Logical Operators

You've worked with if statements in a few lessons already and seen how they can be useful for having code run or not run based on conditions that arise. Often these conditions are simple – for example, if a button pressed, drive in cut power. However sometimes you might have more complicated conditions. Let's revisit your project from lesson 9, DriveStraight, for an example of this. In the DriveStraight project, you added an if statement to line 50 of Drivetrain.java to update the target heading if the user was turning the robot:

```
50      if (Math.abs(zaxisRotate) > Constants.Z_AXIS_ROTATE_DEADBAND_VALUE) {
51        targetHeading = m_gyro.getAngleZ();
52      }
```

This if statement allowed you to turn the robot and also have your orientation-correcting code run, however, it resulted in jerky behavior because it would stop updating the heading immediately after turn joystick stopped registering, even though the robot still had angular momentum meaning the turn was not yet finished yet. We could solve for this problem with a slightly more refined if statement: if the user is actively turning the robot, *or* there's an existing turn that is still finishing, update the target heading. In this sentence, the word "or" is a logical operator. We can use it in an if statement, and if either of the conditions surrounding the "or" or true, the if statement returns true. There are a number of logical operators in Java, but we'll focus on three main ones: and, or, and not.

All three operators are simple enough to understand and use, although you can create complicated logical conditions by combining them. For now let's focus on basics.

- The *and* operator will return true if *both* conditions it checks are true, and is represented by two ampersands, or "&&". If either condition is false, the and statement will return false. Example:

```
if (hour < 12 && date == 10) {
  System.out.println("It's morning on the tenth day of the month!");
}
```

- The *or* operator will return true if *either* of the conditions it checks are true. It will also return true if *both* of the conditions are true. As long as one of the conditions is true, it does not matter if the other conditions are true or false. The or operator is represented by two "pipes", or "||". Example:

```
if (hour < 12 || date == 10) {
  System.out.println("It's either the tenth day of the month, or morning of any day!");
}
```

- The *not* operator will return true if and only if the condition it's checking is false. So it simply returns the opposite of whatever condition it checks. The not operator is represented by a "bang", or exclamation point (!). Example:

```
if (!morning) {
  System.out.println("It's not morning!");
}
```

Different people and different organizations have different feelings on using the not operator. Some people prefer to check if a value is equal to false over using the not operator because they find it easier to read. Here's an example of the same if statement, but without the not operator:

```
if (morning == false) {
  System.out.println("It's not morning!");
}
```

Regardless of your, your teacher's, or your future employer's opinion on this, you'll need to understand the not operator. You'll also see it used in combination with an equal sign to represent "not equal to". Here's an example of that:

```
if (hour != 12) {
  System.out.println("It's not 12 o'clock.");
}
```

You can also combine these operators. You can group conditions together using parentheses to make sure the operators work on exactly the text you want them to. Here's an example:

```
if (hour < 12 && (date == 10 || date == 15)) {
  System.out.println("It's morning on the tenth or fifteenth day of the month!");
}
```

There is no limit to how many clauses you could put in a single if statement, although you don't want to overdo it because if you do your code will become difficult to follow. You can split individual conditions out into boolean variables on their own line to simplify things a bit. Example:

```
boolean morning;
boolean dateIsTenthOrFifteenth;

if (hour < 12) {
  morning = true;
}
else {
  morning = false;
}

if (date == 12 || date == 15) {
  dateIsTenthOrFifteenth = true;
}
else {
  dateIsTenthOrFifteenth = false;
}

if (morning == true && dateIsTenthOrFifteenth == true) {
  System.out.println("It's morning on the tenth or fifteenth day of the month!");
}
```

When using boolean variables, such as "morning" and "dateIsTenthOrFifteenth" above, you don't need to use the == operators – you can simply use the variables since they evaluate to true/false already. Again, different people have different opinions on whether this makes the code more or less readable, but you need to be able to understand it either way. Here's the final if statement from above but stylized differently. This has exactly the same meaning as the final if statement from the example

above:

```
if (morning && dateIsTenthOrFifteenth) {
  System.out.println("It's morning on the tenth or fifteenth day of the month!");
}

if (hour < 12 || date == 10) {
  System.out.println("It's either the tenth day of the month, or morning of any day!");
}
```

And lastly, you can use the evaluation of a statement to set the value of a boolean variable, without using if statements. The same warning about opinions on way to structure code applies. Here's an example. Note the difference between the single = sign being used to assign a variable a value, and the double == signs being used to evaluate an expression.

```
boolean morning;
boolean dateIsTenthOrFifteenth;

morning = hour < 12;
dateIsTenthOrFifteenth = (date == 12 || date == 15);

if (morning && dateIsTenthOrFifteenth) {
  System.out.println("It's morning on the tenth or fifteenth day of the month!");
}
```

The DriveStraight Project

Now that you understand what logical operators are and how they can be used to create compound conditions, let's fix that issue with the DriveStraight project. Right now, our condition is "the driver is pressing on the turning joystick", but we really want a condition along the lines of "the driver is pressing on the turning joystick *or* they were very recently pressing on the turning joystick, and that turn hasn't finished yet." The second clause of that statement covers the time between when the driver releases the joystick, and the turn completes. But how do we actually write that in code? There are different ways one could go about defining that. We'll use a fairly simple one and just use a *timer* to check how much time has passed since the driver released the joystick. Presumably, the turn will finish within some fairly small amount of time after they release the joystick. If the drive straight code is off for a fraction of a second after the turn finishes, it's not a big problem, so we don't need to worry about our estimate for the time being slightly off.

The WPILib Timer Class

The WPILib has a built-in Timer class you can use for timing things in your code. The capabilities of this class are fairly straightforward – you declare a Timer object, and you can start it, stop it, check how much time has elapsed, and reset it to zero. In your DriveStraight project, open Drivetrain.java, and after you declare your targetHeading variable on line 19, add a new line and declare a Timer object called "turnTimer". You'll get a red line under Timer, so use the lightbulb to import the Timer class from edu.wpi.first.wpilibj. This is important because there are other Timer classes that you'll see which are built in to the Java programming language, but we want the WPILib one. You can initialize your Timer in the same line you declare it:

```
18
19  private double targetHeading;
20  private Timer turnTimer;
21
22      Import 'Timer' (edu.wpi.first.wpilibj)
23      Import 'Timer' (java.util)
24      Import 'Timer' (javax.management.timer)
```

```
20  private double targetHeading;
21  Timer turnTimer = new Timer();
```

If you create a new line on line 52 and type “turnTimer.” You’ll get a dropdown of various Timer methods you have access to. You can scroll through this to see the various capabilities of the Timer class. This is good for seeing what’s available to you, but delete this line for now. The methods we’re most interested in for the timer are start, stop, and reset. Resetting the timer is important because if we don’t reset it after each turn, then the next time the user turns, the grace period we check for using the timer will have already elapsed because the timer started from a value above the grace period, instead of starting from zero.

```
51  public void arcadeDrive(double xaxisSpeed, double zaxisRotate) {
52    turnTimer.
53    if (Math.a
54      targetHe
55    }
56
57    double gyr
58
59    zaxisRotat
60
61    m_diffDriv
62  }
63
64  private doub
```

```
advanceIfElapsed(double seconds) : boolean
equals(Object obj) : boolean
get() : double
getClass() : Class<?>
hasElapsed(double seconds) : boolean
hasPeriodPassed(double period) : boolean
hashCode() : int
notify() : void
notifyAll() : void
reset() : void                    Timer.reset() : v…>
start() : void
stop() : void
```

There are a few conditions we need to think about in order to make this work. First off, is the user turning? If so, we need to update the heading. We also need to reset the timer constantly when this is true, because we want to start the grace period from when the user *releases* the joystick, not from when they first press it. The amount of time they hold the joystick down for varies and can’t be predicted, so timing the grace period from when they first press it won’t work. Another important condition is, is the user done turning but the time is within the grace period? If so, we want to continue updating the heading, but we do NOT want to reset the timer, because now we need to time the grace period. Lastly, we want to know if the timer is higher than the grace period value, because if it is, we consider the turn to be finished. In this case we want to stop updating the heading, stop the timer, and reset the timer for future use. Here are the conditions in a list:

- Is the user turning? -> Update the heading, reset the timer, and start the timer

- Is the user NOT turning, BUT the timer is within the grace period? -> Update the heading but DO NOT reset the timer
- Is the user NOT turning, AND the timer is greater than the grace period? -> Do NOT update the heading, but DO reset and stop the timer.

There are a number of different ways you can implement these conditions in your code, and there isn't necessarily a single right way to do it. A good way to approach it is to identify three key boolean variables:

- Is the user turning?
- Is the timer within the grace period?
- Does the heading need to be adjusted *specifically because of the timer?*

In your arcadeDrive method, try creating these three boolean variables. Note that for the third one you need to check if the timer is in the grace period, but you also need to check that the timer is greater than zero, because the timer starts at zero. This means that even if the user has never turned, if you simply check that the timer is less than the grace period, it will return true and your code will update the target heading *all the time*, even when not turning. This would result in your code never correcting the robot's heading because it will always set the target heading to the robot's current heading. This means you have two conditions to check – that the timer is within the grace period *and* that the timer is greater than zero. This is a chance to try out using the and operator. For the grace period, create a constant in Constants.java. You can play around with values to see what works best. A good starting point is .5 seconds. Here's an example of creating these three variables:

```
51    public void arcadeDrive(double xaxisSpeed, double zaxisRotate) {
52      boolean userIsTurning = Math.abs(zaxisRotate) > Constants.Z_AXIS_ROTATE_DEADBAND_VALUE;
53      boolean timerWithinGracePeriod = turnTimer.get() < Constants.TURN_TIMER_TURNING_GRACE_PERIOD;
54      boolean adjustHeadingDueToTimer = turnTimer.get() > 0 && timerWithinGracePeriod;
```

With these three variables, you can now look back at the list of conditionals (the first list of bullet points in this section) and create all the necessary if statements. Give it a try, using the variables and logical operators to emulate the logic described. Inside the if statements, put the necessary logic. The only lines you need inside the statements are lines to start/stop/reset the gyro (turnTimer.start(), .stop(), and .reset(), respectively) and the line to update the targetHeading. When you're done your code might look something like this:

```
    if (userIsTurning) {
      turnTimer.reset();
      turnTimer.start();
      targetHeading = m_gyro.getAngleZ();
    }

    if (userIsTurning == false && adjustHeadingDueToTimer) {
      targetHeading = m_gyro.getAngleZ();
    }

    if (userIsTurning == false && adjustHeadingDueToTimer == false) {
      turnTimer.stop();
      turnTimer.reset();
    }
```

When you've done this, it's time to test your code. With the code you've previously written in this method, your completed method might look something like this:

```
public void arcadeDrive(double xaxisSpeed, double zaxisRotate) {
  boolean userIsTurning = Math.abs(zaxisRotate) > Constants.Z_AXIS_ROTATE_DEADBAND_VALUE;
  boolean timerWithinGracePeriod = turnTimer.get() < Constants.TURN_TIMER_TURNING_GRACE_PERIOD;
  boolean adjustHeadingDueToTimer = turnTimer.get() > 0 && timerWithinGracePeriod;

  if (userIsTurning) {
    turnTimer.reset();
    turnTimer.start();
    targetHeading = m_gyro.getAngleZ();
  }

  if (userIsTurning == false && adjustHeadingDueToTimer) {
    targetHeading = m_gyro.getAngleZ();
  }

  if (userIsTurning == false && adjustHeadingDueToTimer == false) {
    turnTimer.stop();
    turnTimer.reset();
  }

  double gyroAdjust = getGyroAdjustment();
  zaxisRotate -= gyroAdjust;

  m_diffDrive.arcadeDrive(xaxisSpeed, zaxisRotate);
}
```

Go ahead and deploy this and let's test it out!

Testing the Code

Let's talk briefly about how to test robot code. As we do more advanced projects, testing becomes more and more important because it's often easy to make a mistake with your code and not notice it the first time you run your program. You want to test your code in as many situations as possible to make sure you're not missing anything. There is a lot of thought and research dedicated specifically to testing, but for today we'll just cover some basic concepts. First, you want to test as many

edge cases as possible. The definition of an edge case changes slightly depending on the context, but in general, it's a situation that is rare and exists near the "boundaries" of your logic. For example, say you had an if statement that changes the behavior of your program if some variable is greater than 100. Some edge cases to check would be when the variable is 99 (clearly not greater than 100), 100 (also not greater than 100, but might return true if you accidentally used "greater than equal to" in your code instead of greater than), and 101, which is clearly greater than 100. For the program we're working with right now, some edge cases would be to make sure that the robot works if you *never* turn it (e.g. to put it back on the tray and rotate it around, and to make sure it maintains its orientation), to make sure it works if you turn it very briefly, to make sure it works if you turn it for an extended period of time, and to make sure it still passes the tray test *after* you turn it.

Another concept in testing is to add outputs to your program so you can see what is happening in real time. Outputs can take various forms. When we turned on an LED based on the robot's orientation, that was an output in a literal sense – something we can see that gives us feedback on what's going on. Other outputs are all the data you see on the Robot Simulation interface. In this case, let's add a System.out.println() to get output in the terminal. A key variable for this program is the timer. If you output the timer's value after your if statements, you will be able to see the value in the terminal while you run your program:

```
71        System.out.println(turnTimer.get());
```

Doing this allows you to check and see that your timer is starting whenever you stop turning, and resetting after it hits your Constants value. You can use this to check that everything is behaving as you expect based on the value of this variable. If your code doesn't work right after you deploy it, this will be a good trick to debug and figure out where your mistake is.

But let's say your code *does* work immediately after you deploy it. That's good, but it can sometime lead to confusion of its own – *why* does the code work? Did you do everything correctly and you're now done, or are you not testing all of the cases? If you add your SOP line, you can help confirm that everything is correct. You can also test this by changing things to make it *incorrect* and then confirming that your code no longer works, before reverting the change. Let's try that. You declared a constant for the grace period for turning. What if we change the value of that constant? Try a few different values. If you set it to .05, that will be too short and you should be able to observe the same jerky behavior after turning. If you set it to 5, there will be a delay of several seconds after you finish turning before the timer finishes, and you can observe this in the terminal as you watch your timer increase in value to 5 before resetting. During this time you should be able to physically move your robot (pick it up, rotate it, and set it back down within the 5 seconds.) It will not correct its position during this time because it's still in the turning grace period. If you observe these behaviors when you test, but you do not observe them when you set the grace period value back to something reasonable like .5, then you can be confident your code is working.

Tidying Up

At this point, you could be done with the project, but there's a little bit of room for optimization and also another chance to practice your logical operators. Right now, we have two conditionals that

both update the target heading:

```
56      if (userIsTurning) {
57        turnTimer.reset();
58        turnTimer.start();
59        targetHeading = m_gyro.getAngleZ();
60      }
61
62      if (userIsTurning == false && adjustHeadingDueToTimer) {
63        targetHeading = m_gyro.getAngleZ();
64      }
```

Although there are exceptions, duplicated lines in if statements are often an indicator that you could optimize your code. If we rephrase our conditional slightly, we can eliminate this. Try writing three if statements using the following conditions:

- If the user is turning, reset and start the timer
- If the user is turning OR you need to adjust the heading due to the timer, update the heading
- If the timer is greater than the grace period, stop and reset the timer

After you do this, your conditionals might look something like this:

```
56      if (userIsTurning) {
57        turnTimer.reset();
58        turnTimer.start();
59      }
60
61      if (userIsTurning || adjustHeadingDueToTimer) {
62        targetHeading = m_gyro.getAngleZ();
63      }
64
65      if (turnTimer.get() >= Constants.TURN_TIMER_TURNING_GRACE_PERIOD) {
66        turnTimer.stop();
67        turnTimer.reset();
68      }
```

Notice how not only did you shave one line of code off of your program, but there's less text overall. In software development, less is usually better, because the more code you have, the more work you have to do to read through it, test it, make changes, etc. Less is good. Once you've made these changes, test your code (with or without the SOP statement), and when it works, you're done!