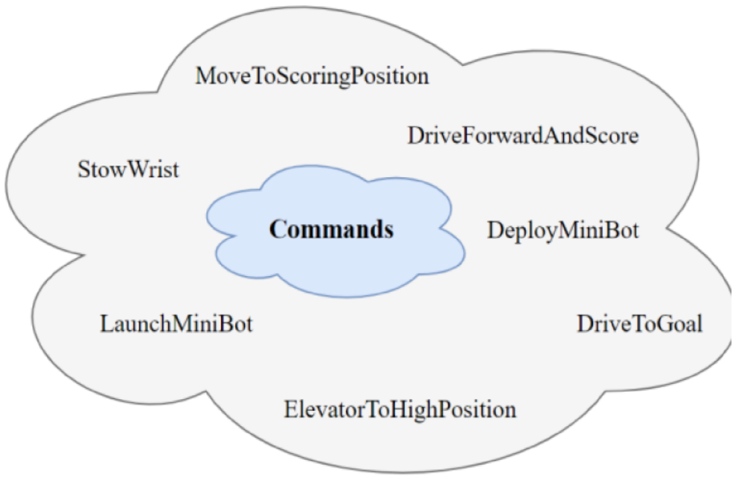
Printed from Claw Subsystem openClaw() and stop() methods.
Printed in robotInit().
Printed from Claw Subsystem.
🟥 = Printed by WPILIB.

During the season *-> Last season we broke up into groups with each group being responsible for programming a subsystem which we might do again this season.*

Commands

Commands represent actions that the robot can take, and they use Subsystems to perform those actions. Commands, like Subsystems, are represented as classes in code and inherit from wpilib's *CommandBase* class. **A Command is performed by being *scheduled*.**

Wpilib docs explanation: https://docs.wpilib.org/en/stable/docs/software/commandbased/c ommands.html
General Command examples from wpilib docs.



Commands inherit several methods from their parent class CommandBase, that define the Command's lifecycle

Command Lifecycle *initialize()*: Called ONCE when the Command gets scheduled.

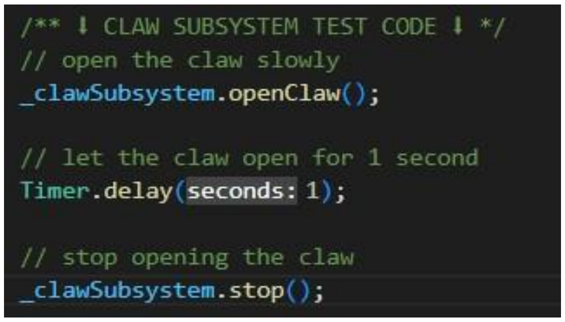
execute (): Called REPEATEDLY while the Command is scheduled.

end(): Called ONCE when the Command gets ended (either from being **interrupted** or from *isFinished* returning **true**).

isFinished(): Is called REPEATEDLY while the Command is scheduled. Returns a boolean that if **true**, the command is ended.

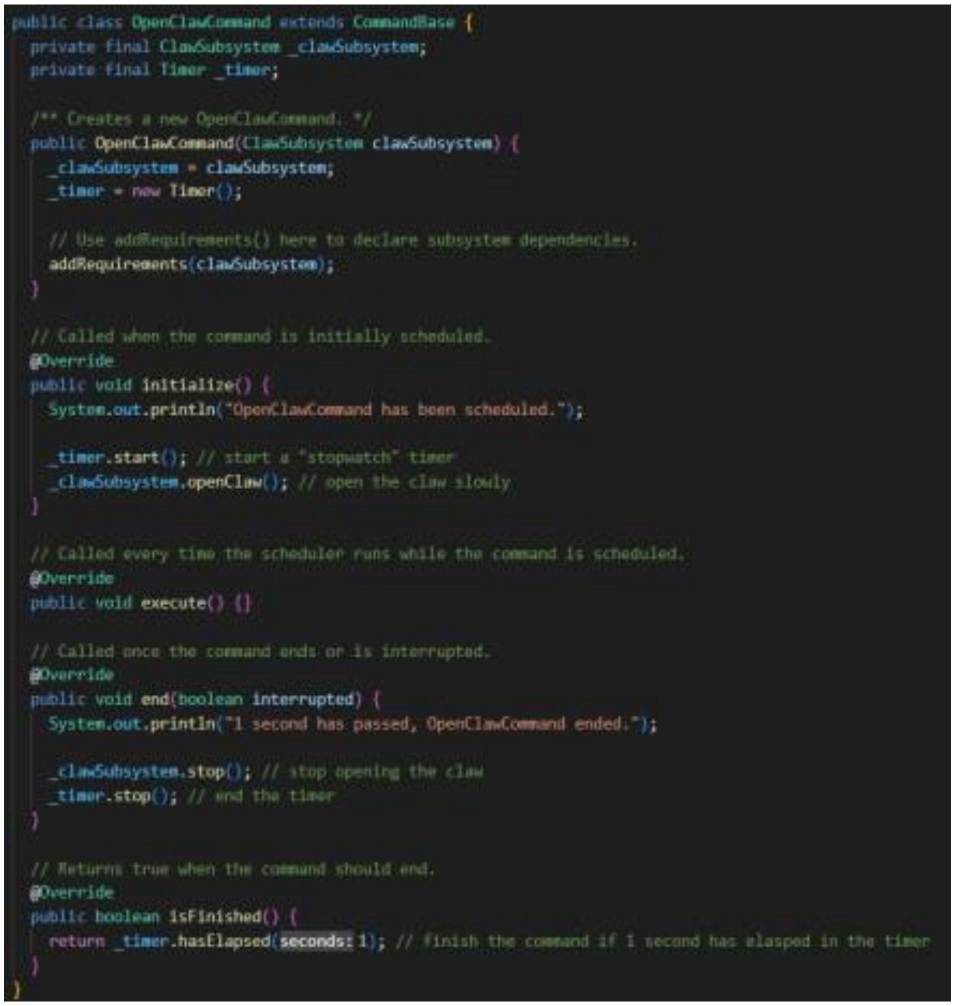
Command example using the claw

The Claw Subsystem test code in the previous image opens the claw, and it is an action performed by the robot, which means it can better be moved into a Command.



```java
/** ⬇ CLAW SUBSYSTEM TEST CODE ⬇ */
// open the claw slowly
_clawSubsystem.openClaw();

// let the claw open for 1 second
Timer.delay(seconds: 1);

// stop opening the claw
_clawSubsystem.stop();
```

OpenClaw Command



```java
public class OpenClawCommand extends CommandBase {
  private final ClawSubsystem _clawSubsystem;
  private final Timer _timer;

  /** Creates a new OpenClawCommand. */
  public OpenClawCommand(ClawSubsystem clawSubsystem) {
    _clawSubsystem = clawSubsystem;
    _timer = new Timer();

    // Use addRequirements() here to declare subsystem dependencies.
    addRequirements(clawSubsystem);
  }

  // Called when the command is initially scheduled.
  @Override
  public void initialize() {
    System.out.println("OpenClawCommand has been scheduled.");

    _timer.start(); // start a "stopwatch" timer
    _clawSubsystem.openClaw(); // open the claw slowly
  }

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

  // Called once the command ends or is interrupted.
  @Override
  public void end(boolean interrupted) {
    System.out.println("1 second has passed, OpenClawCommand ended.");

    _clawSubsystem.stop(); // stop opening the claw
    _timer.stop(); // end the timer
  }

  // Returns true when the command should end.
  @Override
  public boolean isFinished() {
    return _timer.hasElapsed(seconds: 1); // finish the command if 1 second has elasped in the timer
  }
}
```

Brief explanation: The Command does the same thing as the
Subsystem test code, it opens the claw for 1 second. The Command takes in the instanced **ClawSubsystem** as a parameter in the constructor and stores it as a private class attribute. Also, a new wpilib Timer object [https://first.wpi.edu/wpilib/allwpilib/docs/release/java/edu/wpi/firs t/wpilibj/Timer.html](https://first.wpi.edu/wpilib/allwpilib/docs/release/java/edu/wpi/first/wpilibj/Timer.html) is instanced in the constructor and also stored as a private class attribute.

initialize(): The "stopwatch" Timer is started and the claw opens (talonfx in the subsystem is spun).
execute(): (nothing) *end()*: Stop opening the claw and end the Timer, once the Command is ended.
isFinished(): End the Command *if* the Timer has elapsed 1 second.

About *addRequirements()* and interruption

Two or more commands that use the same Subsystem CANNOT be scheduled at the same time (ex: an "OpenClawCommand" and "CloseClawCommand", both requiring the ClawSubsystem, would conflict with each other if they are both scheduled at the same time). If "CloseClawCommand" gets scheduled WHILE "OpenClawCommand" is scheduled, "OpenClawCommand" would be **interrupted** (end() will be called in the command), and "CloseClawCommand" will be scheduled to avoid two Commands using the same Subsystem from running at the same time.

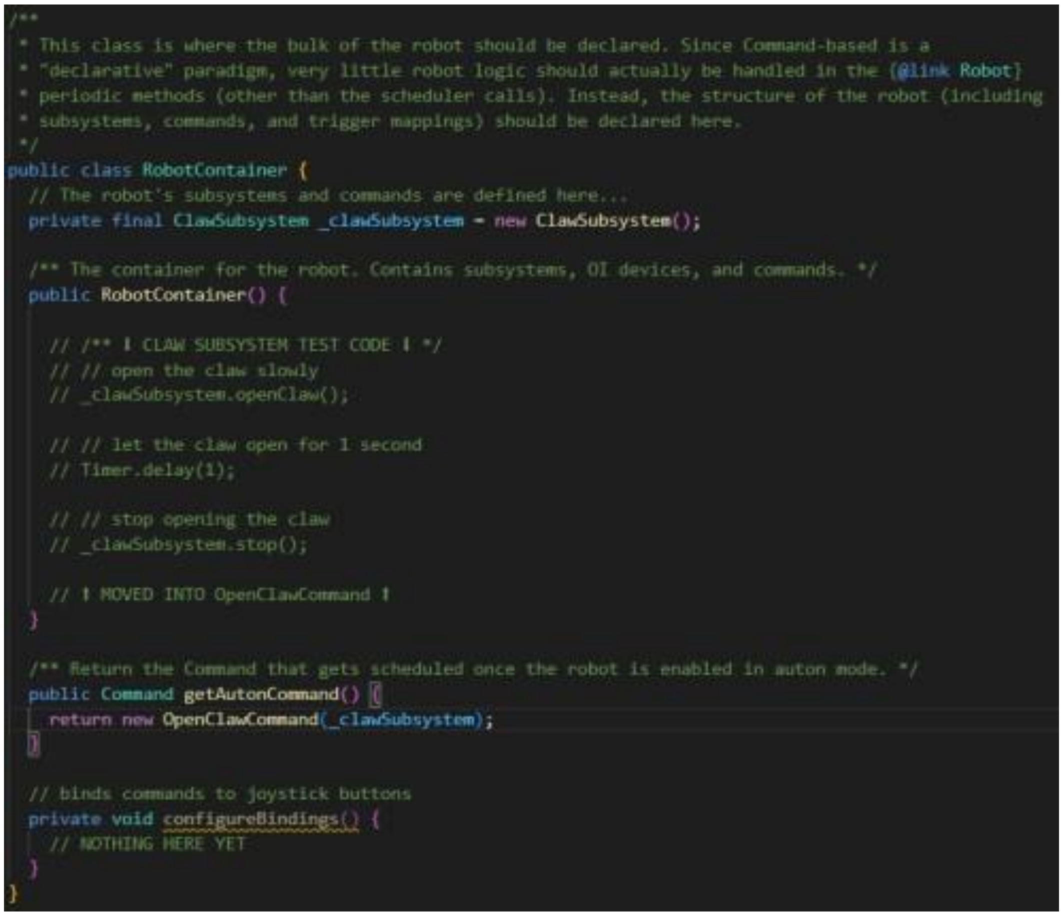
addRequirements(subsystem1, subsystem2, ...): A method part of the CommandBase class that is used to specify the required Subsystems that the Command is using.

boolean **interrupted** in *end()*: The *end()* method takes in a boolean parameter called **interrupted**. If the parameter **interrupted** is **true**, that means that the Command was ended because it was interrupted, and if **false**, the Command was ended because *isFinished()* returned **true**.

Scheduling and using OpenClawCommand

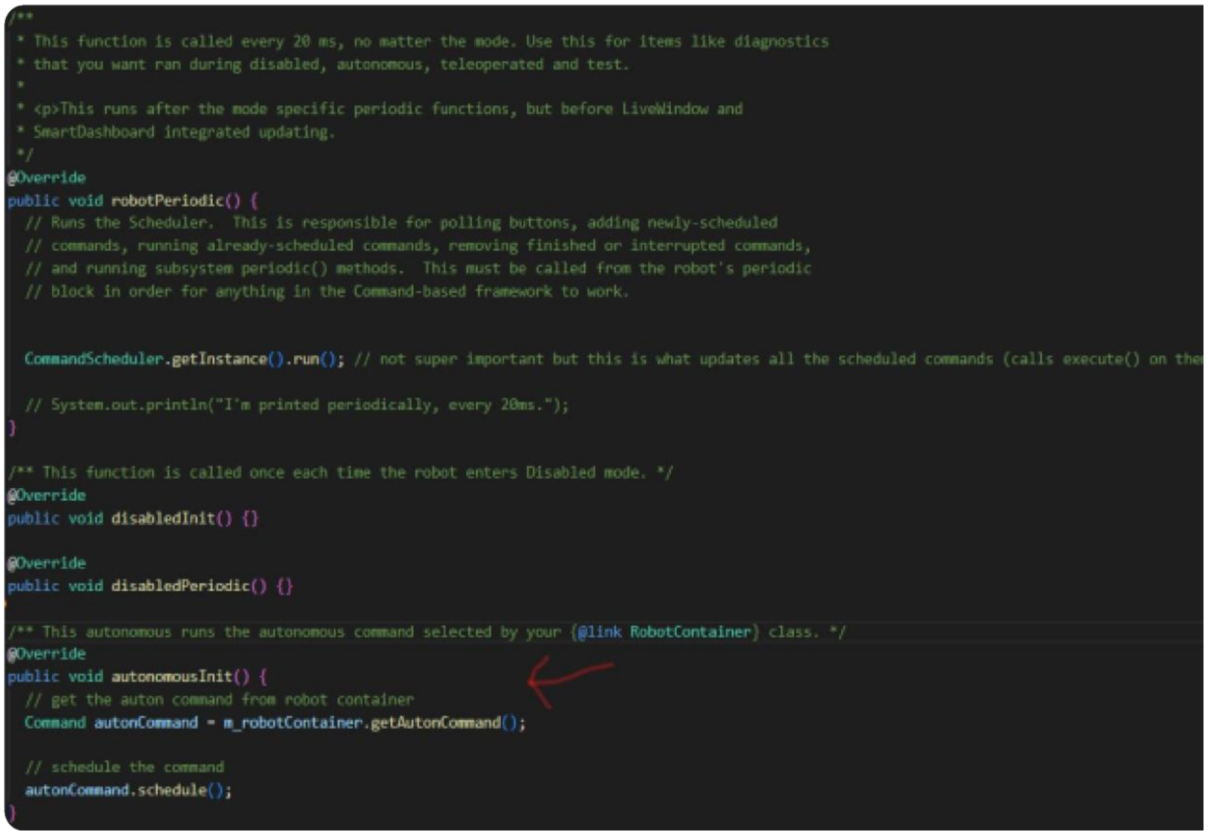
IMPORTANT!: Commands CANNOT be scheduled when the robot is disabled in DS (driver station), and in this example the OpenClawCommand is being scheduled when the robot is enabled in **autonomous** mode.

RobotContainer with OpenClawCommand



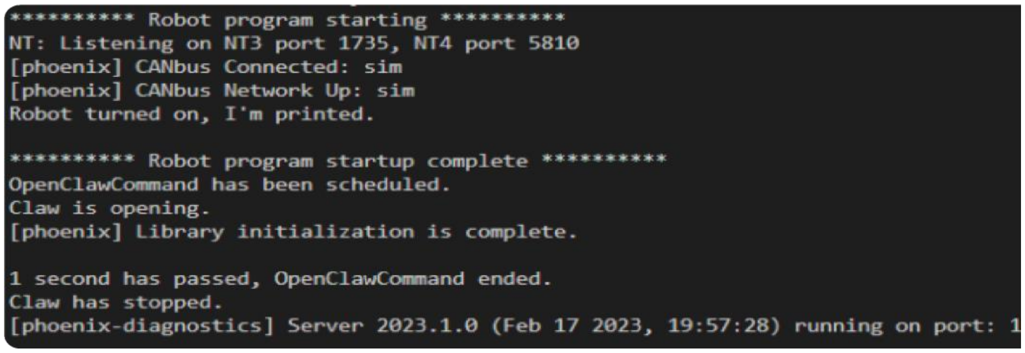
```java
/**
 * This class is where the bulk of the robot should be declared. Since Command-based is a
 * "declarative" paradigm, very little robot logic should actually be handled in the {@link Robot}
 * periodic methods (other than the scheduler calls). Instead, the structure of the robot (including
 * subsystems, commands, and trigger mappings) should be declared here.
 */
public class RobotContainer {
  // The robot's subsystems and commands are defined here...
  private final ClawSubsystem _clawSubsystem = new ClawSubsystem();

  /** The container for the robot. Contains subsystems, OI devices, and commands. */
  public RobotContainer() {

    // /** ⬇ CLAW SUBSYSTEM TEST CODE ⬇ */
    // // open the claw slowly
    // _clawSubsystem.openClaw();

    // // let the claw open for 1 second
    // Timer.delay(1);

    // // stop opening the claw
    // _clawSubsystem.stop();

    // ⬆ MOVED INTO OpenClawCommand ⬆
  }

  /** Return the Command that gets scheduled once the robot is enabled in auton mode. */
  public Command getAutonCommand() {
    return new OpenClawCommand(_clawSubsystem);
  }

  // binds commands to joystick buttons
  private void configureBindings() {
    // NOTHING HERE YET
  }
}
```

Scheduling OpenClawCommand when autonomous mode is enabled through *autonomousInit()* in **Robot.java**



```java
/**
 * This function is called every 20 ms, no matter the mode. Use this for items like diagnostics
 * that you want ran during disabled, autonomous, teleoperated and test.
 *
 * <p>This runs after the mode specific periodic functions, but before LiveWindow and
 * SmartDashboard integrated updating.
 */
@Override
public void robotPeriodic() {
  // Runs the Scheduler.  This is responsible for polling buttons, adding newly-scheduled
  // commands, running already-scheduled commands, removing finished or interrupted commands,
  // and running subsystem periodic() methods.  This must be called from the robot's periodic
  // block in order for anything in the Command-based framework to work.


  CommandScheduler.getInstance().run(); // not super important but this is what updates all the scheduled commands (calls execute() on the

  // System.out.println("I'm printed periodically, every 20ms.");
}

/** This function is called once each time the robot enters Disabled mode. */
@Override
public void disabledInit() {}

@Override
public void disabledPeriodic() {}

/** This autonomous runs the autonomous command selected by your {@link RobotContainer} class. */
@Override
public void autonomousInit() {
  // get the auton command from robot container
  Command autonCommand = m_robotContainer.getAutonCommand();

  // schedule the command
  autonCommand.schedule();
}
```

Console output after deploying the code AND enabling the robot in auton mode (through DS)

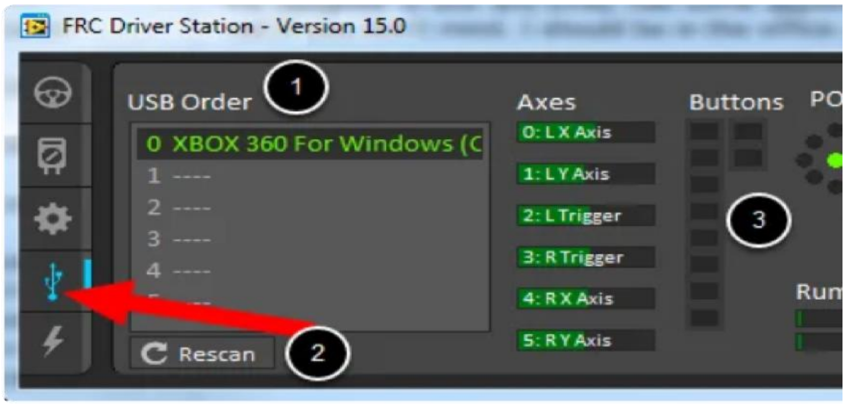


```
********** Robot program starting **********
NT: Listening on NT3 port 1735, NT4 port 5810
[phoenix] CANbus Connected: sim
[phoenix] CANbus Network Up: sim
Robot turned on, I'm printed.

********** Robot program startup complete **********
OpenClawCommand has been scheduled.
Claw is opening.
[phoenix] Library initialization is complete.

1 second has passed, OpenClawCommand ended.
Claw has stopped.
[phoenix-diagnostics] Server 2023.1.0 (Feb 17 2023, 19:57:28) running on port: 1
```

(notice the prints from the ClawSubsystem and OpenClawCommand)
(btw I got rid of the previous "Periodic from ClawSubsystem!" print in ClawSubsystem's periodic())

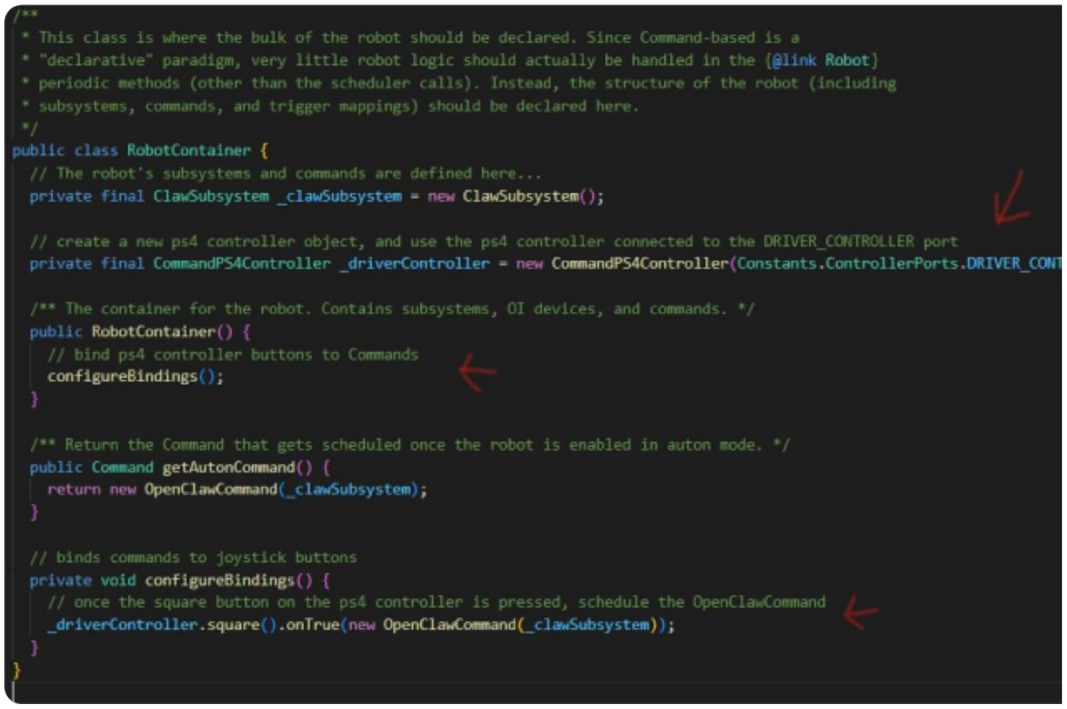
Binding Commands to PS4 Buttons

Apart from **calling.schedule()** on a Command to schedule it, a Command can also be binded to a button on a controller (ps4), and scheduled based on the button's state. Binding Commands to button presses is important for the driver when the robot is enabled in teleop mode.



The USB Devices Tab in DS shows all the controllers connected to the computer. The number on the left of each controller in the image is the **port** that the controller is plugged into (in the image the XBOX 360 controller is in port 0). The port is important because it is needed when accessing the controller in the code.

PS4 Controller in RobotContainer



```java
/**
 * This class is where the bulk of the robot should be declared. Since Command-based is a
 * "declarative" paradigm, very little robot logic should actually be handled in the {@link Robot}
 * periodic methods (other than the scheduler calls). Instead, the structure of the robot (including
 * subsystems, commands, and trigger mappings) should be declared here.
 */
public class RobotContainer {
  // The robot's subsystems and commands are defined here...
  private final ClawSubsystem _clawSubsystem = new ClawSubsystem();

  // create a new ps4 controller object, and use the ps4 controller connected to the DRIVER_CONTROLLER port
  private final CommandPS4Controller _driverController = new CommandPS4Controller(Constants.ControllerPorts.DRIVER_CONT

  /** The container for the robot. Contains subsystems, OI devices, and commands. */
  public RobotContainer() {
    // bind ps4 controller buttons to Commands
    configureBindings();
  }

  /** Return the Command that gets scheduled once the robot is enabled in auton mode. */
  public Command getAutonCommand() {
    return new OpenClawCommand(_clawSubsystem);
  }

  // binds commands to joystick buttons
  private void configureBindings() {
    // once the square button on the ps4 controller is pressed, schedule the OpenClawCommand
    _driverController.square().onTrue(new OpenClawCommand(_clawSubsystem));
  }
}
```

Brief explanation: _*driverController* is a private attribute of the RobotContainer, and its type is *CommandPS4Controller*, a class from wpilib that's used to read ps4 controllers and bind their buttons to commands. The class takes in the **port** of the controller, which is for now an arbitrary integer value in **Constants** (but would usually be found in DS the way it is in the previous image). In *configureBindings()*, OpenClawCommand is binded to the *square* button on the driver controller, such that on the square button's press, the command gets scheduled. *configureBindings()* is called in RobotContainer's constructor.

Class Reference

CommandPS4Controller class reference (methods for reading the joystick values on the controller as well as more buttons to bind to): https://first.wpi.edu/wpilib/allwpilib/docs/release/java/edu/wpi/first/wpilibj2/command/button/CommandPS4Controller.html

Trigger class reference (in addition to onTrue, there is onFalse, whileTrue, whileFalse, and more): https://first.wpi.edu/wpilib/allwpilib/docs/release/java/edu/wpi/first/wpilibj2/command/button/Trigger.html

Default Commands

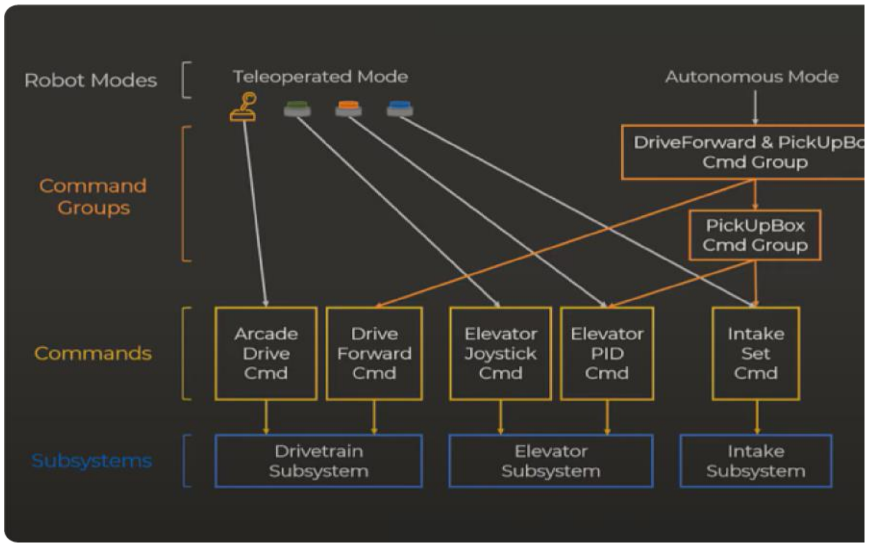
Useful functionality that makes a certain Command run automatically for a Subsystem when no other Command is using that Subsystem.

ex: (inside RobotContainer's constructor)
_armSubsystem.setDefaultCommand(new HoldArmCommand(_armSubsystem));

When no other command that uses the arm Subsystem is scheduled, a default "HoldArmCommand" (holds the arm in place, for instance) is scheduled.

A visual example for the Command-based system.

(didn't go over Command Groups but they're essentially complex commands composed of two or more commands https://docs.wpilib.org/en/stable/docs/software/commandbased/c ommand-compositions.html)


Robot Modes
Teleoperated Mode
Autonomous Mode
Command Groups
DriveForward & PickUpBo
Cmd Group
PickUpBox
Cmd Group
Commands
Arcade Drive Cmd
Drive Forward Cmd
Elevator Joystick Cmd
Elevator PID Cmd
Intake Set Cmd
Subsystems
Drivetrain Subsystem
Elevator Subsystem
Intake Subsystem