In this lesson you'll learn a critical new concept in programming known as *conditionals*, or *if statements*. You use if statements to create conditional logic in your code, for example "if something is true, do something, otherwise do something else". You'll then do a project where you use your new knowledge to give yourself some additional control over your robot so you can perform tricky driving maneuvers more easily.

If Statements

If statements in Java are pretty simple to code, and they tend to make a lot of sense logically as well. The basic form looks like this:

```
1   public class App {
        Run | Debug
2       public static void main(String[] args) throws Exception {
3           int hourOfDay = 6;
4
5           if (hourOfDay < 10) {
6               System.out.println("Good morning!");
7           }
8       }
9   }
10
```

This short program declares an int called hour of day and sets its value to 6, and then checks whether that value is less than 10. If it is, it outputs a line of text. Unsurprisingly, if you run this program, it does, indeed, output "Good morning!". If you were to change the value on line three from 6 to 12, it would output nothing.

There are two additional keywords that go along with if statements: *else if*, and *else*. Else if allows you to specify a second condition, that will be checked *if and only if* the first if statement is false. Here is an example:

```
1   public class App {
        Run | Debug
2       public static void main(String[] args) throws Exception {
3           int hourOfDay = 6;
4
5           if (hourOfDay < 10) {
6               System.out.println("Good morning!");
7           }
8           else if (hourOfDay < 15) {
9               System.out.println("Good day!");
10          }
11      }
12  }
13
```

If you run this program, the output will again be “Good morning!”. Even though 6 is indeed less than 15, line 8 never runs because line 5 is true. If you change the value of hourOfDay to 11, then line 8 will be false, line 8 will run and evaluate to true, and the output will be “Good day!”. Lastly, there is “else”. An else clause will run if and only if all the prior clauses evaluate to false. Here is an example:

```
1   public class App {
        Run | Debug
2       public static void main(String[] args) throws Exception {
3           int hourOfDay = 20;
4
5           if (hourOfDay < 10) {
6               System.out.println("Good morning!");
7           }
8           else if (hourOfDay < 15) {
9               System.out.println("Good day!");
10          }
11          else if (hourOfDay < 18) {
12              System.out.println("Good afternoon!");
13          }
14          else {
15              System.out.println("Good evening!");
16          }
17      }
18  }
19
```

In this case, since hourOfDay is 20, the first three if and else if statements all evaluate to false, so the else statement runs and the output is “Good evening!”.

Operators

You may have noticed in the prior examples that we used the “<”, or “less than” sign, in our if statements. This is called an “operator” and it is used to compare two values. There are a few other operators, which you may be familiar with from math class:

- <: less than
- >: greater than
- <=: less than or equal to
- >= greater than or equal to
- ==: equal to

Notice how instead of just using one equal sign, the equal to operator uses *two* equal signs. This is because if you just use *one* equal sign, you would be assigning a value to a variable. When you write if statements you need to be careful to use both equal signs, because if you use just one, two bad things will happen. First, instead of checking the value of the variable, you will overwrite its old value with whatever you’re checking against. E.g. “if (hourOfDay = 10)” will overwrite hourOfDay with the value 10. Additionally, *the operation of overwriting the value of hourOfDay with 10* will be successful, and so the operation will return true. So not only will your variable be overwritten, but your if statement will return

true because the overwrite was successful. This will cause errors later in your program, so be careful to always use the double equals sign when checking equality. Here's an example:

```
1   public class App {
        Run | Debug
2       public static void main(String[] args) throws Exception {
3           int hourOfDay = 10;
4
5           if (hourOfDay == 10) {
6               System.out.println("It's 10 AM!");
7           }
8           else {
9               System.out.println("It's not 10 AM!");
10          }
11      }
12  }
13
```

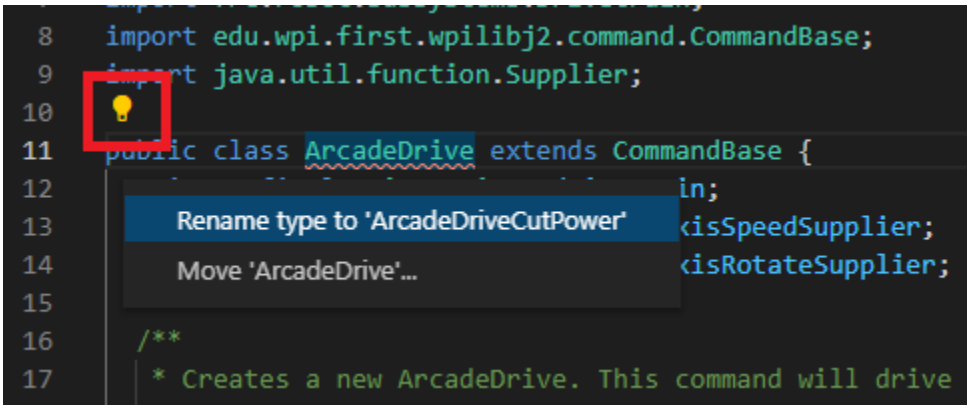
Quick Practice

Try a quick demo for yourself using a non-Romi program, like you did in the last lesson, except instead of hourOfDay, use an int variable called "dayOfWeek". A value of 1 represents Sunday, 2 Monday, etc. through all seven days. Then write an if statement and a series of else if statements that check the value until it finds what day of the week it is, and outputs a line of text saying which day of the week it is. Include an else statement that runs if the day of the week is not a sensible value, or in other words if it's any value that isn't between 1 and 7 inclusive. Then run your program and make sure it works correctly. Try it with a few different values and make sure that your code outputs the correct day, and make sure if you give it a value of, say, 12 that your else statement runs correctly.

Romi Project – Cut Power Mode

In this project we'll add a driving mode called "cut power" and bind it to a button on your controller. While the button is held, all power outputs to the Romi's drivetrain will be reduced, but when the button is not being held everything is normal, letting you switch between drive modes instantly on the fly. This can be useful when you want extra maneuverability while driving. To start, create a new RomiReference project.

Create a new command called "ArcadeDriveCutPower" in the commands folder. There's already a command called ArcadeDrive, and we could modify that one instead, but for this project we'll just create a parallel command so we don't need to update other references to the original command. After you create the new command, open the original ArcadeDrive command and copy its entire text and paste it into your new command (overwrite the default text in the new command.) After you do this you'll get a red squiggly line under "ArcadeDrive" on line 11 because the name of the class doesn't match the filename. Click on the underlined test and a lightbulb appears. Click the lightbulb and choose the first option ("Rename…") to let VSCode fix the class name for you, based on the filename of your command.



Method Arguments

Now that you're more familiar with methods and passing values into them, it's time to flex your knowledge by adding a parameter to your new command. You'll need an additional class variable first. We'll talk about these next lesson but for now navigate to the end of line 14, press enter, and type "private final Supplier<Boolean> m_cutPowerModeSupplier;". Now you can add a parameter to the method on line 26 called "ArcadeDriveCutPower". Notice that this method actually spans a few lines with line breaks after each comma. Position your cursor immediately after "zaxisRotateSupplier" on line 29, add a comma and press enter, and add a parameter of type "Supplier<Boolean>" called "cutPowerModeSupplier". The "Supplier<Boolean>" data type is an advanced concept that we won't cover in this course, but you can essentially think of it like a boolean variable. In this case that boolean will represent true/false for whether cut power mode is activated. Now that we've declared the parameter, it just needs to be initialized. Position your cursor at the end of line 33, press enter, and set your "m_cutPowerModeSupplier" variable equal to the method parameter you just created. Here's what your class should look like so far:

```
11  public class ArcadeDriveCutPower extends CommandBase {
12    private final Drivetrain m_drivetrain;
13    private final Supplier<Double> m_xaxisSpeedSupplier;
14    private final Supplier<Double> m_zaxisRotateSupplier;
15    private final Supplier<Boolean> m_cutPowerModeSupplier;
16
17    /**
18     * Creates a new ArcadeDrive. This command will drive your robot according to
19     * the speed supplier lambdas. This command does not terminate.
20     *
21     * @param drivetrain         The drivetrain subsystem on which this command
22     *                           will run
23     * @param xaxisSpeedSupplier  Lambda supplier of forward/backward speed
24     * @param zaxisRotateSupplier Lambda supplier of rotational speed
25     */
26    public ArcadeDriveCutPower(
27        Drivetrain drivetrain,
28        Supplier<Double> xaxisSpeedSupplier,
29        Supplier<Double> zaxisRotateSupplier,
30        Supplier<Boolean> cutPowerModeSupplier) {
31      m_drivetrain = drivetrain;
32      m_xaxisSpeedSupplier = xaxisSpeedSupplier;
33      m_zaxisRotateSupplier = zaxisRotateSupplier;
34      m_cutPowerModeSupplier = cutPowerModeSupplier;
35      addRequirements(drivetrain);
36    }
```

The last thing you need to do in this command is change one more line in the execute() method, on line 45. Make two changes to this line. First, make a new method, again to keep references to the original method intact. On line 45, where m_drivetrain.arcadeDrive is called, change “arcadeDrive” to “arcadeDriveCutPower”. Lastly, add a parameter to the method call so your execute() methods looks like this:

```
42    // Called every time the scheduler runs while the command is scheduled.
43    @Override
44    public void execute() {
45      m_drivetrain.arcadeDriveCutPower(m_xaxisSpeedSupplier.get(), m_zaxisRotateSupplier.get(), m_cutPowerModeSupplier.get());
46    }
```

You will have a red underline on your method call because the method arcadeDriveCutPower doesn’t exist yet. So far you’ve done all the “plumbing” to make sure that the boolean for whether or not the robot is in cut power mode gets to where it needs to go. The next step will be creating this method that doesn’t exist yet, which will utilize the boolean to actually apply cut power to the robot. Make sure to save your code in this file before proceeding.

The arcadeDriveCutPower Method

Navigate to Drivetrain.java and find the arcadeDrive method on line 45. This is the method that is being called right now when you drive your robot, as you might remember from prior lessons when you modified the drive code. This time you’re going to copy it and modify it using an if statement. You can start by copying the entire method, adding a blank line beneath it, and then pasting the code in. Then change the name of the method to match your method call from line 45 of ArcadeDriveCutPower.java above. Lastly, add a boolean parameter to the new method, to again match the parameter added in ArcadeDriveCutPower.java. Save your code and navigate back to ArcadeDriveCutPower.java. You’ll notice that line 45 will no longer have a red underline, because now the method you’re calling exists. Of course, since you simply copied the existing arcadeDrive method, your code won’t do anything different yet. Let’s modify your copied method to make it work.

Back in Drivetrain.java in your method, add a new line at the start of the method and then add an if statement. Your method has a parameter “boolean cutPowerMode”. You want to check if this variable is true or not. If it is you’ll want to scale the values of both xaxisSpeed and zaxisRotate, similar to how you did in the lesson about variables where you modified the drivetrain code. The exact amount to scale them by to make your cut power mode feel good will depend on what physical surface your Romi is driving on and how much friction there is. You can adjust the values after trying it out, but a decent starting point is to scale them both by .7. Creating this if statement and scaling the values is all you need to do here.

In the prior lesson where we played with these variables, we changed their values in a very standard fashion:

```
xaxisSpeed = xaxisSpeed * .7;
zaxisRotate = zaxisRotate * .7;
```

There’s another way to update variables that’s slightly shorter and is nice to use when the update is pretty simple and understandable, such as multiplying by a static coefficient. You can use the “times-equals” operator, like so:

```
xaxisSpeed *= .7;
zaxisRotate *= .7;
```

The lines of code in these two images have exactly the same function – they will both update the variables by multiplying them by .7. The second way is simply a shortcut you can use if you want. We'll use it from time to time in this tutorial. Whether you use it or not is up to you, but you need to recognize it and understand it. It's also worth noting that you can do the same with other operators, for example +=, -=, etc.

When you're done, your method should look something like this:

```
49  public void arcadeDriveCutPower(double xaxisSpeed, double zaxisRotate, boolean cutPowerMode) {
50    if (cutPowerMode == true) {
51      xaxisSpeed *= .7;
52      zaxisRotate *= .7;
53    }
54
55    m_diffDrive.arcadeDrive(xaxisSpeed, zaxisRotate);
56  }
```

Updating the Default Command

Your new command and method are both ready to go, so the last thing you need to do is update the command that your drivetrain is using, to your new command. Navigate to RobotContainer.java and scroll to line 65. You'll see a method call for "setDefaultCommand". This line sets the default command for the drivetrain subsystem. You can set default commands for any subsystem, of which the Romi has two - the drivetrain, and the on-board IO, which controls the lights. Default commands run all the time as long as there is no other command issued to the subsystem. Using the drivetrain as an example, this is generally the case. In teleop, you just want to drive, so there's generally no reason to issue other commands. There are exceptions to this, for example you could create a command to turn exactly 90 degrees when you press a button. If that command were issued to the drivetrain, the default drive command would stop running until the turn command finishes, and then once the turn command finishes, the default command would turn on again. Other situations where you might not want default commands to run would be during autonomous, where you want to issue specific commands. We'll do all of these things in future lessons, but for now, we're just interested in using the default command so we can drive, and we want that default command to be the new command we created.

Looking at line 65 again, you see the setDefaultCommand method, and there's actually *another* method call inside of it – "getArcadeDriveCommand()". In VSCode, you can drill-in to any method by clicking on the call for that method and pressing F12. Do this on the getArcadeDriveCommand call and VSCode will navigate you to line 93. Move down one line to 94 and update the "ArcadeDrive" command to "ArcadeDriveCutPower". You'll also need to add an additional parameter, since you added a parameter in your command. In order to do this you'll need to figure out which button you want to use to activate cut power mode. You can do this the same way you figured out which button to use to light up the LED, so do that. If you don't have the simulation interface open and can't deploy code because you have errors, you can simply use button 1 for now, and then change it later. Either way, decide on a number for your button, and add a parameter to your method by adding a comma after the "m_controller.getRawAxis(4)", and then typing the following text: "() -> m_controller.getRawButton(6)", where 6 is replaced by the number of your button. If you're using an Xbox controller, 6 will be the right bumper. The reason this line of code looks so messy has to do with the Suppliers that we mentioned earlier won't be covered in this course. Don't worry too much about this – there are ways to do all of this without Suppliers, so you aren't missing out on critical information. We'll cover this in the future but

we're using them for now because that's what the example code uses. When you're done making the changes, your method should look something like this:

```
93    public Command getArcadeDriveCommand() {
94      return new ArcadeDriveCutPower(
95          m_drivetrain, () -> -m_controller.getRawAxis(1), () -> m_controller.getRawAxis(4), () -> m_controller.getRawButton(6));
96    }
```

The last thing you'll need to do is import your new command, which you can do by clicking on the underlined command name, using the lightbulb (VSCode may do this automatically for you.) While you're there you can remove the unused import for the previous ArcadeDrive command if you want; if you don't do this, you'll have a yellow underline pointing out that the import is not used, but it doesn't affect your program. After you do this you can deploy your code and try out cut power mode! If you still need to, you may want to update which button controls cut power mode. Either way, try driving your Romi, and see how you can hold and release the cut power button to change its characteristics. If you want, you can go back to Drivetrain.java and change the values that you scale the inputs by to make cut power mode cut more or less power.