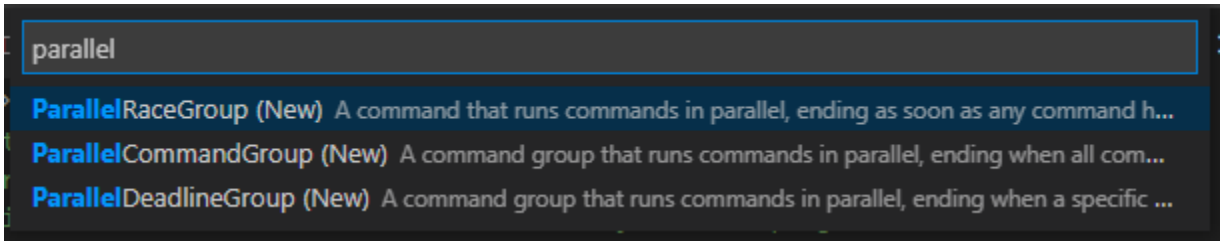
In lesson 10 you learned about commands and command groups and created some autonomous routines, by modifying a sequential command group, where commands are run in sequence, with the next one starting automatically after the last one finishes. In this lesson you'll utilize parallel command groups, where multiple commands can run at the same time across different subsystems. You may want to open lesson 10 for review as you complete this project. In the RomiReference project there are two subsystems, the drivetrain and the on-board IO system, which, among other capabilities, has LED lights that you've used a couple times throughout this course. You'll use these two subsystems in concert to create a parallel command group that runs commands on both systems.

Parallel Command Groups

Parallel command groups (PCGs) are fundamentally pretty similar to sequential command groups. The main difference is that the commands you add to the group using the addCommands method will all begin at the same time. There are a couple different types of PCGs, which you can see brief descriptions of in the dropdown interface you see when you create a new command. To get started, create a new RomiReference project called "ParallelLights", right click on the commands folder, and choose to create a new command. Type "parallel" into the filter bar and you'll see three options:



The simplest type of PCG is the default "ParallelCommandGroup". With this type, the command group will finish when all of the commands in it finish. With the Race group, the command will end as soon as any command finishes, and with the Deadline group, you can specify a specific command, and when that command finishes, the group will finish. Go ahead and create a PCG; it doesn't matter what you name it because you'll delete it and re-create it later when you do the project.

Combining Command Types

Now that you understand class hierarchies in Java, let's take a moment to reason about command groups and commands. In your newly created PCG, on the line declaring the class, you'll see that it extends ParallelCommandGroup, meaning that it is a child class of the parent class, ParallelCommandGroup. Example:

```
public class TurnDegreesAndTurnOnLights extends ParallelCommandGroup {
```

If you F12 into ParallelCommandGroup, you'll see that this, in turn, extends CommandGroupBase:

```
16    public class ParallelCommandGroup extends CommandGroupBase {
```

Both the ParallelCommandGroup and SequentialCommandGroup classes extend CommandGroupBase, meaning that both of these types of objects are also CommandGroupBase objects. If you F12 into CommandGroupBase, you'll that it extends CommandBase and implements Command. Implementing Command is the key thing to notice here. Anything that implements Command can be

used as a Command, because implementing Command tells Java that the class has the critical Command methods such as initialize, execute, etc. Since all the different types of command groups extend CommandBase, and CommandBase implements Command, all the command groups also implement Command, meaning that all command groups can also function as Commands. This is important because this means that when you're adding a command to a command group, you can add a command group instead of a basic command.

Let's look at an example. There is a lot of text to take in here, so feel free to read this a couple times until you wrap your head around it. The RomiReference project has a command group called "AutonomousDistance", which you've worked with in the past. It has four sub-commands that are added using the addCommands method. You could add additional commands to it, and those commands could themselves be command groups. This lets you combine various command types to create complex behaviors. For example, you could create a ParallelCommandGroup that turns lights on the OnBoardIO to turn on, while telling the Drivetrain to drive. You could then add that command to the sequential AutonomousDistance command group. Then as AutonomousDistance runs, it will start going through its commands in sequence. When it gets to your parallel command, it will run all the sub-commands of your parallel command. When all of those sub-commands finish, it will resume its original sequence. The combination of parallel commands, sequential commands, and sub-commands gives you very granular control over all the subsystems on your Romi and lets you easily build complex autonomous modes, once the commands themselves have been coded. It lets you re-use your base commands in different orders and sequences to create different autonomous modes. Mastering parallel and sequential commands, and how to combine them, is an important skill. It also takes a bit of practice so it's time for you to start getting that practice with the project.

Project – AutonomousDistanceWithLights

For the project you'll implement the example described above. The goal is to create a sequential command group that mimics the behavior of the pre-existing AutonomousDistance command group, but replace the sub-commands in AutonomousDistance with ParallelCommandGroup commands that both do the action the original sub-command did (either driving or turning), and also turn the on-board IO LED lights on or off. For this project, you can use the green and red LEDs on the Romi. You can refer back to lesson 3 to see how to create a command to turn the lights on and off. The final sequence of events should be as follows:

1. The Romi drives forward ten inches while turning its lights off.
2. The Romi turns its lights on while turning 180 degrees.
3. The Romi turns its lights off while once again driving ten inches (repeating step 1.)
4. The Romi turns its lights on while turning 180 degrees (repeating step 2.)
5. The Romi turns its lights off.

Steps 1 through 4 will be parallel command groups. Step 5 is a basic command, not a group. The five steps combine to form a sequential command group. When you're done, your autonomous mode should drive ten inches, turn the lights on while turning, turn the lights off while driving back, turn the lights on while turning again, and then turn the lights off and stop. Here are some steps you can take to do this project:

1. Create and implement two commands, TurnLedsOn.java and TurnLedsOff.java. Again, lesson 3 has an example of how to do this. Use the red and green LEDs.

2. Create a new sequential command group called AutonomousDistanceWithLights.java that will replace AutonomousDistance.java.
3. Update RobotContainer.java to use your new command group as the default autonomous command (you can ctrl+f “AutonomousDistance” to see where the default autonomous command is specified.)
4. Create two parallel command groups that you’ll use as replacements for the old driving and turning commands. You can call these “DriveDistanceAndTurnOffLights” and “TurnDegreesAndTurnOnLights” respectively.
5. Implement your two commands by adding their sub-commands. In this case the sub-commands will be the old methods they’re replacing (DriveDistance and TurnDegrees, respectively) plus the correct command that turns the LEDs on or off. You’ll need to pass in the right subsystems (Drivetrain and OnBoardIO) to your command groups and sub-commands.
6. Now that all your commands are created, go back to your new sequential command group, AutonomousDistanceWithLights, and add all the commands that it needs to complete the routine.
7. Before you run your program, you’ll need to specify that the LEDs should function as output, just like you did in prior lessons where you used the LEDs. You can find the line where this is done in RobotContainer.java by ctrl+f’ing “ChannelMode.”. Update the two ChannelModes to ChannelMode.OUTPUT. Example:

```
private final OnBoardIO m_onboardIO = new OnBoardIO(ChannelMode.OUTPUT, ChannelMode.OUTPUT);
```

8. Once you’ve completed all these steps, you’re ready to run and test your autonomous routine!