# WPILib Java Robot Programming Cookbook with Special Recipes for Team 2399

## JavaDocs for the Commands and Subsystems

If you open the FRC JavaDoc, then click on edu.wpi.first.wpilibj.command (it's the sixth one under WPILibJ), you can see the JavaDoc for the base classes Subsystem and Comand, which are what you are basing your subsystems and commands on. You can use any of the methods that are already defined there, or you can overload them (overwrite them with your own version).

## Subsystems

A subsystem is any functional part of the robot. On PIMP, these subsystems are:

* Drivetrain
* Lift
* Grabber

### What goes in a subsystem

Subsystems contain the parts that make the subsystem work:

* Motors
* Sensors
* RobotDrive (for the drivetrain)
* Useful values (preset height positions for a lift, for example)
* Basic functions

### Subsystems vs. commands

The last one is a bit tricky--what goes in a subsystem, and what goes in a command? The first answer is that a command should never be directly controlling a motor or other actuator. Commands should ONLY be calling the public methods of the subsystems. Since commands can only use the public methods of the subsystem, then the very basic commands have to be public methods of the subsystem class. These are things like setting speed for regular subsystems, setting position for PID subsystems, and turning motors on and off for subsystems that don't need speed control.

So then what goes in a command? The first clue is anything that takes time to execute. Any subsystem method should do one thing that takes almost no time--turn a motor on or off for instance, and then be done. If what you want to do involves waiting for a time or waiting for a sensor to trip, then it should be in a command. Any time you're reading information from the controls (joysticks or buttons), that's a command (see the Teleoperated Mode section for more). Since the default behavior of a subsystem has to be a command, you'll need to define any default behaviors as commands (like driving the arm with the joystick when no presets are pushed).

### PID Subsystems

In a normal subsystem, you have sensors and motors and you tell your motors what speed to go. In a PID subsystem, you don't tell your motors what speed to go, you tell them what position to go to. In order to do this, the robot needs a motor to move the device and a sensor to know where it is. An example of a good use of a PID subsystem is the grabber on PIMP. It has a motor to rotate it and a magnetic encoder to tell the robot what angle it's at. A PID controller tells the motor what speed to go at based on what angle the grabber is at and what angle you want the grabber to go to.

### PID Terminology

* Control effort - How hard to make the system's actuator work. In our case, this is usually the speed we command a Jaguar.
* Error - The difference between the system's current position and your setpoint. Error = setpoint - position.
* PID control - Proportional, Integral, Derivate control. Uses the error between your system's position and setpoint to determine the control effort.
* Position - Where the system is now, according to the sensor(s). You can also lie about this to make your PID controller do interesting things.
* Setpoint - Where you want the system to go.

PID subsystems are pretty similar to regular subsystems, but there are a couple more things you need to do:

* In the constructor:
  + super(kp, ki, kd); calls the constructor for the parent PIDSubsystem class and sets the PID constants.
  + setSetpointRange(UPPER\_BOUND, LOWER\_BOUND); sets the limits that you don't want to let the system go past (for example the maximum and minimum height that the lift can safely go on PIMP)
  + Set the default setpoint with setSetpoint();
  + enable(); enables the PID controller
* public boolean atSetpoint() - You'll have to overload this function, which tells commands that use the system if it's currently at the setpoint. It should return true if it's at the setpoint, or false if it's not.
* protected void usePIDOutput(double output) - You'll have to overload this function, which tells the subsystem what to do with the PID controller's output. Usually you'll be sending this to a Jaguar to control a motor.
* protected double returnPIDInput() - You'll have to overload this function. The output of this function is the input to the PID controller. In the simplest case, this would read a sensor and return its value. In some cases, you might want to do some math on the sensor value (to convert units or something) before you send it to the PID controller. In some cases, you might want to lie to the PID controller to force it to do something. On PIMP's lift, I lied to the PID controller when the lift was going to the bottom position so that even if the encoder reads 0, it doesn't stop going down until it hits the limit switch. I did this in case the encoder isn't properly reset to 0 (like if we start the robot with the lift already up).

A couple more things about PID subsystems:

* You can use them to control speed instead of position. That's a way to make something spin at a very precise rate. You just need a sensor that measures speed instead of position.
* You can disable them if you want to override the PID controller and manually set motor speed, but it's probably a better idea to adapt your code to use the PID controller. For example, to manually move the lift on PIMP, you can use the joysticks to move the setpoint around.

### Steps for Making a SubSystem

1. Make a new SubSystem class. You can do this by copying the ExampleSubSystem class and changing the name of the class and of the constructor.
2. Create any properties and write any methods for the subsystem class.
3. Add the class to the CommandBase class by importing it and adding it as a property of the CommandBase class. This is IMPORTANT. If you don't do it, the rest of the code won't know the subsystem exists.

## Commands

Commands are anything you want the robot to do. Anything a button does is a command. Any step in the autonomous mode is a command. Even something like driving with the joysticks is a command.

### Commands we would want PIMP to execute:

* Drive with joysticks
* Move arm with joystick
* Grab Red Tube
* Grab Blue Tube
* Grab White Tube
* Drop Tube
* Move arm to position (we can specify the height as an argument)
* Follow line

### Command Groups

Some commands are made up of a list of commands. These commands can executed in order (serial) or at the same time (parallel).

For example, last year during autonomous, we wanted to raise the arm while we drove forward on a line. When we reached the end of the line AND the arm was at the right height, we dropped the tube. We could accomplish this with a command group that runs FollowLine and ArmTo commands in parallel, then a DropTube command after they both complete.

## The RobotMap

The RobotMap class is sort of like a configuration file for the robot. We put all the port numbers for the sensors, motors, and stuff here so that if electrical changes things around on us, we only need to change the number in one place.

## Teleoperated Mode

Unlike the old system, you don't explicitly write out what to do on every update of the teleoperated mode. Instead, the command "scheduler" takes care of it for you. Any subsystem with a default command (such as driving with joysticks) will just do its default command until you tell it otherwise.

The only line in the teleopPeriodic() function of the robot is:

Scheduler.getInstance().run();

If you want to assign something to a button on a joystick or on PEEF, you do this in the OI class. There are several types of buttons. Here are the three we're likely to use:

* JoystickButton (a button on a Joystick)
* DigitalIOButton (a button going into one of the digital inputs on the FIRSTTouchIO board in PEEF. These are PEEF's buttons 1-8)
* AnalogIOButton (a button going into one of the analog inputs on the FIRSTTouchIO board in PEEF. PEEF button 9 is an analog button because we ran out of digital)

Once you create your buttons in the OI class, all three types behave the same way. You can set their behavior in the OI constructor with one of three methods:

* button.whenPressed(new *CommandName*) - Do the command CommandName when you press the button
* button.whenPressed(new *CommandName*) - Do the command CommandName when you press the button
* button.whenPressed(new *CommandName*) - Do the command CommandName when you press the button

A single button could even have multiple behaviors (like a button that grabs a tube when you press it and drops it when you release it).

## Autonomous Mode

Autonomous mode is usually just a list of commands. That sounds a lot like a CommandGroup, so that's exactly how you do autonomous mode. Define a Command or CommandGroup that you want to run during autonomous. In the main robot class, create an instance of that command group. In autonmousInit() (which runs at the start of autonomous mode), start the Command (or CommandGroup) with the command's start() method.

## Tip - JavaDoc Your Code

You can add JavaDocs to your code by putting a description before every class and function inside of /\*\* \*\*/ comments (note the double \*). Then, in netbeans, right click the project (in the list on the left side of the screen) and click Generate Javadoc. NetBeans will generate a JavaDoc website and open it for you.