

In Lesson 1, you got your Romi driving around and made a small code change. In this lesson, we'll talk a little bit about Java as a programming language so you can start to make sense of all the folders and files you see in VSCode.

Printing Output

One of the simplest things you can do in Java is tell the program to output text. You can do this by calling one of two "methods". We will talk about what a method is shortly, but for now just think of it as some piece of code can you "call", or invoke, to do something. Java executes code starting at the top of the file, and then executes lines one by one from top to bottom of the file. So, inserting lines into a program will result in them being executed. The two methods to output text are:

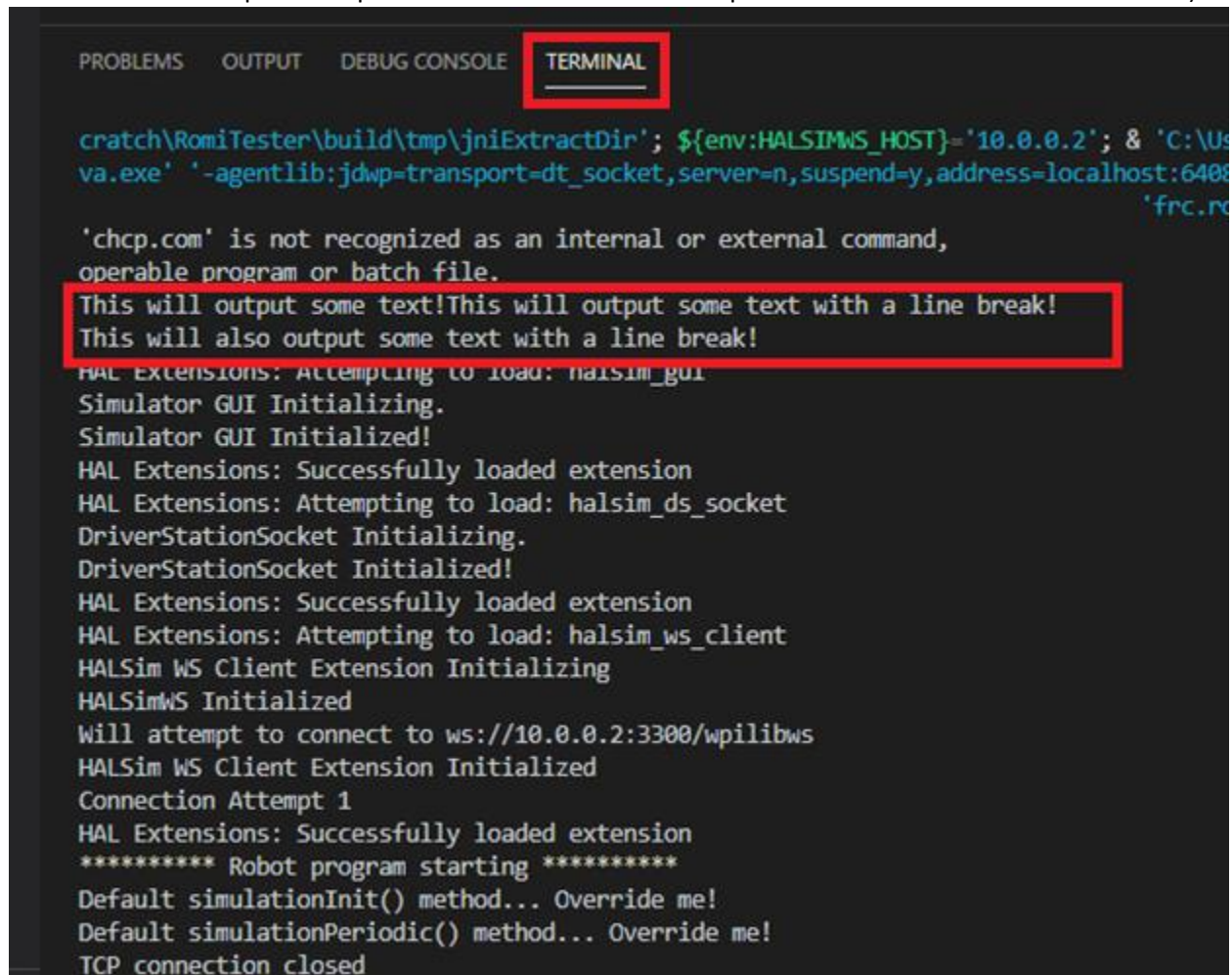
```
System.out.print("This will output some text!");  
System.out.println("This will output some text with a line break!");
```

In both the method calls above, the text in quotation marks will be outputted. In the second one, a line break will be added to the end of the text. The only difference is the addition of "ln" (with a lowercase l) after "print", which is abbreviation for "line". So, in English, system-dot-out-dot-print, and system-dot-out-dot-print-line. You can add these lines of code to your Romi program and try them out for yourself. Open Main.java, which is in the same location as RobotContainer.java from the previous lesson (explorer->src->main->java->frc->robot) and scroll down to line 23. Here you can insert these lines. Make sure to put them *before* the line that says RobotBase.startRobot(Robot::new);. Here is an example:

```
13  /  
14  public final class Main {  
15      private Main() {}  
16  
17      /**  
18       * Main initialization function. Do not perform any initialization here.  
19       *  
20       * <p>If you change your main robot class, change the parameter type.  
21       */  
22      Run | Debug  
23      public static void main(String... args) {  
24          System.out.print("This will output some text!");  
25          System.out.println("This will output some text with a line break!");  
26          System.out.println("This will also output some text with a line break!");  
27          RobotBase.startRobot(Robot::new);  
28      }  
29  }
```

After you do this, you press F5 to run your program and click on the TERMINAL output window; example shown below. (It will take a few seconds to compile and run. You do not need your Romi powered on to do this test. You may need to scroll up in the terminal window, but if you look closely, you will see your output text. Here's the output from running the example above; you can see the

difference between print and println because some of the outputs create new lines and some do not.)

A screenshot of a terminal window with a dark background. At the top, there are tabs labeled 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is selected and highlighted with a red rectangle. The terminal shows a command being executed: `cratch\RomiTester\build\tmp\jniExtractDir'; ${env:HALSIMWS_HOST}='10.0.0.2'; & 'C:\Us` followed by `va.exe' '-agentlib:jdwp=transport=dt_socket,server=n,suspend=y,address=localhost:6408` and `'frc.ro`. Below the command, the output shows an error: `'chcp.com' is not recognized as an internal or external command, operable program or batch file.` This is followed by two lines of text enclosed in a red rectangle: `This will output some text!This will output some text with a line break!` and `This will also output some text with a line break!`. The output continues with several status messages: `HAL Extensions: Attempting to load: halsim_gui`, `Simulator GUI Initializing.`, `Simulator GUI Initialized!`, `HAL Extensions: Successfully loaded extension`, `HAL Extensions: Attempting to load: halsim_ds_socket`, `DriverStationSocket Initializing.`, `DriverStationSocket Initialized!`, `HAL Extensions: Successfully loaded extension`, `HAL Extensions: Attempting to load: halsim_ws_client`, `HALSim WS Client Extension Initializing`, `HALSimWS Initialized`, `Will attempt to connect to ws://10.0.0.2:3300/wpilibws`, `HALSim WS Client Extension Initialized`, `Connection Attempt 1`, `HAL Extensions: Successfully loaded extension`, `***** Robot program starting *****`, `Default simulationInit() method... Override me!`, `Default simulationPeriodic() method... Override me!`, and finally `TCP connection closed`.

In general, you do not want to put code in the location where we put these output statements, so after you run your test and get it working, delete these lines. We will talk more about why this is later.

Methods

With the output statements above, you've added your first method calls. In Java and most other programming languages, complex programs are broken down into many smaller chunks of code by writing methods. While we used a built-in method with `System.out.println`, you can also write and call your own methods. Almost everything that happens in a Java program is a compilation of methods, because methods can call other methods. As an example of this, think about a laptop computer, when you turn it on. When it turns on, a lot of things need to happen – it needs to start powering its hardware, it loads its operating system, it turns on the screen, etc. – but you only push one button to make all of this happen. You can think of pushing the button as calling a method that turns on the computer. That method might call some of sub-methods, for example one to power the hardware, one to load the operating system, and so forth. The code to do each of these individual tasks is localized in their own methods, and the person calling the method (in this case you, when you power on the

machine) doesn't really care about *how* those things work, you just call the method by pushing the power button and everything works. Methods are a critical building block of programs because they allow code to be re-used – the code to, for example, open a web browser window might be called when you click on the browser icon on your desktop, but also if you click a link in this document and it opens a window on its own. In both cases, a method to open a browser window was called, but the person programming that functionality only had to write that method once – they did not have to re-write it for every possible way to open a browser window.

Variables

We'll talk more about methods in a bit, but before we do that, let's talk about another concept called a "variable". A variable is some value that exists in your program that has a name and can be changed and used. For example, you could have a variable called "robotSpeed" that represents the current speed of your Romi in inches per second. Since the Romi has encoders on the drive motors, which measure the rate of rotation, you could do some math to determine the current speed at which the wheels are spinning, and then store that value in robotSpeed. Then you could do something like outputting the speed to the terminal using System.out.println (SOP) and get a real-time update of your robot's speed. You could even write a method that does all of this for you!

Variables have different *data types*. For example a data could be an integer, in Java called an *int*, or it could be a decimal, which in Java could be called a *double*. There are other ways to represent these numbers in Java, but for now we'll focus on int and double.

To create, or *declare*, a variable you give it a name and a data type. In Java you do this like so:

```
int numberOfWheelsOnRobot;  
double robotSpeedInchesPerSecond;
```

You can assign values to variables using *the assignment operator*, which is an equal sign. If you've taken algebra in school, then you'll be used to the equal sign meaning something slightly different – it signifies that two halves of an equation are, in fact, equal. In Java, it *sets* the first half of the equation equal to the second half. (If you haven't had algebra yet, don't worry about this comparison.) Here's an example of assigning values to variable:

```
numberOfWheelsOnRobot = 2;  
robotSpeedInchesPerSecond = 3.1;
```

You can also declare and *initialize*, or set a value, for a variable in a single line of code:

```
int numberOfWheelsOnRobot = 2;  
double robotSpeedInchesPerSecond = 3.1;
```

Once variables exist, they can be changed. For example, a robot's speed changes frequently during operation, so if you were to have a variable representing the robot's speed, it would need to constantly updated. You do this the same way as you did when you assigned the variable's value, using the equal sign. Here is a short sample program that declares and initializes two variables, outputs their values, and then updates one of them and outputs its value again. There are a couple important things to notice here. First, we're declaring and initializing variables. Then we use SOP to output the values, *but instead of a hardcoded text string, we're using a variable when we call SOP*. Then, we change the value

of one of the variables and output it again. Also, we change between using print and println, when we want to have text on the same line vs using line breaks. Try writing this program for yourself in the same place in Main.java where you tested your SOP statements earlier.

```
Run | Debug
22 public static void main(String... args) {
23     int numberOfWheelsOnRobot = 2;
24     double robotSpeedInchesPerSecond = 3.1;
25
26     System.out.print("Number of wheels on the robot: ");
27     System.out.println(numberOfWheelsOnRobot);
28
29     System.out.print("The robot's current speed: ");
30     System.out.println(robotSpeedInchesPerSecond);
31
32     // The robot has stopped! Update the speed to 0 inches per second.
33     robotSpeedInchesPerSecond = 0;
34
35     System.out.print("The robot stopped! Now its new speed is: ");
36     System.out.println(robotSpeedInchesPerSecond);
37
38     RobotBase.startRobot(Robot::new);
39 }
40 }
41 |
```

If you haven't clicked the red square toward the top of your screen to stop your prior program from running yet, do so now. As mentioned in lesson 1, you'll want to click the red square to stop the already-running program each time you want to deploy a code change, so keep that in mind going forward. Here is the output from that program in the terminal (again, you may need to scroll up to see it):

```
'chcp.com' is not recognized as an internal or external command,
operable program or batch file.
Number of wheels on the robot: 2
The robot's current speed: 3.1
The robot stopped! Now its new speed is: 0.0
HAL Extensions: Attempting to load: halsim_gui
Simulator GUI Initializing.
Simulator GUI Initialized!
HAL Extensions: Successfully loaded extension
HAL Extensions: Attempting to load: halsim_ds_socket
DriverStationSocket Initializing.
DriverStationSocket Initialized!
HAL Extensions: Successfully loaded extension
HAL Extensions: Attempting to load: halsim_ws_client
HALSim WS Client Extension Initializing
HALSimWS Initialized
```

The last thing you might notice about this short program is that on line 32, there is text in green that reads as an English sentence, not Java code. The line is prefaced by two forward slashes - `//`. Putting two forward slashes in a line indicates that the rest of that line is a *comment* – Java will ignore it, so you can type whatever you want without causing an error. This is a helpful tool for writing text that explains what the code does, for humans to read. We'll talk more about comments later. For now, undo all your new additions to `Main.java`, because as mentioned before we generally don't want to put code here. Next we'll do a small project to get you a little bit more practice with methods and variables so you can see how you can use them to modify the behavior of your Romi.

Modifying Your Romi's Drive Behavior

Let's take a look at how you can use variables in code to change the behavior of your robot. Open the file called `Drivetrain.java`. You can find it in the "Subsystems" folder in `src/main/java/frc/robot`. On lines 45-47 there is a method called `arcadeDrive`. Unlike `System.out.println`, this method is a custom-created method in our code, as opposed to a method native to the Java programming language. We will talk more about user-created methods soon. For now, take a look at line 45. There are parentheses, inside of which you see "double `xaxisSpeed`, double `zaxisRotate`". These are variables that are being declared. This is a slightly different way to declare variables which we'll talk more about soon. What's important about these variables is when this code runs, they are assigned values dynamically. This is different than when you declared variables above and set them equal to a value. Again, we'll discuss it more soon, but for now, just know that whenever this code runs, these variables *are assigned the values that are read from the joysticks*. Remember, the joysticks range from -1 to 1, so that's the possible range these variables could receive. If one joystick axis is halfway forward and the other is all the way back, they would have values of .5 and -1, respectively.

On line 46, we call another method, and *pass in* the same variables. We didn't really talk about "passing in" when we were using SOP, but to pass in means to send some information to a method. With SOP, you're passing in the text you want to print. The method prints whatever it's given, so you give it what you want to print. Line 46 functions the same way, except you're passing in joystick values instead. We want to get some practice seeing how variables can affect our robot. If you press F5 right now and deploy your code to your Romi, you can try driving it around for a moment. Get a feel for how fast it goes when you give it full throttle, half throttle, etc. We're going to modify the throttle power and then you can compare the two.

In this method, the throttle output is represented by the variable called "`xaxisSpeed`". To modify the power, simply insert an additional line on line 46, before the previously existing line 46, that looks something like this:

```
public void arcadeDrive(double xaxisSpeed, double zaxisRotate) {  
    xaxisSpeed = xaxisSpeed * .5;  
    m_diffDrive.arcadeDrive(xaxisSpeed, zaxisRotate);  
}
```

You don't have to use .5, but it's a good choice to make the effect obvious. *Do* choose a number between 0 and 1. Then run your code and try driving your robot around. You'll notice that the power output is scaled by whatever number you chose. Thinking about the code, this makes sense – you took the power and multiplied it by some coefficient. Now try a few other values. Try a number greater than

one, try zero, and try a negative number. Before you deploy each change, try to predict what will happen. Were your predictions correct?

A number greater than one will not increase the maximum speed of the robot, because that is physically limited by the hardware, however it will reduce controllability because even a small amount of joystick input will result in sending maximum power output. Zero will make any input from that joystick irrelevant. A negative number will scale the output by whatever coefficient you chose and also invert forward and backward. If you want, you can also try playing around with the `zaxisRotate` variable, or even both variables at once.

Playing with these variables demonstrates how you can use variables to control your robot. For example, with the default code, the turning controls are pretty sensitive – if you're going full power forward and try to turn at all full power at the same time, your robot will turn, but it will stop moving forward at all. It might be more comfortable to drive if the robot turned while *still* moving forward if both joystick axes are given max power. Scaling the rotate by .65 and leaving the speed at full power, for example, would achieve this, although it would reduce the max speed at which one could turn even when *not* driving forward. In a future lesson we'll cover how we could get the benefits of the scaled turn at full power without sacrificing anything. For now, you've developed an understanding of what variables are and how you can use them, and we'll keep building upon this knowledge.