How to code XCP Swerve XS

By Daniel Sabalakov, Class of 2026

January 2, 2025

You are here to learn how to code swerve. Is it hard, yes. Is it easy, also, yes. The following guide will lead you through a comprehensive guide of swerve coding, commanding, and subsystem work.

To start, open up your preferred program editor of choice. Your choice may differ based on various choices, but I am using WPILib VScode as my IDE. Then, view diagram 1A on the basics of what the swerve will do.

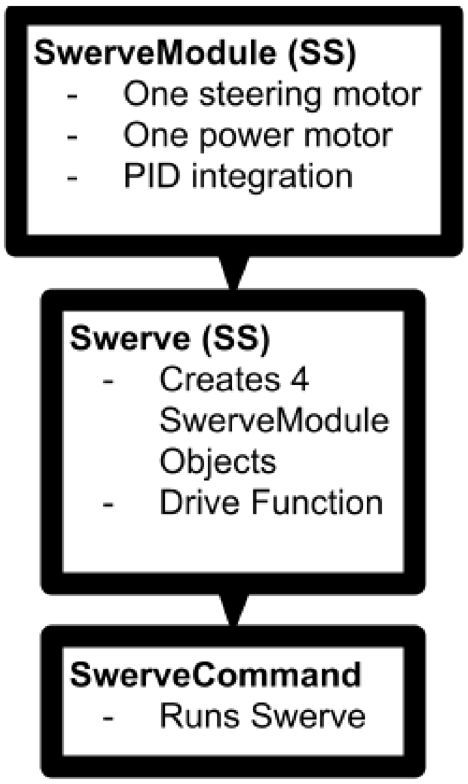

Figure 1: UML Diagram of what is coded

Alas. Time to code the most primitive subsystem: The SwerveModule. At the very top, the following packages are needed.

```java
package frc.robot.subsystems;

import com.ctre.phoenix.motorcontrol.ControlMode;
import com.ctre.phoenix.motorcontrol.FeedbackDevice;
import com.ctre.phoenix.motorcontrol.can.WPI_TalonSRX;
import com.revrobotics.CANSparkMax;
import com.revrobotics.RelativeEncoder;
import com.revrobotics.SparkPIDController;
import com.revrobotics.CANSparkLowLevel.MotorType;

import edu.wpi.first.math.controller.PIDController;
import edu.wpi.first.math.geometry.Rotation2d;
import edu.wpi.first.math.kinematics.SwerveModulePosition;
import edu.wpi.first.math.kinematics.SwerveModuleState;
import edu.wpi.first.wpilibj.Encoder;
import edu.wpi.first.wpilibj2.command.SubsystemBase;
import frc.robot.Constants.SwervePIDConstants;
```

These imports will be used later to create several different objects. Then, after, write the following line:

```java
public class SwerveModule extends SubsystemBase {
```

This line, if using a proper IDE, should generate another "}" at the end, but just hit enter. Word by word, the following keywords mean the following things:

public: The public essentially states that the class can be used by anyone. If it said "Private", then that would mean that it could only be used in certain scope.

class: This is a keyword used to define a class; the very foundation of what Java is based on.

SwerveModule: This is the class name

extends: This essentially takes all the methods of the extended class, and allows you to use and override them.

SubsystemBase: This is what is getting extended. The functions of SubsystemBase, such as periodic(), sometimes is used.

```java
public static final double kWheelRadius = 0.0379; //convert LATER, in m
public static final int kEncoderResolution = 4096;
//CONSTANTS
public static final double kGearReduction = 0.5;
public static final double encoderToRadians = (2*Math.PI)/(kEncoderResolution);
public static final double kMaxSpeed = 2.0;
public static final double kMaxAngularSpeed = Math.PI;

public CANSparkMax powerController;
public WPI_TalonSRX steeringController;
public RelativeEncoder powerEncoder;
public SparkPIDController powerPIDController;
```

This code snippet above initializes some constants that are used above. Furthermore, it initializes our two motor,s CANSparkMax (named powerController), and WPI_TalonSRX (steeringController)

The RelativeEncoder (powerEncoder) and SparkPIDController (powerPIDController) are used in conjunction with the CANSparkMAX motor encoder.

Some key terms to note:

static: reduces usage of space, used and stays constant throughout any instance of the class.

final: means that it is a constant and doesn't change

double: a type of number, includes any number including decimals.

int: a primitive data types, includes only integers

Math.PI: A usage of the built in Math class, calling a constant using the "."

Moving onto the first method, we will complete the initialize method. When the instance of the subsystem is created, the initialize method, named the same as the class, is run. The syntax is as follows: For a class named "TestClass", –: public SwerveModule(){}

Recall, though, that with any method, you can add arguments. So, when we initialize the four swerve modules that make up the swerve subframe, we will need a port (ID) for both the power and the steering motor, as well as a boolean to check if the power should be inverted. (IE: the right sometimes has to be inverted on driveframes.)

```
public SwerveModule(int powerID, int steeringID,boolean powerInvert){
```

Next, hit enter and now we will be developing the powerController.

```
powerController = new CANSparkMax(powerID,MotorType.kBrushless);
powerController.setInverted(powerInvert);
powerEncoder = powerController.getEncoder();
powerPIDController = powerController.getPIDController();
powerEncoder.setPosition(0);
//PID
powerPIDController.setP(0.2);
powerPIDController.setI(0);
powerPIDController.setD(0.002);
//Encoder Factors
powerEncoder.setPositionConversionFactor(
                    2*Math.PI*kWheelRadius/(42*kGearReduction));
powerEncoder.setVelocityConversionFactor(
                    2*Math.PI*kWheelRadius/(42*kGearReduction*60));
```

The powerController is first created (initialized) as a new CANSparkMax instance. The two parameters, the powerID that is inputted above in the initialize statement, and the MotorType are passed in. In a CANSparkMax controller, the swerve modules that we (5902) use are always kBrushless. Then, we set the power direction to what the initialized powerInvert is. Now, the powerEncoder is set to the powerController's encoder value. The powerEncoder is set to 0, and also the powerPIDController (For position control) is initialized. Next, the powerPIDController has some values set to them. These values have been determine by yours truly, and work in this case. For PID tuning, it's too complex to describe in detail in a swerve drive guide, but essentially the proportional part (P) is the main factor, then the integral adds up error over time, and the derivative takes the slope of error to correct it. The powerEncoder now has to set some conversion factors, which allows the output of the encoder to be in correct units.

```
steeringController = new WPI_TalonSRX(steeringID);
steeringController.configFactoryDefault();
//sets the configured sensor to various predetermined constants
steeringController.configSelectedFeedbackSensor(
          FeedbackDevice.CTRE_MagEncoder_Relative,
          SwervePIDConstants.kPIDLoopIdx,
          SwervePIDConstants.kTimeoutMs);
//ensure sensor is positive when output is positive
steeringController.setSensorPhase(SwervePIDConstants.kSensorPhase);
//sets the steeringController to inverted if the motor tells it to do so.
steeringController.setInverted(SwervePIDConstants.kMotorInvert);
//set peak and nominal outputs
steeringController.configNominalOutputForward(0,
          SwervePIDConstants.kTimeoutMs);
steeringController.configNominalOutputReverse(0,
          SwervePIDConstants.kTimeoutMs);
steeringController.configPeakOutputForward(1,
          SwervePIDConstants.kTimeoutMs);
steeringController.configPeakOutputReverse(-1,
          SwervePIDConstants.kTimeoutMs);
//configures position closed loop in slot0
steeringController.config_kF(
          SwervePIDConstants.kPIDLoopIdx,
          SwervePIDConstants.kGains.kF,
          SwervePIDConstants.kTimeoutMs);
steeringController.config_kP(
          SwervePIDConstants.kPIDLoopIdx,
          SwervePIDConstants.kGains.kP,
          SwervePIDConstants.kTimeoutMs);
steeringController.config_kI(
          SwervePIDConstants.kPIDLoopIdx,
          SwervePIDConstants.kGains.kI,
          SwervePIDConstants.kTimeoutMs);
steeringController.config_kD(
          SwervePIDConstants.kPIDLoopIdx,
          SwervePIDConstants.kGains.kD,
          SwervePIDConstants.kTimeoutMs);
steeringController.configSelectedFeedbackCoefficient(
          1/kEncoderResolution,0,0);

//sets the relative sensor to match 0
steeringController.setSelectedSensorPosition(
          0,
          SwervePIDConstants.kPIDLoopIdx,
          SwervePIDConstants.kTimeoutMs);
```

This is a lot of code, eh? However, most of this is quite simple. The steeringController, which is controlled by a WPI_TalonSRX controller, is initialized in the port given. It is reset, (.configFactoryDefault()), then is configured to set specific coefficients of encoder code. Then, it ensures that the sensor is positive when the motor direction is positive. Then, it sets the controller to inverted if it is told to. The max voltages are applied, and then the kF (Again, PIDF tuning), and the rest of the coefficients are tuned. The steering controller is then configured as a feedback coefficient of the reciprocal of motor resolution. Finally, it sets the sensor position to 0 to “reset” it.

Key words used, are mostly: SwervePIDConstants. Rather than a library, it is a set of constants found in a different file: Constants.java. Specifically, the subsection of code responsible for the behavior of constants is below:

```
public static final int kSlotIdx = 0;

public static final int kPIDLoopIdx = 0;

public static final int kTimeoutMs = 0;

public static boolean kSensorPhase = true;

public static boolean kMotorInvert = false;

public static final Gains kGains = new Gains(0.6, 0.0000, 0.0, 0.0, 0, 1.0);
```

The new keywords found here is the class, Gains. Gains is a custom class that acts as a wrapper for PIDFIZone control. The Gains.java code is rather simple: found below. It essentially acts as a storage location of the various different values that different configurations of swerve might have.

```
package frc.robot;

public class Gains {
        public final double kP;
        public final double kI;
        public final double kD;
        public final double kF;
        public final int kIzone;
        public final double kPeakOutput;

        public Gains(double _kP, double _kI, double _kD,
                        double _kF, int _kIzone, double _kPeakOutput){
                kP = _kP;
                kI = _kI;
                kD = _kD;
                kF = _kF;
                kIzone = _kIzone;
                kPeakOutput = _kPeakOutput;
        }
}
```

This is a complete class, which consists of the Class variables and the initialize method that takes in all of the various different values. NOTE that since kP is a public variable, outside the class, a person might change the value of kP by calling it, IE: Gains.kP = 0.3;

Back to the SwerveModule class, the next method, resetSteerPosition, allows you to reset the steering position of the motor.

```
public void resetSteerPosition(){
    steeringController.setSelectedSensorPosition(0,
                SwervePIDConstants.kPIDLoopIdx,
                SwervePIDConstants.kTimeoutMs);
}
```

All it does is copy from the initialize method: it resets the sensor position. Some key words to note is void, which just like in the video Minecraft, is an empty place. Essentially void means that the function isn't giving anything back to you.

The next method, getPosition, is more complicated. It returns SwerveModulePosition, which is a class built by WPILib.

```java
public SwerveModulePosition getPosition(){
    double distanceMeters = powerEncoder.getPosition();
    double angleRadians =
                steeringController.getSelectedSensorPosition() * encoderToRadians;
    return new SwerveModulePosition(
        distanceMeters,
        new Rotation2d(angleRadians)
    );
}
```

Here, there are many things to note. The distance in meters is received by the powerEncoder, and the angle in Radians is received by multiplying the sensor position by how many encoder ticks equal a radian. It then returns a new SwerveModulePosition, where the constructor used is: SwerveModulePosition(double, Rotation2d).

SetDesiredState() sets the desired state of the steering and the drive controllers. By holding the constructor as: setDesiredState(desiredState), (and also returning void btw), the following code is run:

```java
public void setDesiredState(SwerveModuleState desiredState){
    double currentAngleRadians =
                steeringController.getSelectedSensorPosition() * encoderToRadians;
    Rotation2d currentRotation = new Rotation2d(currentAngleRadians);
    SwerveModuleState optimizedState =
                SwerveModuleState.optimize(desiredState,currentRotation);
    double angleError = optimizedState.angle.minus(currentRotation).getRadians();
    double scaledSpeed =
                optimizedState.speedMetersPerSecond * Math.cos(angleError);
        // Clamp between -1 and 1
    double driveOutput =
                Math.max(-1, Math.min(1, scaledSpeed / kMaxSpeed));

    powerController.set(driveOutput);

    double targetAngleRadians = optimizedState.angle.getRadians();
    double targetEncoderPosition = targetAngleRadians / encoderToRadians;
    steeringController.set(ControlMode.Position, targetEncoderPosition);
}
```

The program first determines the currentAngle, then converts it to a Rotation2d object. Then, the optimized state is calculated using a built in class by WPILib. The angleError is figured out, and then the scaled speed that the motor should run at is determined by multiplying the speed in meters a second times the cosine of the angle of error. The drive output is then calculated, (and clamped between -1 and 1), by using Math.max(), a built in library in the Math class. The power controller is then set to that value determined. Target angle (in radians) is then calculated. The steering controller is then told, "hey, go to this position". The ControlMode.position is used for this, because using a different control mode (such as velocity or percent output), could skew with the PID tuning.

The final method, getState(), is a very short stub used to give the state of the swerve module.

```
public SwerveModuleState getState(){
    double speedMPS = powerEncoder.getVelocity();
    double angleRadians =
                steeringController.getSelectedSensorPosition() * encoderToRadians;
    return new SwerveModuleState(speedMPS,new Rotation2d(angleRadians));
}
```

The code gets the speed in meters per second, then the angle, then returns as an instance of a SwerveModuleState.

That was ALL of the SwerveSystem.java subsystem (as of 1/2/2025 at 10:00 PM)

Now, it is time for the next step of the UML diagram: the Swerve subsystem. The Swerve subsystem handles all 4 corners of the Swerve system, and also contains the important drive() method.

Up top, the following imports are done:

```
package frc.robot.subsystems;

import edu.wpi.first.math.geometry.Pose2d;
import edu.wpi.first.math.geometry.Rotation2d;
import edu.wpi.first.math.geometry.Translation2d;
import edu.wpi.first.math.kinematics.ChassisSpeeds;
import edu.wpi.first.math.kinematics.SwerveDriveKinematics;
import edu.wpi.first.math.kinematics.SwerveDriveOdometry;
import edu.wpi.first.math.kinematics.SwerveModulePosition;
import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;
import edu.wpi.first.wpilibj2.command.SubsystemBase;
import frc.robot.Constants;
import frc.robot.Constants.SwerveCANConstants;
```

Then, as with the SwerveModule.java subsystem, the class is declared.

```
public class Swerve extends SubsystemBase{
```

Before the initialize method, the following objects are created

```
public static final double kMaxSpeed = 3.0;
public static final double kMaxAngularSpeed = 0.5*Math.PI;

public Gyro myGyro = new Gyro();

public Translation2d m_frontLeftLocation =
        new Translation2d(-0.573,0.573);
public Translation2d m_frontRightLocation =
        new Translation2d(-0.573,-0.573);
public Translation2d m_backLeftLocation =
        new Translation2d(0.573,0.573);
public Translation2d m_backRightLocation =
        new Translation2d(0.573,-0.573);

public SwerveModule m_frontLeft =
        new SwerveModule(
                SwerveCANConstants.kFrontLeftDrivingCanId,
                SwerveCANConstants.kFrontLeftTurningCanId,
                false);
public SwerveModule m_frontRight =
        new SwerveModule(
                SwerveCANConstants.kFrontRightDrivingCanId,
                SwerveCANConstants.kFrontRightTurningCanId,
                true);
public SwerveModule m_backLeft =
        new SwerveModule(
                SwerveCANConstants.kRearLeftDrivingCanId,
                SwerveCANConstants.kRearLeftTurningCanId,
                false);
public SwerveModule m_backRight =
        new SwerveModule(
                SwerveCANConstants.kRearRightDrivingCanId,
                SwerveCANConstants.kRearRightTurningCanId,
                true);
public SwerveDriveKinematics m_kinematics
                = new SwerveDriveKinematics(
            m_frontLeftLocation,
            m_frontRightLocation,
            m_backLeftLocation,
            m_backRightLocation
        );
```

Wow, that was a lot! However, once you simplify the code, it isn't as daunting as it seems. First, the max speeds are re-done, (Even though it isn't necessarily necessary.) The Gyroscope is a seperate class, that I will talk about later. Translation2d essentially states the distance between the center of robot and the swerve module. Positive x values represent moving toward the front of the robot while positive y values represent moving toward the left of the robot. Then, the four SwerveModules (from SwerveModule.java) are initialized, with SwerveCANConstants (which I also will discuss later). Then, the kinematics is created using the locations of the swerve modules as inputs: since the input is SwerveDriveKinematics(Translation2d, Translation2d, Translation2d, Translation2d).

This is a good segment to the gyroscope. The gyroscope code hasn't been 100% figured out (as of 10:14 PM, 1/2/2025), but the code for the class is below.

```
package frc.robot.subsystems;

import com.kauailabs.navx.frc.AHRS;

import edu.wpi.first.wpilibj2.command.SubsystemBase;
import edu.wpi.first.math.geometry.Rotation2d;
import edu.wpi.first.wpilibj.ADXRS450_Gyro;
import edu.wpi.first.wpilibj.AnalogGyro;
import edu.wpi.first.wpilibj.I2C;
import edu.wpi.first.wpilibj.SPI;
import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;

public class Gyro extends SubsystemBase{
    public ADXRS450_Gyro m_gyro;
    public double originalPosition;

    public Gyro(){
        m_gyro = new ADXRS450_Gyro();
        reset();
    }
    public Rotation2d getAng(){
        return m_gyro.getRotation2d();
    }
    public void reset(){
        // m_gyro.zeroYaw();
        m_gyro.reset();
    }

}
```

The class initializes a gyroscope using the proper port, then resets it. To get the angle, it gets the Rotation2d. The gyroscope is used for field oriented drive (which also hasn't been quite figured out yet but I am working on it). The CANConstants also mentioned are below:

```
public static final class SwerveCANConstants{
    public static final int kFrontLeftDrivingCanId = 5;
    public static final int kRearLeftDrivingCanId = 7;
    public static final int kFrontRightDrivingCanId = 6;
    public static final int kRearRightDrivingCanId = 8;

    public static final int kFrontLeftTurningCanId = 1;
    public static final int kRearLeftTurningCanId = 3;
    public static final int kFrontRightTurningCanId = 2;
    public static final int kRearRightTurningCanId = 4;
}
```

which are essentially ID's used and found in Phoenix Tuner X.

Back to the Swerve.java, in the initialize method, all that happens is the Gyroscope is reset.

```
public Swerve(){
    myGyro.reset();
}
```

Then, the big method in this class is the drive method.

```
public void drive(
                double xSpeed,
                double ySpeed,
                double rot,
                boolean fieldRelative,
                double periodSeconds){
    ChassisSpeeds chassisSpeed =
                fieldRelative ? ChassisSpeeds.fromFieldRelativeSpeeds(
                        xSpeed,
                        ySpeed,
                        rot,
                        myGyro.getAng())
                       :
                        new ChassisSpeeds(
                                xSpeed,
                                ySpeed,
                                rot);
    var swerveModuleStates = m_kinematics.toSwerveModuleStates(chassisSpeed);
    SwerveDriveKinematics.desaturateWheelSpeeds(
        swerveModuleStates,kMaxSpeed
    );
    m_frontLeft.setDesiredState(swerveModuleStates[0]);
    m_frontRight.setDesiredState(swerveModuleStates[1]);
    m_backLeft.setDesiredState(swerveModuleStates[2]);
    m_backRight.setDesiredState(swerveModuleStates[3]);
    SmartDashboard.putNumber(
                "FL Distance: ",
                m_frontLeft.steeringController.getSelectedSensorPosition(0));
    SmartDashboard.putNumber(
                "FL Speed: ",
                 m_frontLeft.steeringController.getSelectedSensorVelocity(0));
}
```

The drive method first takes in the xSpeed, ySpeed, rotation, and a boolean if it is field relative (and also period seconds). The chassisSpeed is then updated in some if-then short hand notation. (= If value then this else that) (=a?b:c). If the field is relative, then it will calculate how much the ChassisSpeeds should go based on the gyroscope rotation. Else, it will calculate based on just x, y, and rotational speed. The moduleStates is created as an array from the kinematics chassisSpeed state. The kinematics then desaturates (clamps between possible values of speed), and the program sets all four modules to move at the calculated speeds. SmartDashboard is an output class, so we put the number with key "FL Distance" and number of the position and velocity.

resetAll() is the final method, and it just resets all the steer positions.

```
public void resetAll(){
    m_frontLeft.resetSteerPosition();
    m_frontRight.resetSteerPosition();
    m_backLeft.resetSteerPosition();
    m_backRight.resetSteerPosition();
}
```

Now, lets move on to the final part of this guide: The SwerveCommand. Since I tried coding my own library for calculating chassis Speeds, I called the class "BetterSwerveCommand". However, the naming is up to you.

We only need 3 imports:

```
package frc.robot.commands;

import edu.wpi.first.wpilibj.Joystick;
import edu.wpi.first.wpilibj2.command.Command;
import frc.robot.subsystems.Swerve;
```

Then, the class declaration, instead of extending the SubsystemBase, extends Command.

```
public class BetterSwerveCommand extends Command {
```

The variables initialized contain some important values that are used throughout the command:

```
public Joystick myJoystick;
public Swerve mySwerve;
private static final double kDeadband = 0.2;
public double daDirection = 1;
private static final double kZedDeadBand = 0.4;
private static final double kMaxSpeed = Swerve.kMaxSpeed;
private static final double kMaxAngularSpeed = Swerve.kMaxAngularSpeed;
double xSpeed, ySpeed, rot;
```

Some important values to pull are: the deadband is for a deadzone on a joystick, Joystick is the physical joystick and is called "myJoystick", on the Z axis, the deadband is 0.4, the max speeds are again here, and the xSpeed, ySpeed, and rot are the values passed in.

10:28:40 PM Cuba Standard Time

```
public BetterSwerveCommand(Joystick js, Swerve m_s){
    myJoystick = js;
    mySwerve = m_s;
    addRequirements(mySwerve);
}
```

The BetterSwerveCommand has inputs of a Joystick js and Swerve subsystem m_s.

Then, it maps to the variables that were initialized above, and adds the requirement of having mySwerve existing.

```
@Override
public void execute() {


    boolean fieldRelative = false;

    if (myJoystick.getRawButton(2)){
        xSpeed = -daDirection*applyDeadband(myJoystick.getY()) * kMaxSpeed*0.7;
        ySpeed = daDirection*applyDeadband(myJoystick.getX()) * kMaxSpeed*0.7;
        rot = -applyZedDeadband(myJoystick.getZ()) * kMaxAngularSpeed*0.2;
    }
    else {
        xSpeed = -daDirection*applyDeadband(myJoystick.getY()) * kMaxSpeed*0.3;
        ySpeed = daDirection*applyDeadband(myJoystick.getX()) * kMaxSpeed*0.3;
        rot = -applyZedDeadband(myJoystick.getZ()) * kMaxAngularSpeed*0.8;
    }
    if (myJoystick.getRawButton(1)){
        daDirection = -1;
    }
    else {
        daDirection = 1;
    }
    if (myJoystick.getRawButton(3)){
        mySwerve.resetAll();
    }
    mySwerve.drive(xSpeed,ySpeed,rot,fieldRelative,0.02);

}
```

Above, the first line @Override allows to override the classes base. Since Command has an execute() method, the @Overrides writes over what it says. FieldRelative is set to false, and then the following statements are put. If the joystick's number 2 button is pressed, it activates the "turbo", which allows the robot to move way faster. Otherwise, the robot runs at a slower 30% of max speed. Then, the raw button 1 allows the robot to change directions. If the raw button 3 is pressed, it will reset all of the motor locations. Then, mySwerve is forced to drive the robot at the proper speeds.

The final 2 methods are used as deadbands (to avoid stick drift). They are inherently the same and don't require much explanation:

```
private double applyDeadband(double value) {
        return Math.abs(value) > kDeadband ? value : 0.0;
}
private double applyZedDeadband(double value){
    if (Math.abs(value) > kZedDeadBand){
        value = (Math.abs(value) - kZedDeadBand)*value/Math.abs(value);
        return value;
    }
    else {
        return 0;
    }
}
```

You thought you were done? Nope, you have to work on the RobotContainer (the area where commands and subsystems are initialized.)

Where the Subsystems are initialized, initialize the swerve by doing this:

```
public final Swerve m_swerve = new Swerve();
public final Joystick joystick1 = new Joystick(0);
public final BetterSwerveCommand swerveCommand =
            new BetterSwerveCommand(joystick1 , m_swerve);
```

This now creates the 2 subsystems needed for swerve, as well as the Joystick at port 0.

The second to last step is to add the default command to RobotContainer() (the initialize method).

```


   m_swerve.setDefaultCommand(swerveCommand);
```

Finally, you must add the Joystick to return the joystick properly. The code is here:

```
public Joystick getJoystick(){
      return joystick1;
}
```

That is it! Have a great day, whoever is reading this, and the source code is found at this link:

`https://github.com/PHSWireClippers5902/2025-Preseason/blob/main/src/main/java/frc/robot/subsystems/Swerve.java`