Note: there is no included solution program for lesson 10.

In this lesson you'll work with the Romi's autonomous mode and learn more about commands and how they can be combined together using *command groups*, which are lists of commands that run sequentially one after another, or potentially at the same time if they act on different subsystems. For today we'll focus only on commands running sequentially. Learning about commands and command groups will connect a lot of the concepts that you've been working with and when you finish this lesson, you'll have a much greater understanding of how all these classes, files, and methods work together to make your robot code run. For the project you'll make your own autonomous routines.

Commands

You've used commands already in the course, but we haven't covered the details of how they're structured. Let's talk about that a little bit more. The RomiReference project is written using the "command-based" paradigm, which allows a lot of things to happen without us doing them manually, for example your arcadeDrive method running all the time. There are other ways to write robot programs that aren't command-based, but this course focuses on command-based because it has a lot of advantages which will become clear to you during this lesson. We will not discuss other paradigms in this course, but you're free to explore them on your own.

In the command-based paradigm, there is one class that is responsible for doing all the heavy lifting behind the scenes, called the CommandScheduler. We discussed this very briefly in the loops lesson and you can find this object on line 45 of Robot.java. It's accessed every cycle of robotPeriodic, which runs constantly while your robot is turned on. The CommandScheduler looks at all the commands in your code, figures out which of them should be started, executed, and ended, and performs those actions. In the process of doing this, the CommandScheduler is looking for specific things in a command, for example the execute() method. It will call the execute method, which is how the code you've put in the execute method of the commands you've written or modified gets run. We've also discussed how only one command can run on any given subsystem (like Drivetrain) at once. The CommandScheduler is what manages this, overriding old commands when new commands are issued. If you wanted to, you could click on the green text "CommandScheduler" on line 45, press F12, and start investigating it yourself. However, this is pretty advanced and I don't necessarily recommend you do this – it's just interesting to know that you could. Most people programming Romis never need to do this and you can just let the CommandScheduler run, working its magic behind the scenes. However, you will need to understand the various parts of what makes up a command, so let's take a look at one that you've already worked with.

Create a new RomiReference project called "Autonomous" and open ArcadeDrive.java. Let's dive into the different portions of this file. All commands share a number of features, by nature of being a class, and by nature of being a command.

- Lines 12-14: class fields. You've learned about these as part of learning about classes and doing the projects. All classes in Java have this portion, although you could have a class without any class fields. (This "area" of the code technically still exists, even if you don't have any fields there.)

- Lines 24-32: class constructor, and where parameters are passed to the class. You've worked with this as well. All Java classes have constructors, although like with the fields, they aren't always explicitly declared.
- Lines 34-36: initialize method. On line 35 you'll see "@Override". We'll talk about overrides more in lesson 14. Whenever you see this tag, it means that the method exists *because of what kind of class it is*. In this case, that means that all classes that are commands have initialize() methods. Whenever you create a new command using the VSCode interface, it will have an initialize method. If you read the comment on 34, it explains the initialize method: "Called when the command is initially scheduled." Scheduling a command is often the same as running it, although not always, as it's possible that the scheduled command will have to wait for the subsystems it needs to become available to it. Don't worry about that right now – the point of the initialize method is that it runs when the command is scheduled, or in other words, when it's sent to the CommandScheduler to be run.
- Lines 38-42: execute method. We've talked about this one a bit already. Since it has an override tag, you know this is another method that exists because it's a command. This method gets called constantly while the command is running. In the case of ArcadeDrive, it's how you drive your robot.
- Lines 44-46: end method. This method gets called when the command is either interrupted, such as by another command overriding it using the same subsystem, or when the command finishes (see the next bullet.)
- Lines 48-52: isFinished method. This method runs every cycle as well, just like execute. If it ever returns true, the command schedule will terminate the command and call the end method. In the case of ArcadeDrive, we always want to be able to drive the robot, so this method is hardcoded to always return false, so the command does not end. This is also the default code that you'll see any time you create a new command. However, many commands *should* end after they do something. For example, say you had a command to drive forward ten inches in autonomous mode. Such a command should end after the robot drives ten inches forward. So instead of always returning false, isFinished would check how far the robot has traveled, and return true or false accordingly.

These four methods – initialize, execute, end, and isFinished – are the four main features of a command. Let's look at another command that uses more of these features. In the commands folder, open DriveDistance.java. As you might infer from the name, this command makes the robot drive a specified distance, in inches. Let's go through this command line-by-line and gain a complete understanding.

- Lines 10-13:

```
10  public class DriveDistance extends CommandBase {
11    private final Drivetrain m_drive;
12    private final double m_distance;
13    private final double m_speed;
```

Line 10 declares the class, and the "extends CommandBase" text specifies that this class is a command. We'll talk about this in a future lesson – for now you can just recognize it and understand

that it means it's a command. Then there are three class fields declared – the Drivetrain system to operate upon, and the speed and distance with which to drive.

- Lines 24-28:

```
23  public DriveDistance(double speed, double inches, Drivetrain drive) {
24    m_distance = inches;
25    m_speed = speed;
26    m_drive = drive;
27    addRequirements(drive);
28  }
```

This is the constructor for DriveDistance – remember, constructors have no return types, and their method names match their class names. *Only* constructor methods have these features. This constructor initializes the class variables declared on lines 11-13, and calls the addRequirements method to specify that this command requires the drivetrain. Specifying requirements is how the CommandScheduler knows how to manage which commands run on which subsystems.

- Lines 30-35:

```
30  // Called when the command is initially scheduled.
31  @Override
32  public void initialize() {
33    m_drive.arcadeDrive(0, 0);
34    m_drive.resetEncoders();
35  }
```

These lines define the initialize method for DriveDistance. In this case, line 33 tells the robot to stop, by giving it zero power and zero turn. Line 34 resets the encoders to make sure that they're measuring from zero when the command starts. If the robot had already been driven around (or rolled by hand) when the command starts, and the encoders weren't reset, then the distance measurements would be wrong. There are other ways to account for that, but by default, this command just resets all the measurements to zero when the command is scheduled.

- Lines 37-41:

```
37  // Called every time the scheduler runs while the command is scheduled.
38  @Override
39  public void execute() {
40    m_drive.arcadeDrive(m_speed, 0);
41  }
```

These lines define what the command should do while executing. This command is called "DriveDistance", and only drives in a straight line, so this method is pretty simple – drive the robot with the speed specified when the command was initialized, and supply 0 power to turning to go in a straight line. As this is the execute method, it will be called continuously while the command is running.

- Lines 43-47:

```
43    // Called once the command ends or is interrupted.
44    @Override
45    public void end(boolean interrupted) {
46      m_drive.arcadeDrive(0, 0);
47    }
```

These lines define the end method, which as the comment on line 43 states, is called when the command either finishes or gets interrupted. Commands end when their isFinished method returns true, and the most common way to be interrupted is if another command that requires a subsystem the command requires is scheduled. For example, you could add code to your robot such that if you press a button on your controller, it immediately defaults back to operator control, which would interrupt DriveDistance because operator control would require the drivetrain. Either way, this method tells the robot to stop when DriveDistance ends. You may want the robot to continue moving after DriveDistance ends. For example, the default autonomous mode will have the robot spin 180 degrees after it drives ten inches. This works because unlike the execute method, the end method is called only one time. So while the end method *will* stop the robot, it will be very brief and only last until the next line of code that sets the robot's speed/turning executes. (Of course, it is possible that the next line that sets the robot's speed continues to set its speed to zero.) Note that line 45 takes a boolean variable "interrupted", so you can know whether your command finished on its own or something interrupted it. This can be useful if you have logic that depends on knowing whether or not your command successfully completed or was overridden.

- Lines 49-55:

```
49    // Returns true when the command should end.
50    @Override
51    public boolean isFinished() {
52      // Compare distance travelled from start to desired distance
53      return Math.abs(m_drive.getAverageDistanceInch()) >= m_distance;
54    }
```

These lines define the last method if of the DriveDistance class/command, the isFinished method. All the other isFinished methods we've looked at so far simply return false, however for driving a set distance we want to stop after the distance has been driven, so this isFinished method needs to actually check something. In this case, line 53 calls a method called "getAverageDistanceInch", which returns a number of inches that the robot has traveled using the encoders, and compares it to the distance the robot is supposed to travel. You can F12 into this method (and its sub-methods) if you want to see how it works, or you can take it for granted that it returns accurate values. Either way, if the distance traveled is greater than or equal to m_distance, isFinished will return true, and when that happens, the command's end method will run and then the command will be over. Note that Math.abs is used here because the distance returned by getAverageDistanceInch will be negative if the robot is driving in reverse.

We've now looked through every line of code in this class, and hopefully you now have a decent understanding of an entire file of Java code! Obviously there are still a few things we haven't gone in depth into just yet, but you should now understand the various methods and why they exist. Commands

can have more than just these methods. For example, you could define additional methods in DriveDistance that are called by these standard methods while your command is running. But these methods are the methods you'll see in every command and that you need to understand.

Command Groups

Now that you understand more about individual commands, it's time to call about *command groups*. Command groups are fairly simple – they are literally just groups of commands. The commands can be run in sequence or in parallel. In order to run them in parallel they need to run on different subsystems because only one command can be run on any given subsystem at once. For today we'll focus only on commands running sequentially. In a sequential command group, the logic flow is pretty straightforward. As soon as one command finishes running – its isFinished returns true, and its end method gets called – the next command will start running. Your code already has a sequential command group defined by default so let's take a look at it. Open AutonomousDistance.java in the commands folder.

Line 10 of AutonomousDistance declares the class and you'll see that it says "extends SequentialCommandGroup". Just like when we investigated DriveDistance, we'll ignore what the word "extends" means just for now, but just like in DriveDistance "extends CommandBase" meant that DriveDistance was a command, "extends SequentialCommandGroup" in AutonomousDistance means that AutonomousDistance is a sequential command group. That is a lot of words, so just to repeat: the text "extends SequentialCommandGroup" means that the class is a sequential command group.

Sequential command group classes tend to be fairly short and simple because basically all they do is list a bunch of commands that are meant to run in a sequence. AutonomousDistance is a great example of this. Other than line 10, this class only has a couple lines of code. Line 17 declares the constructor, which requires a drivetrain to act upon. Lines 18-22 are actually a single logical line of code, but it's split over several lines of code for clarity reasons. Notice how there is a semi-colon at the end of line 22, but not at the end of the other lines. We touched on semi-colons earlier, and you've been using them in your code. You may or may not have noticed this, but you can put as much whitespace as you want in your code without it having any effect, even on an individual line of code. There is only one semicolon for lines 18-22 because it's a single line, but with line breaks splitting out each sub-line after each comma to make it more readable.

We'll talk about the details of how it works in a moment, but chances are, if you look at lines 18-22 and read the text there, you can take a good guess at what it's doing, especially if you've already run your Romi's autonomous code. It makes the robot drive a distance, specifically 10 inches. Then it makes the robot turn 180 degrees. Then it drives ten more inches. Finally, it turns 180 degrees again. Each of these actions corresponds to one of the sub-lines between lines 19 and 22 of the file. Each of these sub-lines is a command, and as soon as one of them finishes, the next one automatically starts. That's the magic of sequential command groups. You don't need to worry about managing when your commands stop and start – the CommandScheduler does that for you.

Let's dive into the details. As discussed, line 18 is one method call. Click on "addCommands" and press F12 to see the method declaration. This takes you to line 31 of the definition for the class SequentialCommandGroup:

```
31    public final void addCommands(Command... commands) {
```

You can see here that addCommands takes a Command parameter, but instead of just saying "Command commands" it says "Command… commands". In Java, the three periods means that the method can take a variable number of objects of the specified type. It could be one, it could be zero, it could be ten or a hundred. We're not going to cover how the receiving method deals with receiving these objects, but just understand what the three periods mean. You can pass any number of Command objects to the addCommands method. Now go back to AutonomousDistance.java. On lines 18-22, we use the *new* keyword to create four Command objects to send to addCommands. A DriveDistance command, a TurnDegrees command, another DriveDistance command, and finally, one last TurnDegrees command. These four commands are declared inline and sent straight to addCommands. That is literally all that has to be done to create this command group – simply creating and adding the commands, and the CommandScheduler takes care of the rest, because for each of the commands, they have isFinished methods that can tell when they finish.

This makes command groups easy to create and modify. Let's try it out. See if you can add a couple more commands to AutonomousDistance. The only thing to watch out for is that on line 22, there is an extra ")", which closes the "(" on line 18, and also a semicolon. You can add commands in-between the existing commands or at the end, just ensure that the overall addCommands method call still ends with ");" after the last command that is added. You'll also need to add commas after each new command you add. Here's an example of adding a couple more commands:

```
17    public AutonomousDistance(Drivetrain drivetrain) {
18      addCommands(
19          new DriveDistance(-0.5, 10, drivetrain),
20          new TurnDegrees(-0.5, 180, drivetrain),
21          new DriveDistance(-0.5, 10, drivetrain),
22          new TurnDegrees(0.5, 180, drivetrain),
23          new DriveDistance(-0.5, 10, drivetrain),
24          new TurnDegrees(-0.5, 90, drivetrain),
25          new DriveDistance(-0.5, 10, drivetrain));
26    }
```

Try playing around and making a few auto routines of your own. You can also delete the existing commands and change the robot's speed. Note that if the robot's wheels slip on the surface you're driving on, it will make the sensors' readings of how far you've driven or rotated inaccurate. For practice exercises, try the following:

1. Drive the robot in a square with 12-inch sides.
2. Drive the robot in the shape of an equilateral triangle with 9-inch sides.
3. Drive the robot in a + shape, of a size of your choosing.
4. Be creative and make your own path!

Wrap-Up

Through these ten lessons, you've learned a lot about both programming robots and writing software using Java as the programming language. You now know enough to modify programs to write and modify basic programs. You can add your own commands and edit existing functionality. You've learned how to access different systems on the Romi such as the drivetrain, the LEDs, and the gyroscope, and how to interface with your controller and its joysticks and buttons. There's still a lot to

learn, and in the next course we'll dive deeper into some of the concepts we had to gloss over in this one, such as the "extends" keyword, the @Override tags, and how to build up your own programs from scratch. But if you've made it this far, you are well on your way to becoming an advanced programmer. Take a moment to be proud of what you've learned so far and feel free to spend some time having fun with the Romi. Don't be afraid to modify anything you find in the code and see what happens. Worst case, you can always create a new RomiReference project from scratch and start over! When you're ready, you can dive into Course 2 and continue your learning.