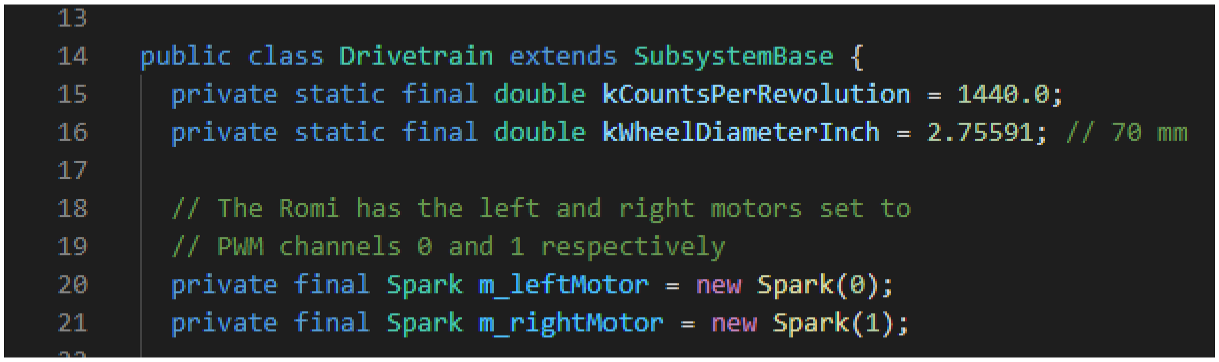
In Lesson 2 you learned what methods are, but we only scratched the surface. In this lesson we'll introduce two more critical concepts – *classes* and *objects*. We'll learn how to utilize the buttons on your joystick and make the on-board lights on the Romi turn on and off. Start off by creating a new RomiReference project called "Lights".

Classes & Objects

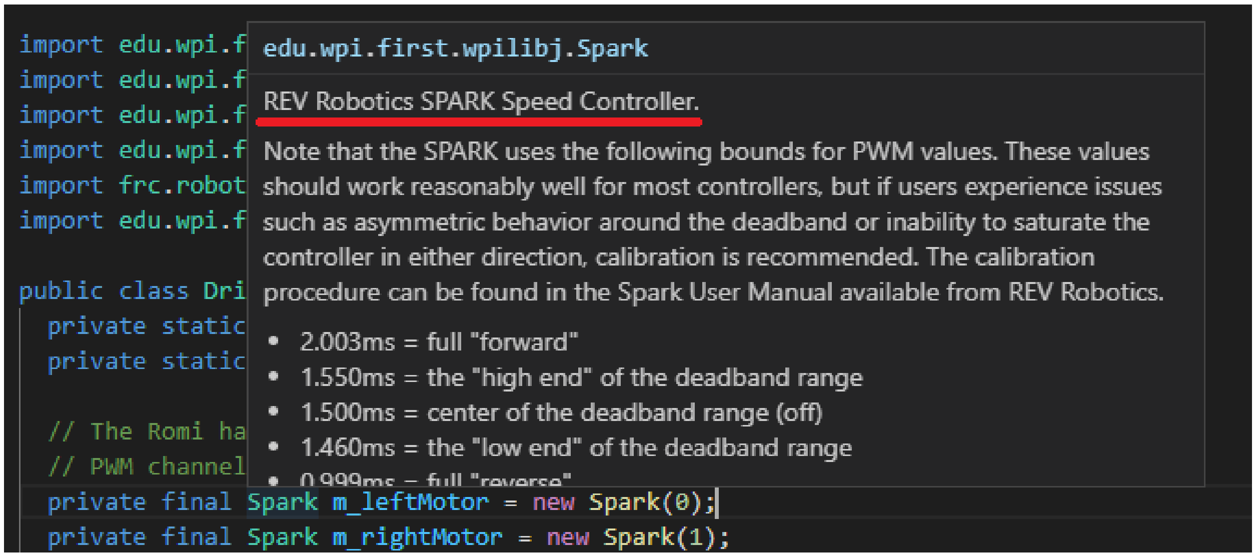
Java is what is known as an "object-oriented" programming language, or OOP. The basic concept behind OOP is that there are *classes*, which represent real-world idea or concepts, that can be described in code. For example, you could have a "Car" class, and in code, you could describe a car, by doing things like keeping track of its model, make, max speed, current speed, etc. Structuring code like this makes it easy to think about because your code relates to your real-life concept of what you're working with. An *object* is an *instance* of a class. For example, you could have a Car class, and then individual objects that represent individual cars. In Java, each class is typically represented by one file, and each file represents one class. This is why you in the explorer you see filenames that correspond to concepts and ideas that make sense in English. For example, there is "Robot.java" which represents a robot, "Drivetrain.java" that represents a drivetrain, and so forth. We'll do a deep dive on classes and objects soon, but for today we'll keep moving quickly and get you writing code before going too in-depth. As you read through the next few paragraphs, think about the big picture and don't worry too much about remembering each specific keyword, because you'll remember them better after we go in more depth, and you can refer back to this lesson and re-read the following paragraphs as needed until then.

You can create classes by defining them in files. For example, the Drivetrain.java file defines the Drivetrain class as mentioned above. If you open Drivetrain.java and look at line 14, you'll see the text "public class Drivetrain" which, declares the Drivetrain class. Then you'll see several lines that look like normal variable declarations "double kWheelDiameterInch = 2.75591;", but with extra words in front. Don't worry about those extra words for now. These are variables that belong to the Drivetrain class and help create a logical model of a drivetrain. Classes can be composed of variables that are relevant to the class, which can then later be accessed when using the class.



```
13
14  public class Drivetrain extends SubsystemBase {
15    private static final double kCountsPerRevolution = 1440.0;
16    private static final double kWheelDiameterInch = 2.75591; // 70 mm
17
18    // The Romi has the left and right motors set to
19    // PWM channels 0 and 1 respectively
20    private final Spark m_leftMotor = new Spark(0);
21    private final Spark m_rightMotor = new Spark(1);
```

If you look at lines 20 and 21, you'll see two more variables declared – m_left and m_rightMotor. This makes sense – thinking about the Romi, it has two motors, one on each side. However, these variables are unlike variables you've seen before. Instead of being of type int or double, they're of type *Spark*. What is a Spark? Well, turns out you can hover your mouse over the word Spark to get a brief description. There's a lot of technical information in the pop-up that you can ignore for now, but the key part is the first sentence: "REV Robotics SPARK Speed Controller." A Spark represents a real-world object called a speed controller, which physically handles outputting voltage to motors to make them spin. In this case it's called a SPARK and manufactured by the company REV Robotics.



```
import edu.wpi.f
import edu.wpi.f
import edu.wpi.f
import edu.wpi.f
import frc.robot
import edu.wpi.f

public class Dri
  private static
  private static

  // The Romi ha
  // PWM channel
  private final Spark m_leftMotor = new Spark(0);
  private final Spark m_rightMotor = new Spark(1);
```

edu.wpi.first.wpilibj.Spark

REV Robotics SPARK Speed Controller.

Note that the SPARK uses the following bounds for PWM values. These values should work reasonably well for most controllers, but if users experience issues such as asymmetric behavior around the deadband or inability to saturate the controller in either direction, calibration is recommended. The calibration procedure can be found in the Spark User Manual available from REV Robotics.

- 2.003ms = full "forward"
- 1.550ms = the "high end" of the deadband range
- 1.500ms = center of the deadband range (off)
- 1.460ms = the "low end" of the deadband range
- 0.999ms = full "reverse"

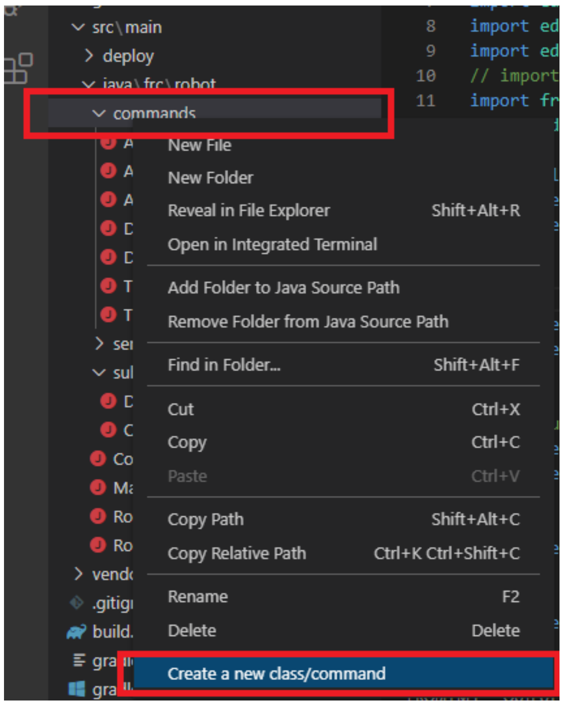
The Java programming language is used worldwide for billions of devices and programs, and certainly cannot natively support every piece of hardware ever manufactured, like a SPARK motor controller. So how can we declare a variable of type Spark? The answer is, once a class is created in Java, you can declare variables of that type. So, somebody made a class called Spark, just like we have a class called Drivetrain. Once the Spark class exists, we can create Spark variables. These variables are different than more "basic" variables like ints and doubles. Ints and doubles are called *primitive* variables because they are inherent to Java. There are a few other primitive variable types, for example *boolean* variables which represent either "true" or "false". But variables of non-primitive types, or in other words variables that are of the type of a class, are called *objects*. Once a class exists, objects of that class can be created. An object of a class is called an *instance* of that class. So, referencing the Drivetrain class we're looking at, on line 14 the class is declared. On lines 15 and 16, two primitive variables are declared of type double, and on lines 20 and 21 two objects are defined of type Spark. The Spark class is defined somewhere else, but we're not worried about there here. Someone else did the work of making the Spark class work, and now you get to use it. Notice how when variables that are objects are created, they use an additional keyword "new", and then repeat the object name – for example on line 20 when declaring the m_leftMotor spark object, it says "new Spark(0)". Again, we'll talk about this more later on but for now, when you see the "new" keyword, that's your clue that it's an object being created instead of a primitive. Another clue is that for objects the variable type is capitalized – e.g. Spark instead of spark, but not for primitive variables (e.g. int instead of Int.)

That's a lot of information to take in so let's do a quick recap. In Java there are *class*es, which can be defined, and after doing so they can be used by creating an *instance* of the class, which is called and *object*. Classes can be given variables which describe the class, for example describing a drivetrain by declaring that it has a left and a right motor. These variables can then be used when working with the class. Let's show how that happens.

You already saw how on lines 20 and 21, two Spark objects are created. On line 29, these objects are used to create a "DifferentialDrive" object. DifferentialDrive is another class that has been created for us that has some built-in functionality that handles the math that converts joystick values to motor outputs. For this class to work it needs motors to output to, which is why you can see at the end of line 29 that the two Spark objects are being used to create the DifferentialDrive object, called m_diffDrive. Scroll down and on line 46 you'll see m_diffDrive.arcadeDrive(xaxisSpeed, zaxisRotate), which you're familiar with from lesson 2. This is an example of using an object to do something – in this case, the m_diffDrive object, and we're using the "arcadeDrive" method that has been declared as part of the DifferentialDrive class. This is another critical concept when it comes to classes and objects – methods can be defined as part of the behavior for the class. Then those methods can be called using objects of that class.

We'll go in more depth soon but now let's dive into our project of using buttons on the joystick to turn a light on and off. To do this there are a couple steps. First, we're going to have to create two *commands*, one to turn the light on, and one to turn it off. Commands are another concept we'll talk about soon but for now you can think of a command as "something the robot can do". For example, you could make a command to drive forward, a command to stop, a command to toggle a light, etc. In this case we'll make two; one each for turning the light on and off. Let's walk through it step by step. After the commands are created, you'll need to add a couple lines of code that tell the commands to run when a button on the joystick is pressed. Let's get started.

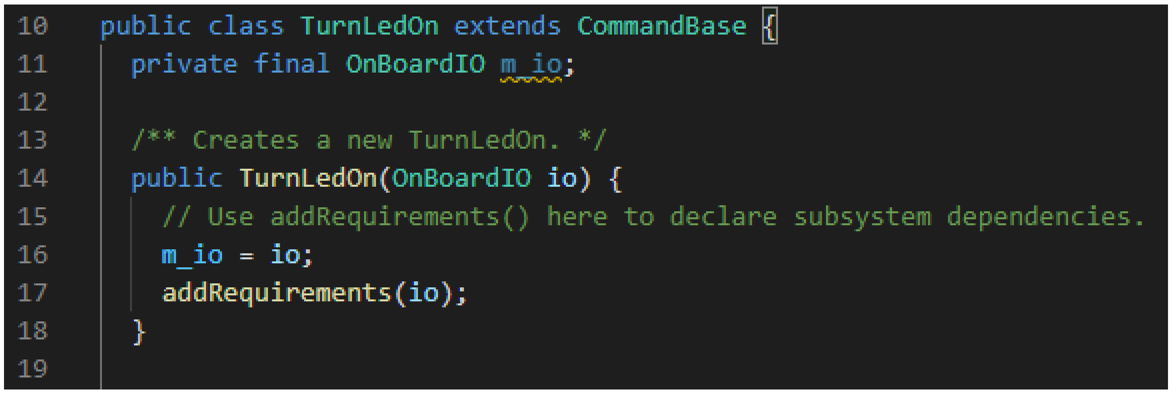
The first thing is to create the commands. Let's start with the command to turn the light on. Navigate to the "commands" folder in src/main/java/frc/robot. Right click on the folder itself in VSCode and select "Create a new class/command" at the bottom of the popup.


src\main
deploy
java\frc\robot
commands
New File
New Folder
Reveal in File Explorer Shift+Alt+R
Open in Integrated Terminal
Add Folder to Java Source Path
Remove Folder from Java Source Path
Find in Folder... Shift+Alt+F
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Copy Path Shift+Alt+C
Copy Relative Path Ctrl+K Ctrl+Shift+C
Rename F2
Delete Delete
Create a new class/command

```
import ed
import ed
// import
import fr
```

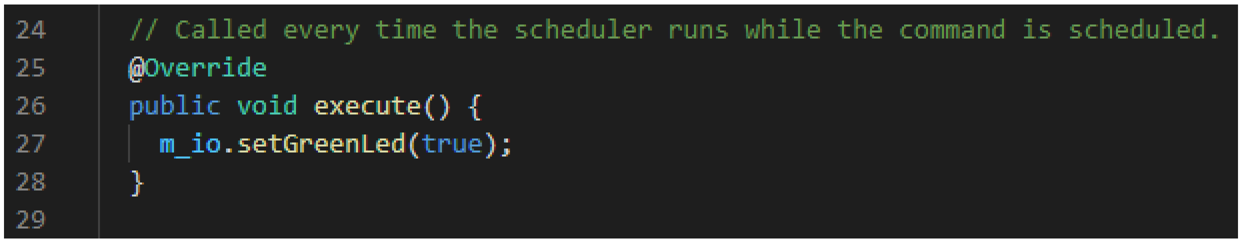
This will bring up a dropdown with a list of options. Select "Command (New)". This brings up a prompt to type a class name. Type "TurnLedOn" and press enter. This brings up a new file with some method *stubs* – a stub is a method without any contents. Lines 17, 21, 25, and 29 are examples of this. Now follow these steps to create this command. Note that you need to follow the steps in order, or the line numbers won't sync up, because the line numbers assume that you have entered the lines described in the previous steps.

1. Move your cursor to the very end of line 7, after the semicolon, press enter, and then type "import frc.robot.subsystems.OnBoardIO;". This line tells the program that it will need to use the robot's on-board IO (input/output) subsystem. This is the subsystem that has the lights we're going to turn on. Note how the line ends with a semicolon. Most lines of code in Java end with semicolons; this is how you tell Java that the line is finished. The exceptions are lines that end with opening and closing curly brackets, for example lines 10, 14, and 18 in the image in step 3. You'll learn more about the difference between these kinds of lines soon.
2. Put your cursor at the end of line 10 and press enter, and then type "private final OnBoardIO m_io;", then press enter. Again – don't forget the semicolon. In step 1 we told our program that it will need to use an OnBoardIO subsystem, and on this line we declare that OnBoardIO subsystem for it to use.
3. Put your cursor *inside* the parentheses on line 14 (NOT the curly bracket at the end) and type "OnBoardIO io". Then go to the end of line 15 (the next line), press enter, and add two lines of code. These lines initialize the OnBoardIO object.
    a. m_io = io;
    b. addRequirements(io);
    c. When you're done with these steps, lines 10-19 should look like this:



```
10  public class TurnLedOn extends CommandBase {
11    private final OnBoardIO m_io;
12
13    /** Creates a new TurnLedOn. */
14    public TurnLedOn(OnBoardIO io) {
15      // Use addRequirements() here to declare subsystem dependencies.
16      m_io = io;
17      addRequirements(io);
18    }
19
```

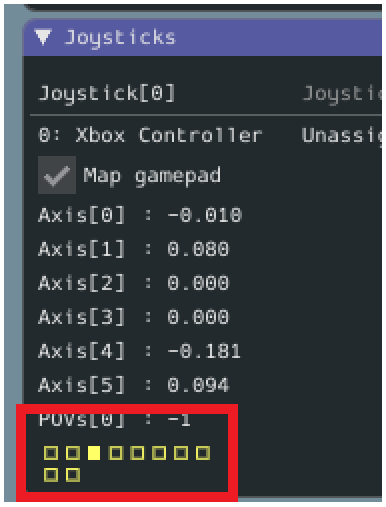
4. Move your cursor to inside the curly brackets on line 26 (this time NOT the parentheses.) Press enter and then add a line of code by typing "m_io.setGreenLed(true);". Your execute method should now look like this:



```
24    // Called every time the scheduler runs while the command is scheduled.
25    @Override
26    public void execute() {
27      m_io.setGreenLed(true);
28    }
29
```

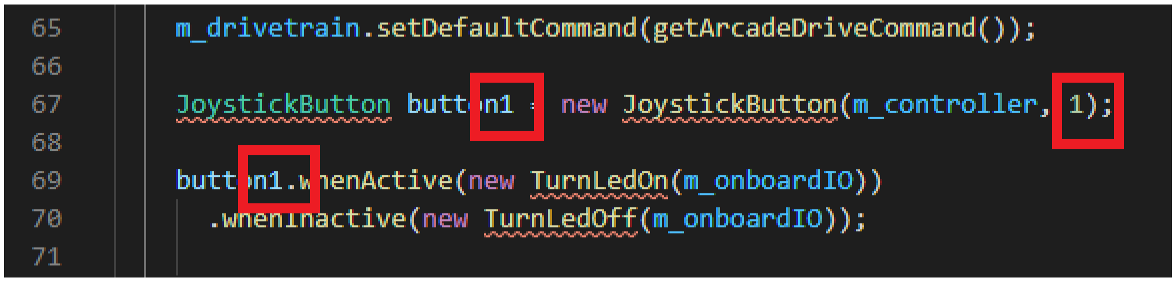
Congratulations! You've now created your first robot command. Unfortunately, there's no way to activate this command yet, but we'll add that in a moment. First, let's create the sister command that turns the LED off. Follow the steps above again to create another command in the same folder, but call it "TurnLedOff". Follow all the same steps to add the code to the new command, but when you get to line 27 where you do m_io.setGreenLed(true);, use "false" instead of "true". Once you've created the second command, it's time to hook them up to a joystick button.

The first step in hooking them up to a joystick button is figuring out which joystick button is which. This is a similar process to how you figured out which joystick axis was which in lesson one. Open the robot simulator interface if you don't already have it open (you can press F5 to deploy your code right now if you're connected to the Romi) and look at the joysticks window. You might need to re-drag your joystick from the Systems Joystick window, to the Joysticks window. Press whatever button on your controller you want to control the light turning on and off, and look at the yellow squares in the joystick window. One of them will light up. Count which number it is, from upper left, starting at one. For example, in this image, button 3 is lit up. Make note of which button lights up as you will need to remember the number in your code.



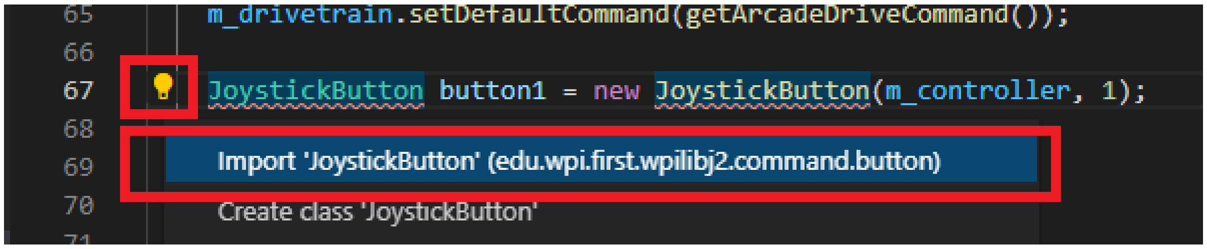
```
▼ Joysticks

Joystick[0]               Joysti
0: Xbox Controller        Unassi
✓ Map gamepad
Axis[0] : -0.010
Axis[1] : 0.080
Axis[2] : 0.000
Axis[3] : 0.000
Axis[4] : -0.181
Axis[5] : 0.094
POVs[0] : -1
☐☐■☐☐☐☐☐
☐☐
```

Now go to RobotContainer.java, put your cursor at the end of line 65, press enter twice, and type out code to match the image below. However, where there are red boxes, change the number 1 to whatever number your button is. These lines of code create a "JoystickButton" object and bind the commands you created to execute when you press and release the button, respectively. You'll see the red underlines indicating you have errors, and we'll fix those next. Note that it's possible that VSCode will fix the red underline under "JoystickButton" on its own. If it does so, that red underline will go away, and all the line numbers in your code will increase by one relative to the image below.



```
65    m_drivetrain.setDefaultCommand(getArcadeDriveCommand());
66
67    JoystickButton button1 = new JoystickButton(m_controller, 1);
68
69    button1.whenActive(new TurnLedOn(m_onboardIO))
70      .whenInactive(new TurnLedOff(m_onboardIO));
71
```

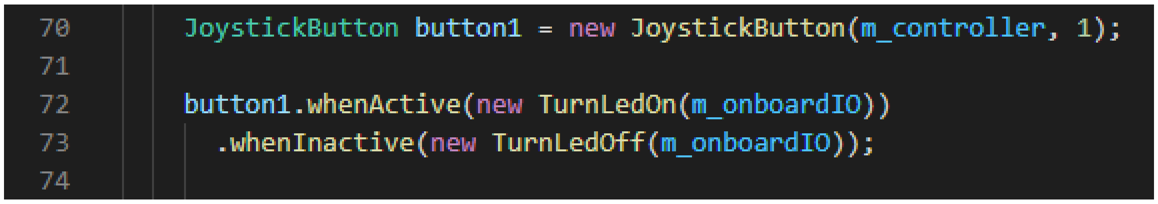
If VSCode does not fix the red underline under JoystickButton automatically, you'll need to do so yourself. To do so, click on the green text JoystickButton, and you'll see a little yellow lightbulb pop up to the left. VSCode has limited capabilities to fix common problems for you, and you can access this functionality by clicking on yellow lightbulbs when they appear. Do so and then click "Import 'JoystickButton'. This will add a line of code to the top of your program, telling the RobotContainer class to look for the JoystickButton class, which you use when you declare your button1 variable on line 67 above (now 68 after it inserts a line of code.) You will have the same errors on your TurnLedOn and Off commands, because RobotContainer needs to look for those files as well. Use the same method to import those two files as well, and your errors will go away. As alluded to above, sometimes VSCode will fix these errors on its own without you needing to make these clicks, but often you'll have to do this.



```
65      m_drivetrain.setDefaultCommand(getArcadeDriveCommand());
66
67      JoystickButton button1 = new JoystickButton(m_controller, 1);
68
69
70
```

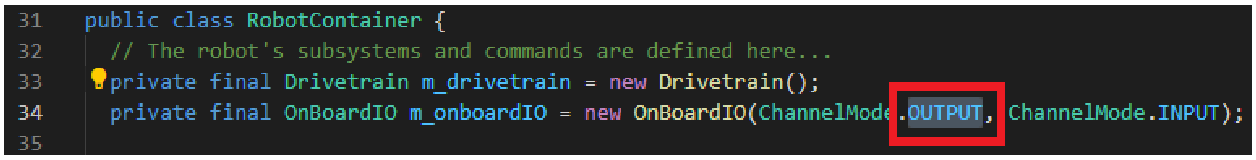
Import 'JoystickButton' (edu.wpi.first.wpilibj2.command.button)
Create class 'JoystickButton'

When you're done importing the classes using the lightbulb, your code should be error-free and look like the photo below:



```
70      JoystickButton button1 = new JoystickButton(m_controller, 1);
71
72      button1.whenActive(new TurnLedOn(m_onboardIO))
73        .whenInactive(new TurnLedOff(m_onboardIO));
74
```

You're now almost done. The last thing to do is to tell the Romi to use the green LED as an output device. The way the Romi is physically constructed, its green and red LEDs use "digital input/output", or *DIO*, ports, which can be used for either input or output. Using an LED is an example of output – the Romi will *output* the light. An example of input would be if you plugged a button into the DIO port, which would let the Romi receive input. We will not cover that in this course. By default, when you create a RomiReference project, both the red and green LED DIO ports are set to receive input, instead of output, so you'll need to change that. Head to RobotContainer.java, go to line 34, and change the first instance of the word "INPUT" to "OUTPUT". The second "INPUT" refers to the red LED, which you're not using in this lesson. Example:



```
31  public class RobotContainer {
32    // The robot's subsystems and commands are defined here...
33    private final Drivetrain m_drivetrain = new Drivetrain();
34    private final OnBoardIO m_onboardIO = new OnBoardIO(ChannelMode.OUTPUT, ChannelMode.INPUT);
35
```

Now you're done! You can deploy your code to your Romi, enable it, and try pressing your button. You will notice that a light near the back of the Romi lights up while you're holding the button down, and turns off when you release it.

Congratulations on finishing the first three lessons. You've now learned a lot of the basics of Java and robot programming. There's still a lot to learn but if you've made it this far, you're doing well and you should have a basic understanding of how software can be used to effect the behavior of the robot. Over the next lessons we'll build on that understanding, which will increase both your skills and your confidence in writing your own programs.