Table of Contents:

Introduction:

About Us:

We are FTC Team 16660 CyberWing and we are located in West Anchorage High School in the state of Alaska! In addition to building robots and competing at FIRST events, we also strive to promote STEM education in kids of any age.
In this book, we will teach you how to advance in programming to compete on par with veteran teams. We will cover advanced programming topics such as using sensors, PID control and using encoders. Check out more about us here: https://westeagles.akfirstrobotics.org/

What is FIRST?:

For Inspiration and Recognition of Science and Technology, or FIRST for short, is a global youth-serving nonprofit organization that hosts FIRST LEGO League (FLL) events, FIRST Tech Challenge (FTC) events, as well as FIRST Robotics Competition (FRC) events. Through these challenges that it hosts, FIRST exposes young people to the world of STEM (science, technology, engineering, mathematics) and engages them to build skills in those areas while acquiring life capabilities such as self-confidence, communication, and leadership.

What is FTC?:

First Tech Challenge is one of the programs FIRST offers, aiming at kids age 12-18. It is considered a step up from FLL, as FTC introduces kids to more complex ideas in the field of STEM. However, fret not! FTC encourages people of any skill level to join and compete, and there are also many opportunities for people to gain interpersonal skills as FIRST also values communication in the community.

What is Java?:

Java is the programming language that we will be using to program the robot. The robot is controlled by two android devices communicating with each other and giving instructions to the REV Robotics Hubs. Since this book primarily focuses on the programming aspect of FTC, we will not have an in-depth explanation of setting up and building the robot.

Advanced Programming:

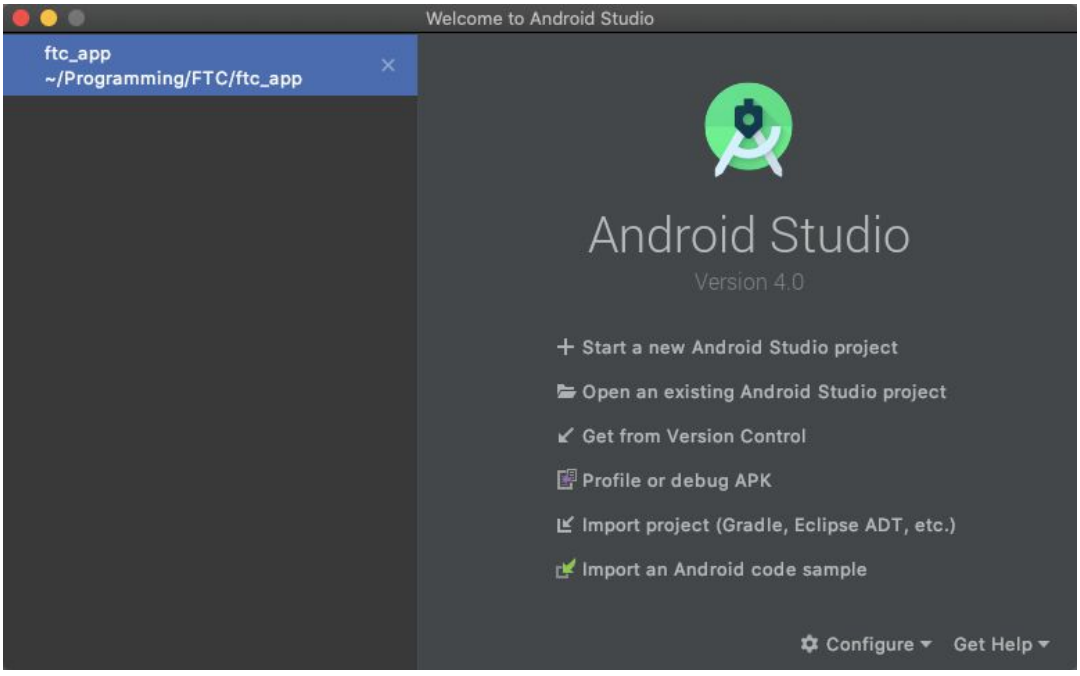
Setting up Android Studio:

This is the first difference between an experienced team and a less experienced team: the IDE (where people do the programming). Most veteran teams abandon OnBot Java at some point and migrate to another app called Android Studio. Why? Because it's much better for many many reasons. First, it's much more professional. Second, it has a much better User Interface than OnBot Java and it is simply much easier to navigate through your code. Third, it has code autocompletion, meaning Android Studio will *guess* what you are trying to type which is helpful in many ways. All in all, Android Studio >>> OnBot Java.

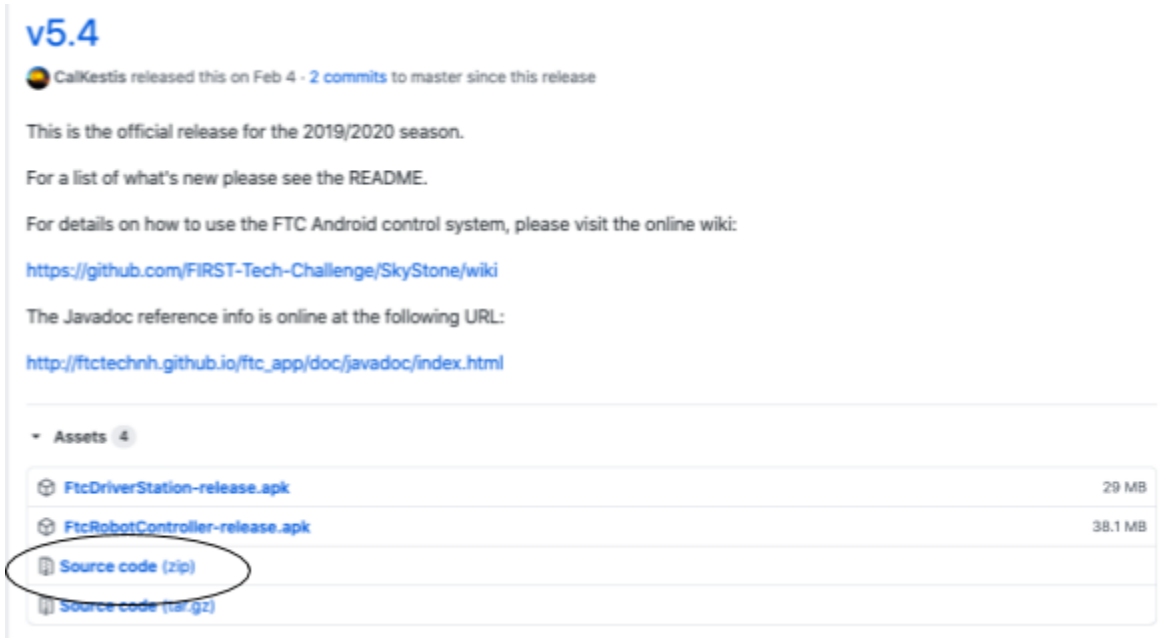
So why do people not start with Android Studio? Because it can be a *pain* to set up. We couldn't find a comprehensive guide to starting up with Android Studio on the internet, so here it is:

First of all, you need to download Android Studio to your computer. Here is the link to Android Studio: https://developer.android.com/studio. We are guessing anyone living in the 21st century knows how to download software to your local computer so we won't go in-depth, but what you have to do is accept the license terms and press DOWNLOAD ANDROID STUDIO. Then, launch the application and follow the instructions to install.

Once Android Studio is installed, it should look like this:

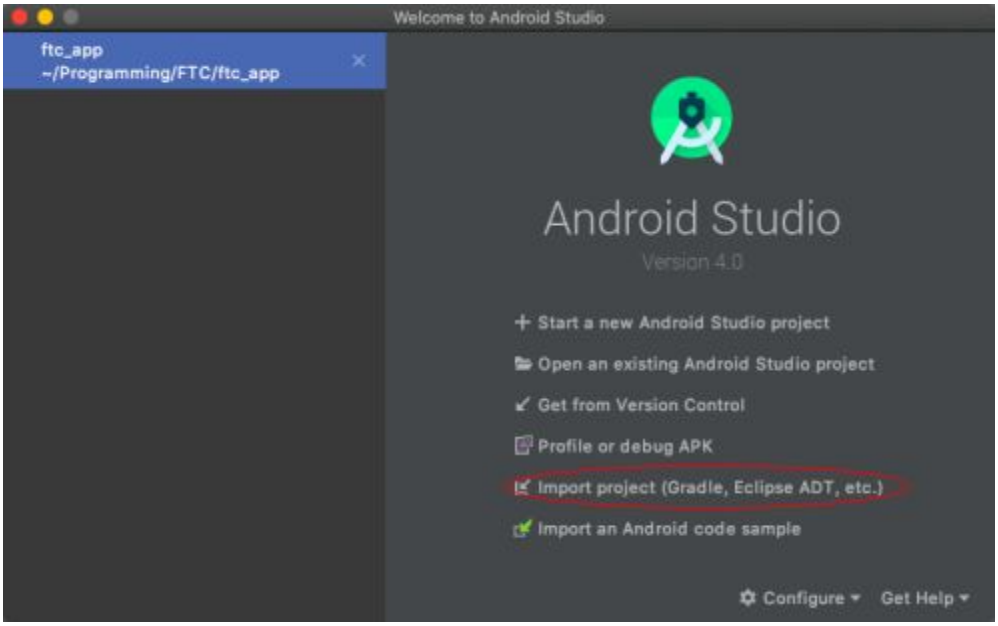


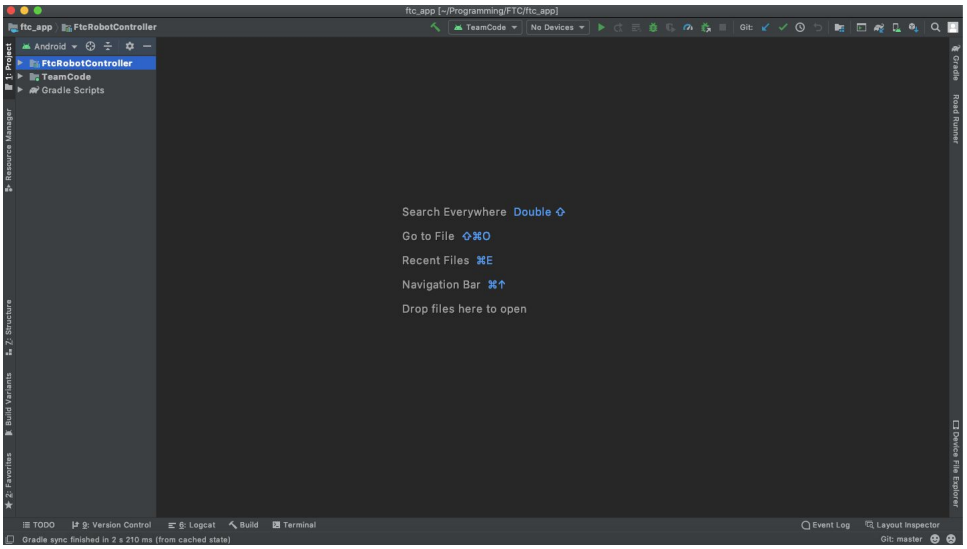
Next, you have to download the FTC SDK (Software Development Kit). First, go to this link: https://github.com/FIRST-Tech-Challenge. Then, click on the repository (the different options) that has the current season name (e.g. SkyStone for 2019/2020). Click on releases (on the right-hand side of the screen). And then click the Source Code (zip) for the latest release:



Extract the zip file to your desired destination. Then go back to Android Studio and click “Import project”:

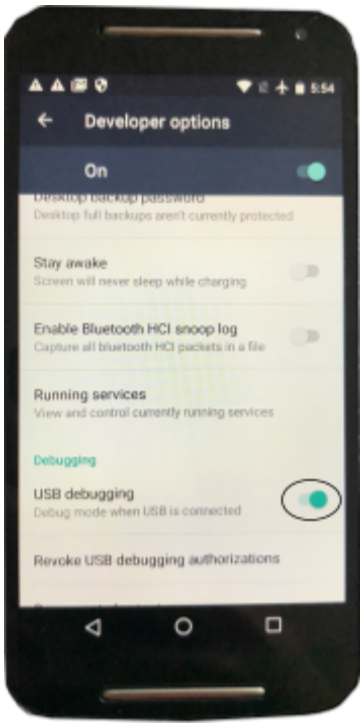
And choose your extracted folder.





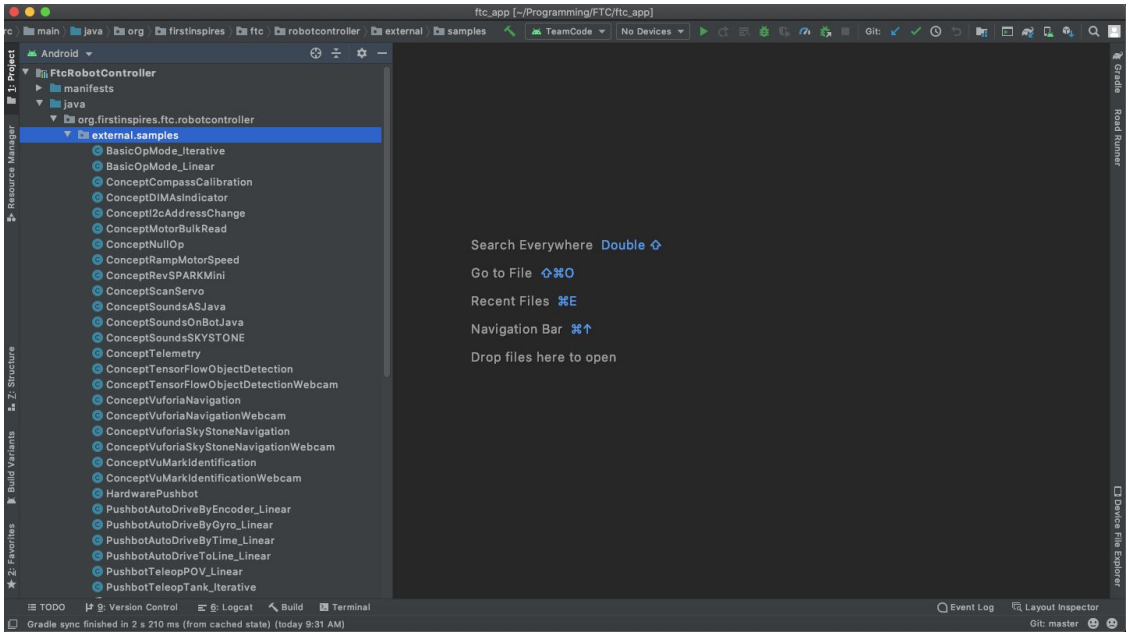
<- Once everything is set up, it should look like this

Next, you have to enable USB debugging for both (or only one if you are using a Control Hub) of your phones. To do that, go to settings -> developer options -> enable USB debugging



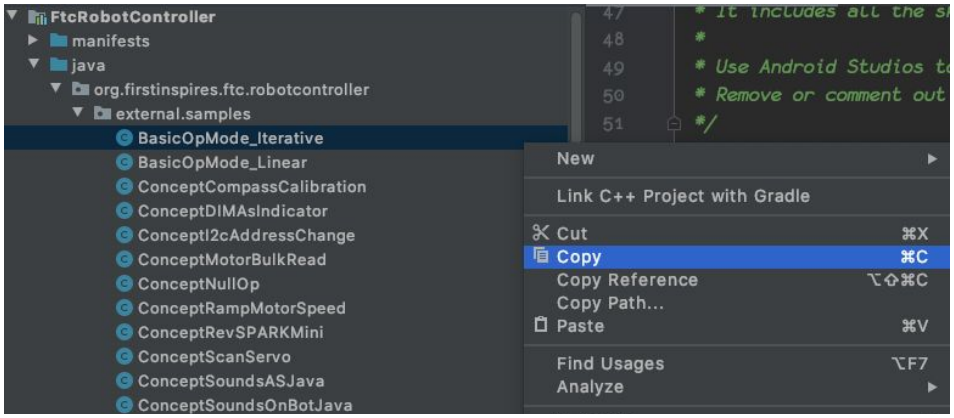
Now you're ready to use Android Studio!

On the left side, you can see the project structure of the FTC SDK. In FtcRobotController -> java -> external.samples, you can find all the example codes.

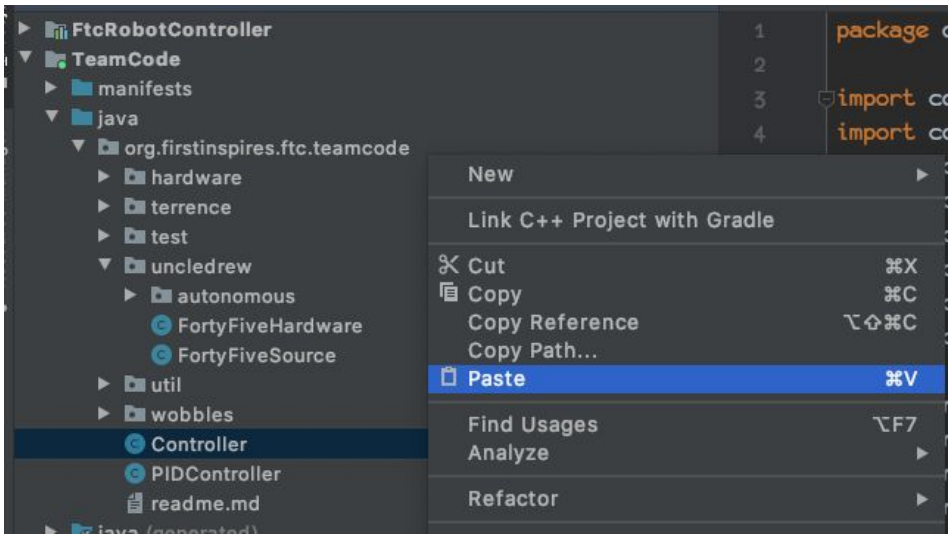


Something to take note of is that in the sample code, you notice that it says @Disabled:

This means that this OpMode will not show on the Driver Station App. To use the sample codes, you should copy the sample you want to use and then paste it into TeamCode -> java -> org.firstinspires.ftc.teamcode



Paste it here (note: we already have multiple files because we've been using Android studio for some time):



To deploy code, you either have to connect your computer to your Control Hub or your Robot Controller phone. If you're using a Control Hub, make sure that you are connecting your computer to it through the USB-C port.

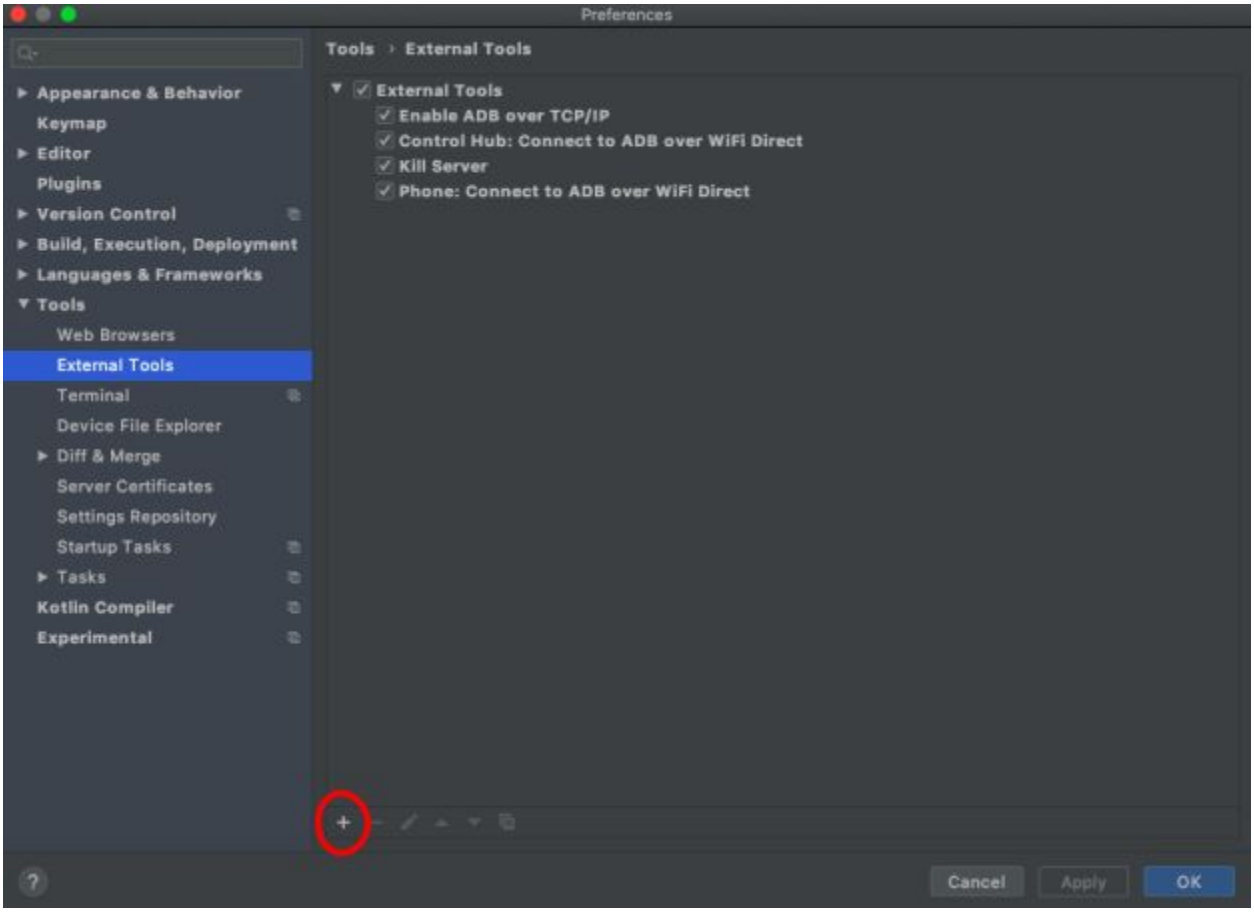
At the top of the screen, you should choose your device, and then you should be good to deploy code (if an "application installation failed" shows up, just click "ok")!

But wait...

We're in the 21st century, the wireless age. And we need wires to deploy code??? Unacceptable!

Here's how to deploy code *wirelessly*:

First, go to settings. On a Mac, it can be accessible by pressing Cmd + , while on Windows you can click "File" -> "Settings".



Then, go to tools -> external tools, and for each of the following tools below, press the '+' button and enter the information:

Name: Enable ADB over TCP/IP
Program: $ModuleSdkPath$/platform-tools/adb
Arguments: tcpip 5555
Working Directory: $ProjectFileDir

Name: Phone: Connect to ADB over WiFi Direct
Program: $ModuleSdkPath$/platform-tools/adb
Arguments: connect 192.168.49.1:5555
Working Directory: $ProjectFileDir

Name: Control Hub: Connect to ADB over WiFi Direct
Program: $ModuleSdkPath$/platform-tools/adb
Arguments: connect 192.168.43.1:5555
Working Directory: $ProjectFileDir

To deploy code wirelessly, follow these steps after each time your Android device reboots (on our experience, you only have to do this once if you're using the control hub):

1) Connect the Robot Controller phone or the Control Hub to your computer
2) Make sure that a file (e.g. an OpMode) is open and you have your cursor in that window
3) Go to Tools -> External Tools -> Enable ADB over TCP/IP
4) Disconnect your Android device and connect your computer to it through WiFi Direct
5) Run the Robot Controller app if you are using a phone
6) If you are using a phone, go to Tools -> External Tools -> Phone: Connect to ADB over WiFi Direct. If you are using a Control Hub, go to Tools -> External Tools -> Control Hub: Connect to ADB over WiFi Direct

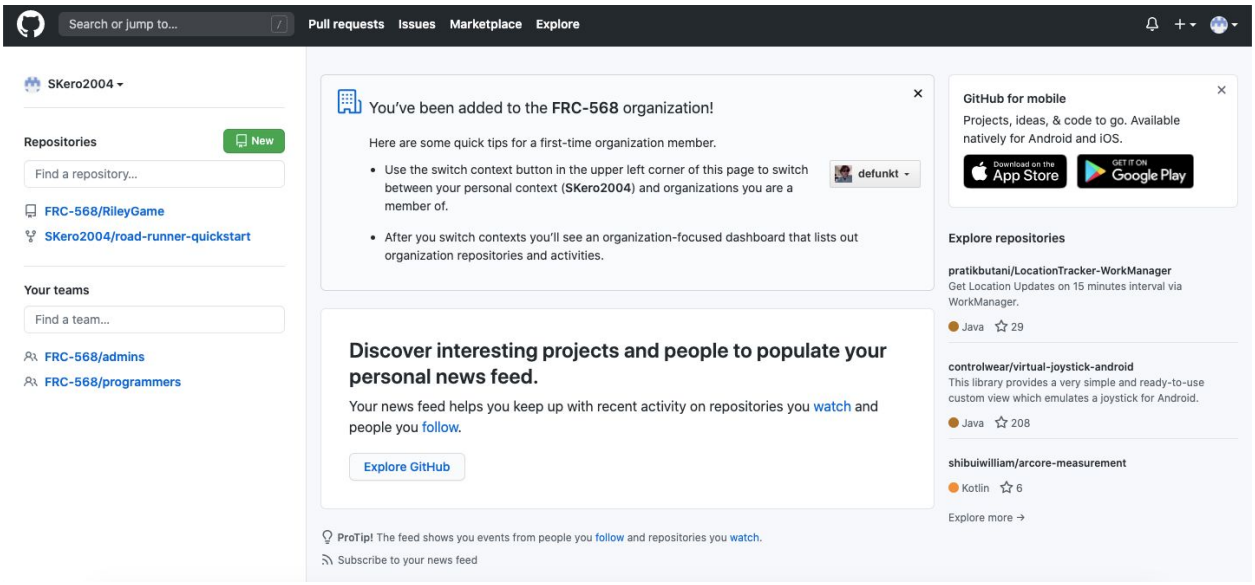
After all those steps, you are ready to deploy code wirelessly! Just press the  Run button and you're done!!!

Setting up Version Control:

There's another problem with Android Studio right now (we promise that all these setups will be worth it in the end!). Since the FTC SDK is downloaded to your local computer and not directly onto the Android Device, only *you* can see all the programming you did. This is a problem of course because you want your other team members to program the robot as well! Thankfully, fixing this problem is not hard, you just need to use something called **version control**.

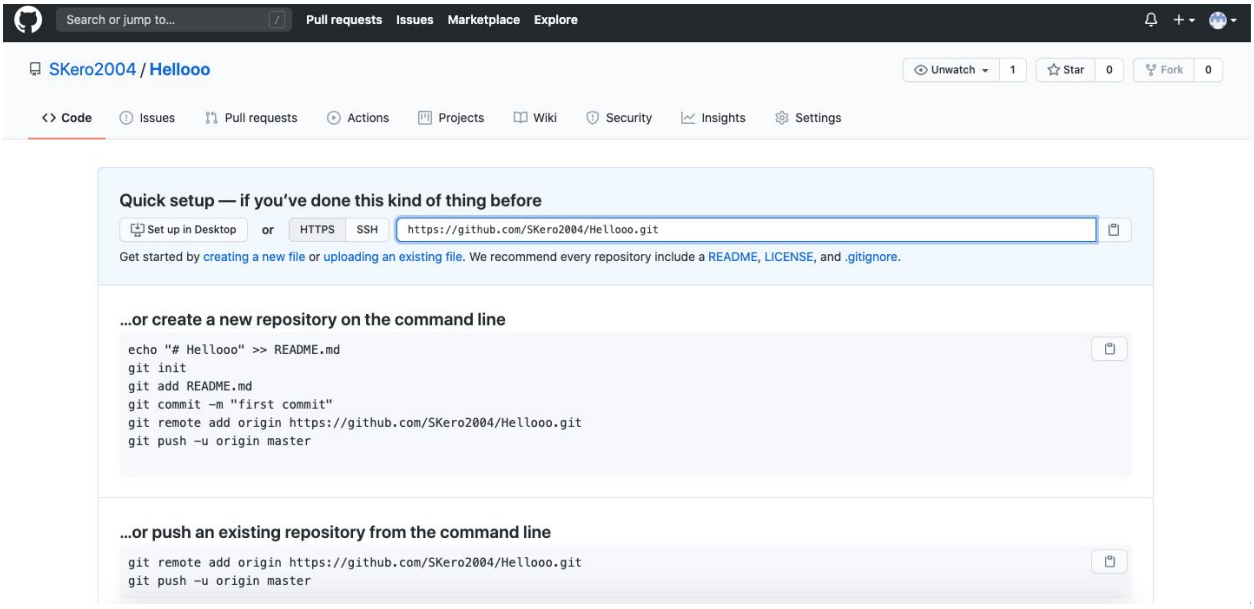
First download Git: https://git-scm.com/downloads. Next, get an account on GitHub for your team: https://github.com/.

Your GitHub account should look like this (except with fewer activities):



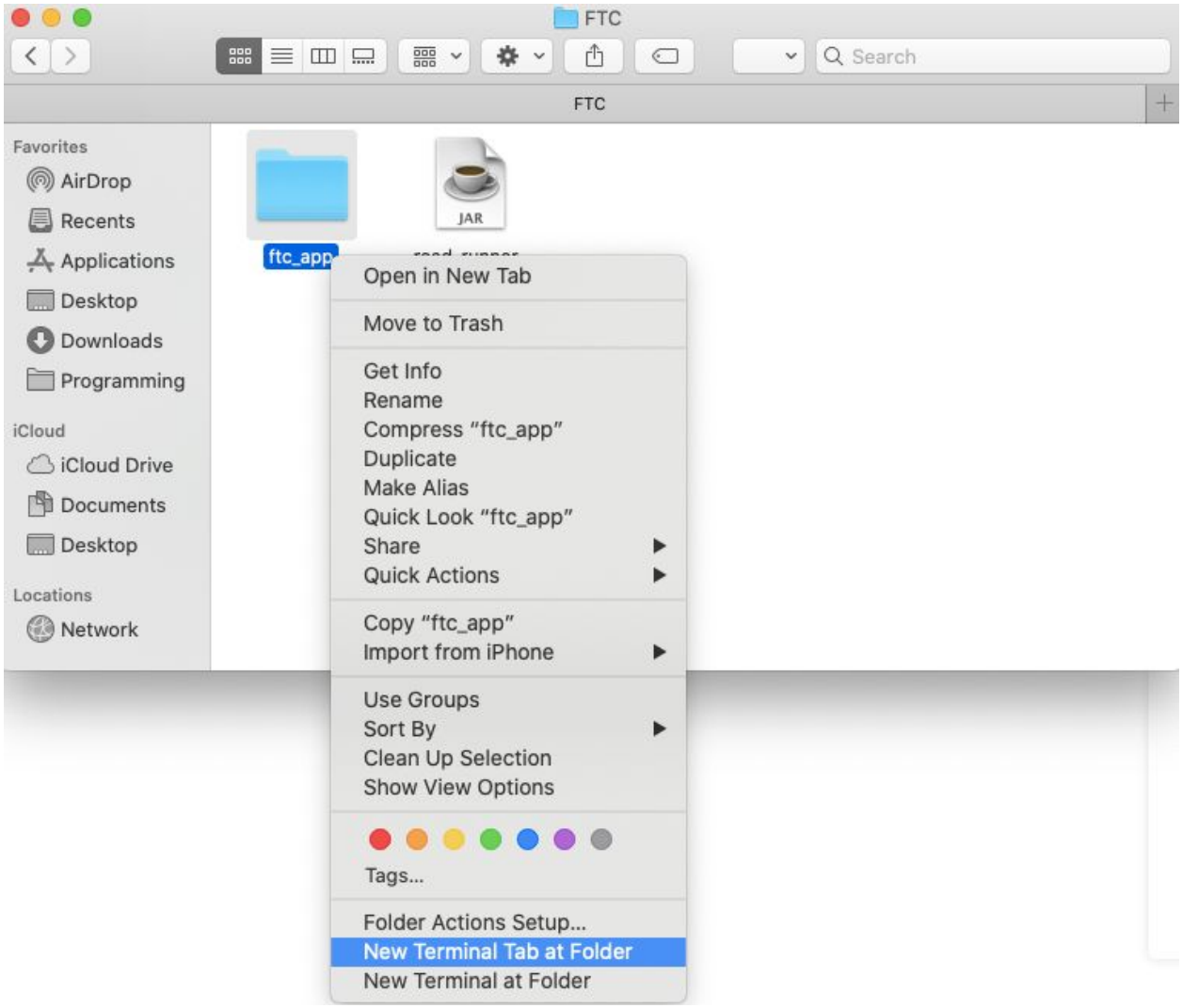
Click the  symbol to create a new repository (a place to put your code). Enter the name of your repository (it can be anything, something like ftc_app is sufficient), make it public, and create your repository.

Then you should see something that looks like this (except, of course, the name is different):



Take note of the URL. In this example, it is *https://github.com/SKero2004/Hellooo.git*

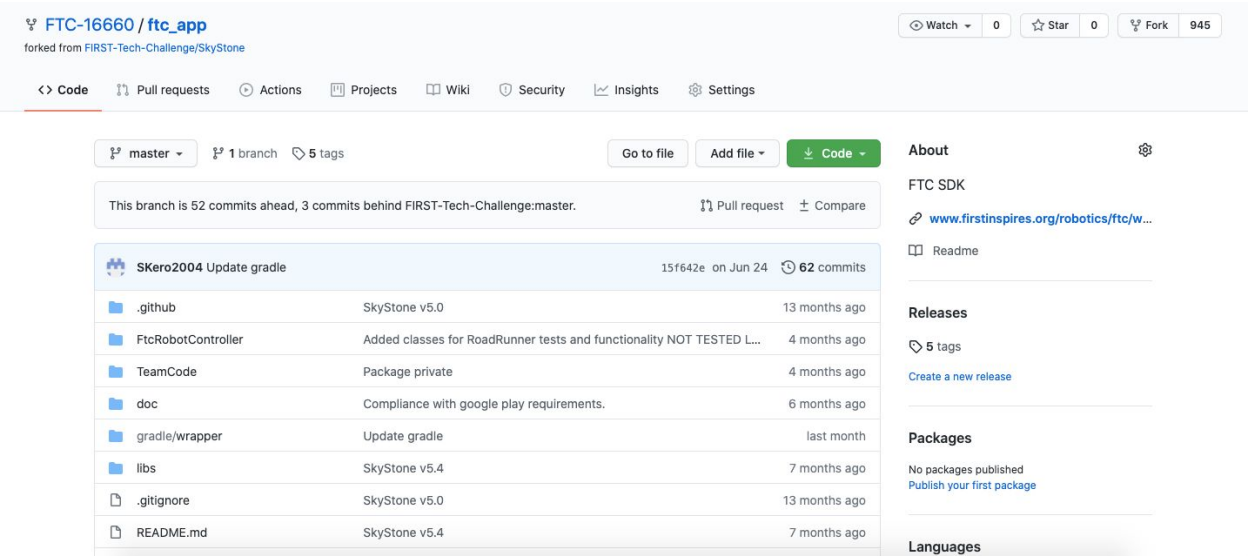
Now, go to your FTC SDK folder and right click -> New Terminal Tab at Folder



In your terminal, type in the following commands, replacing *remote repository URL* with the URL that you took note:

git init
git add .
git commit -m "First commit"
git remote add origin *remote repository URL*
git push -u origin master

Now, you should see all of your files in your new repository.



To make steps easier, we also recommend downloading a software known as Sourcetree (link: https://www.sourcetreeapp.com/) to facilitate version control. Run the installation process; it will ask you to create an Atlassian account and connect it to GitHub. You will be asked for an SSH key for now, but you can click cancel since you won't need one (and SSH keys are outside the scope of this book). Once the software is installed, go to File -> New -> New... -> Add Existing Local Repository and then choose your FTC SDK file.

Only a few more steps to go!

Another thing you want Sourcetree to do is to update your FTC SDK whenever FIRST updates it.

To do this, first click on this link: https://github.com/FIRST-Tech-Challenge. Then, click on the repository that you downloaded the Source Code from (the repository with current season name), and copy the URL (something like this: https://github.com/FIRST-Tech-Challenge/SkyStone).

Go back to Sourcetree and go to your repository. Click "Settings" on the top right, click "Remotes" -> Add. In "URL / path", paste the URL you copied, and in "Remote Name", type in "upstream" (it can be named whatever but it is conventional to name it this way). Then click "OK".

One last thing!

Remember, the whole point of version control is to allow your other team members to have access to the code you updated. So now, the “receiver” of your code (whoever that may be: your team member, mentor, etc.) should also install Sourcetree (connect it to your team’s GitHub account). Then, he/she should go to GitHub and copy the URL of your FTC SDK repository (for example, for us, it’s: https://github.com/FTC-16660/FtcRobotController).

Next, go to Sourcetree, click File -> New -> New... -> Clone from URL, and paste in the copied URL, choose where the FTC SDK will be, and type in a name for it.

Now you have all the version control set up!

How to use Sourcetree

You just set up everything for version control, but how do you use it? Let’s learn how version control works.

First, let’s say you added something new to your code:

```
// Wassuuupppp this is the changeeddddd line

package org.firstinspires.ftc.teamcode.uncledrew;

import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.hardware.bosch.BNO055IMU;
import com.qualcomm.robotcore.hardware.CRServo;
import com.qualcomm.robotcore.hardware.HardwareMap;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.DistanceSensor;
```

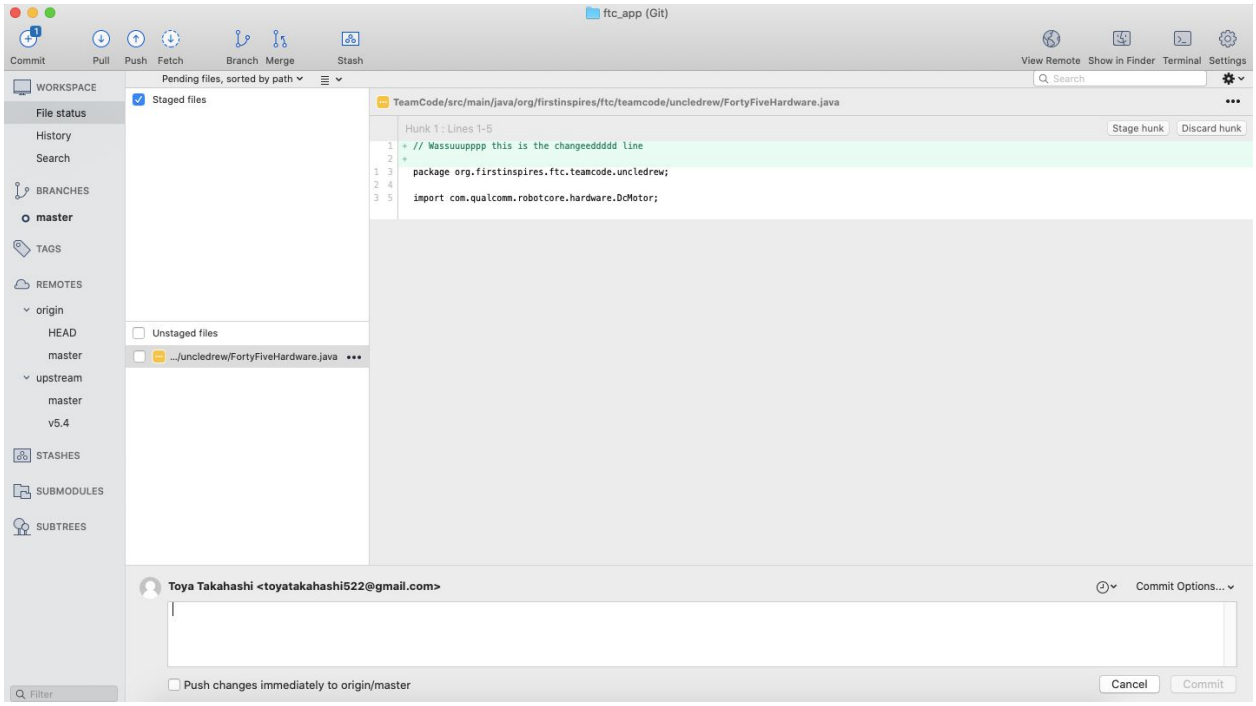
If you go to Sourcetree now, you should see the top left corner look like this:

This is Sourcetree’s way of saying “hey it seems like your code got updated! Do you want to commit it?”

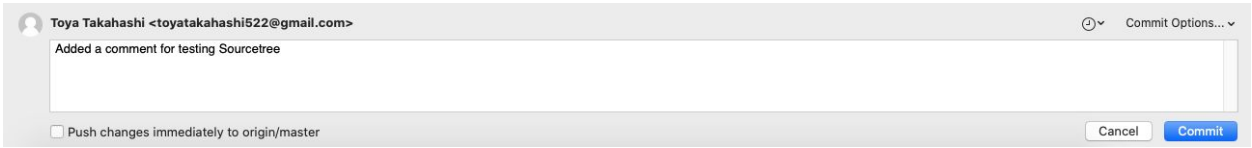
Wait what even is “commit”?

Commit is a way of saving the state of your code to your repository. When you click the 

button, you should be navigated to a screen that looks like this:

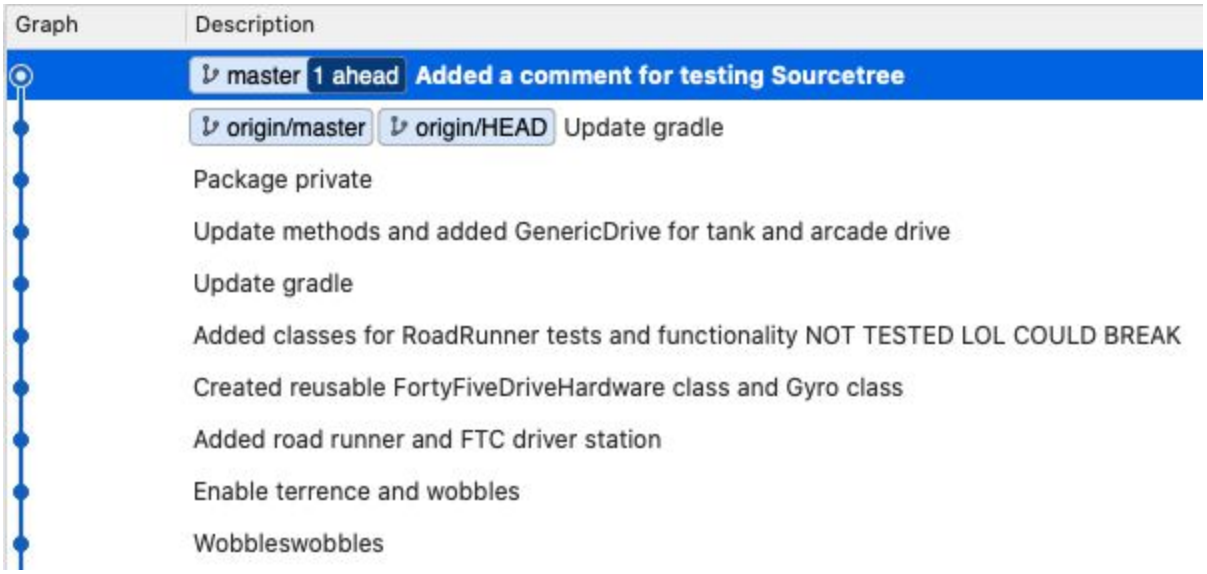


First, click the “Unstaged files” checkbox. Then, type in your commit message. A commit message tells you and your teammates (or even other people) about what the commit was about. For example, in our case, we can just type:



And click the bottom right “commit” button.

Now you should see something that looks like this:



This is your “tree” of your commits (hence SourceTREE). For you, it shouldn’t have too much activity but it should still say “master 1 ahead *blahblahblah*” and the *blahblahblah* is your commit message. So the next time you or anyone else sees that commit, you know what exactly changed.

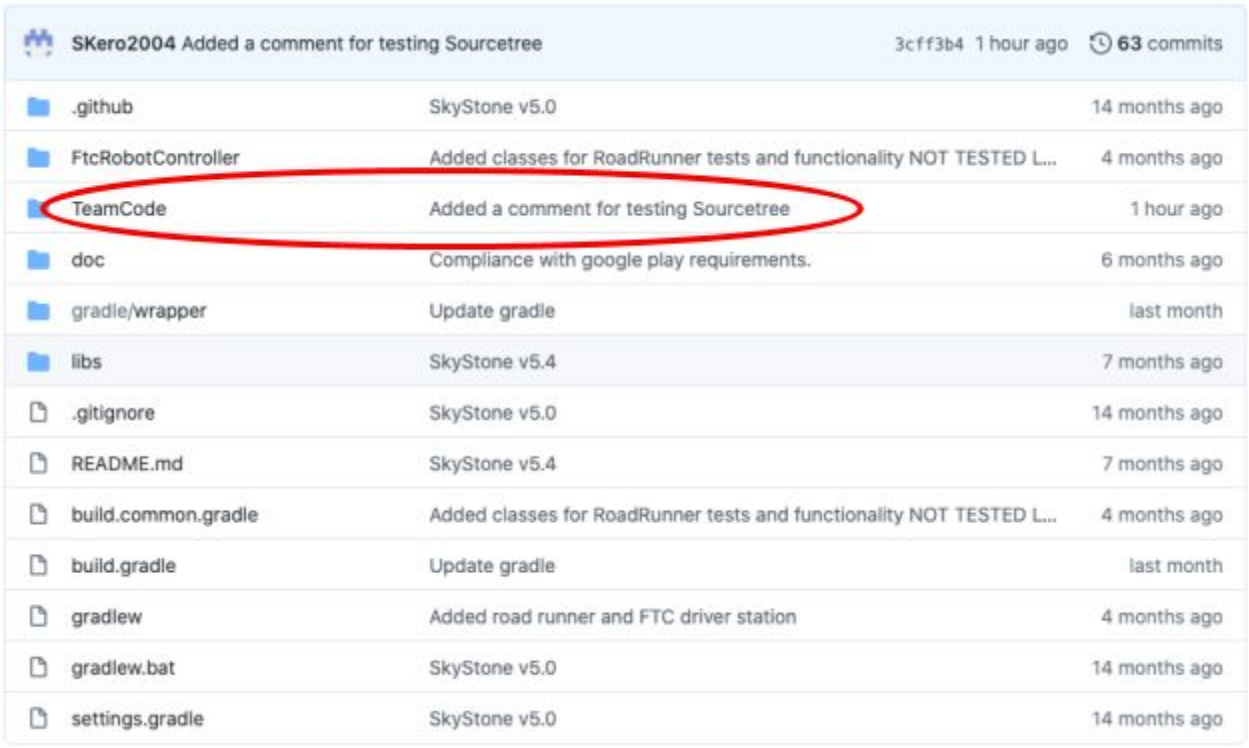
You probably also realized that the “push” button now looks like this:

This is telling you that there is one commit that you can “push” to your repository.

What is “push”?

In git, “push” means to send your local commits to your remote repository, in our case, GitHub.

When you press Push -> OK and go to your GitHub FTC SDK repository, you should now see that your most recent commit is shown:



Now, for your team member to receive the code, he/she must open Sourcetree, click:

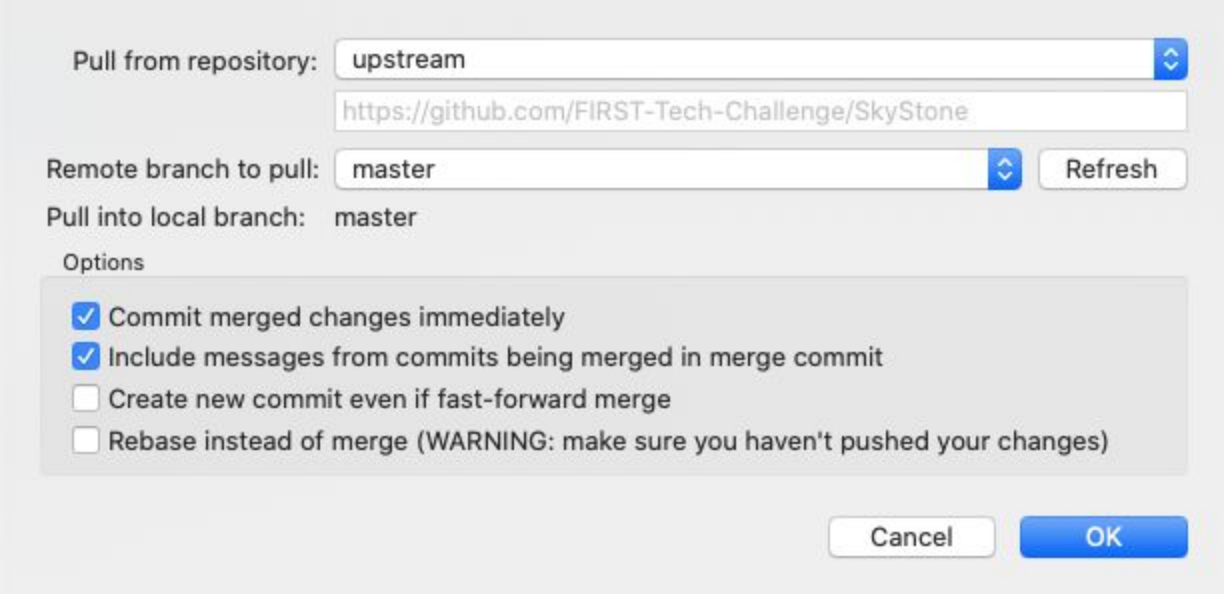
then:

-> OK.

Now, that person should have received all the code from you!

Finally, to get updates on the FTC SDK from FIRST, you first click “Fetch”, then “Pull” (as usual),

but then you have to change “Pull from repository” to “upstream” and select “Remote branch to pull” to “master” and click OK.



Now you have the updates from FIRST on your local computer. All you have to do now is update it to your remote repository. To do that, click “Push” -> “OK”.

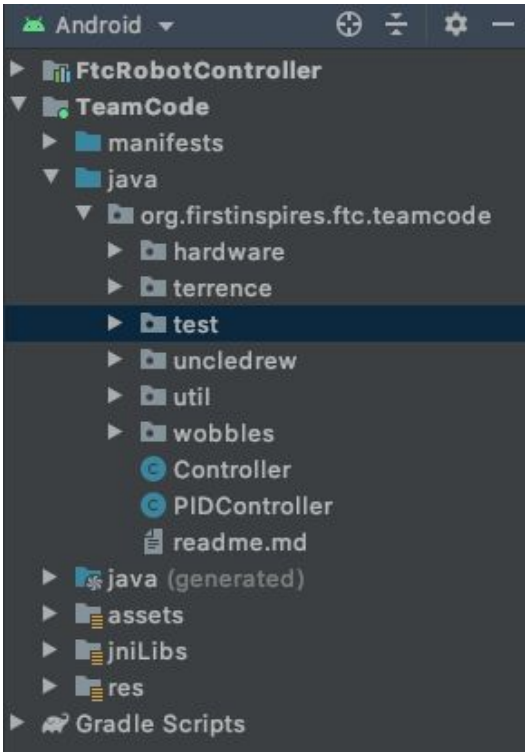
That’s all! Happy version controlling!

Organize Your Code:

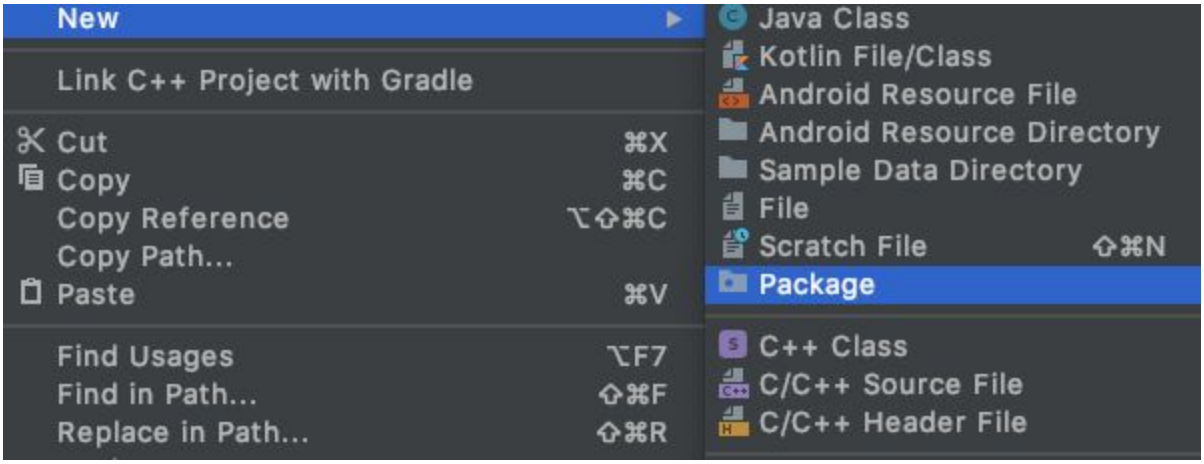
Hey, let’s get into the actual programming stuff because *that*’s the fun stuff! There are many ways to organize code but here we will show you the way we organize code in Android Studio.

Packages:

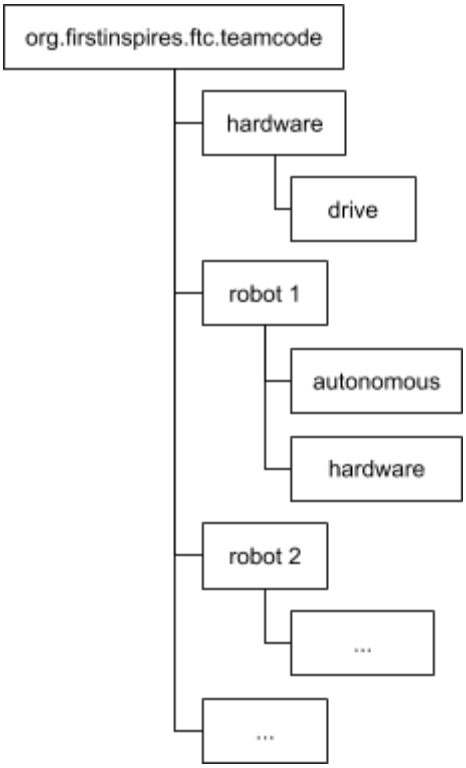
Here is how we organize our files:



Notice how we have many “packages” (the thingy that looks like a file merged with a camera) to organize our code. Packages are used to avoid naming conflicts and to create readable and easily readable code. To create a new package, simply right click on the folder you would like to create your new package, hover to New and click Package. (Note: make sure your package names don’t have spaces in between!).

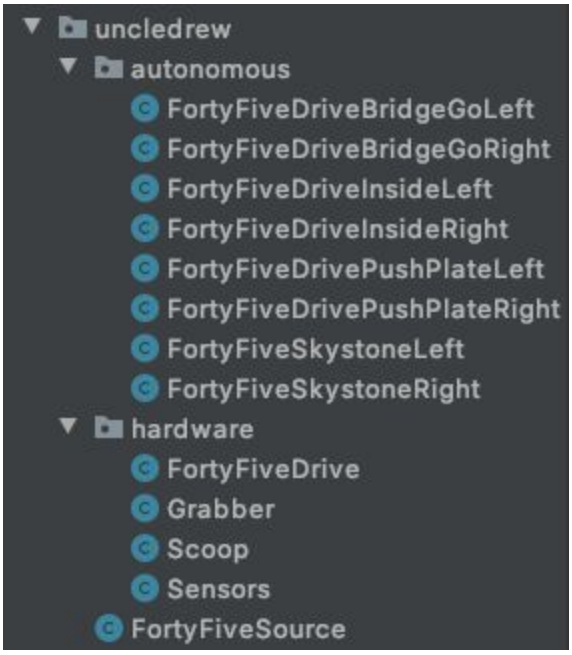


We recommend a package structure like this:



Inside “hardware”, we will put basic hardware that can be reused each year. You will also name each “robot” package with your robot name (e.g. uncledrew) and inside that package, you will include two packages, “autonomous” and “hardware”. Inside “autonomous” and “hardware”, you will put autonomous and hardware code for the robot respectively.

For example, let's look at our 2020-2021 robot, Uncle Drew:



Inside "autonomous", we have all the autonomous code. Inside "hardware", we have all the hardware code for the robot. Finally, we have our TeleOp code inside the "uncledrew" package.

Differential Drive

To get used to the new way of organizing code, let's start by making a simple teleop differential (robot controlled by 2 wheels on motors) drive code.

First, go to the "drive" package and create a new Java class called "DifferentialDrive":

```
package org.firstinspires.ftc.teamcode.hardware.drive;

public class DifferentialDrive {
}
```

At the top of the code, it says "package org.firstinspires.ftc.teamcode.hardware". This tells Java which package the file belongs to.
Then, add the following lines of code:

```
private DcMotor leftDrive;
private DcMotor rightDrive;
```

What this does is it creates DcMotor objects called “leftDrive” and “rightDrive” (same in OnBot Java).

But wait… what’s that “private” thing at the beginning?

According to W3Schools, “[t]he private keyword is an access modifier used for attributes, methods and constructors, making them only accessible within the declared class”, which is a very complicated way of saying “you can only access/find the variable/object in that same class”. For example, since you declared “leftDrive” as private, you will only be able to modify and read the value of “leftDrive” in the DifferentialDrive class. On the other hand, without the private modifier, “leftDrive” will be accessible from classes in the same package (this is called package-private).

Why do we make variables/objects private? The main reason is to prevent errors from occurring. For example, in our case, the value of “leftDrive” can be changed from any classes in the “org.firstinspires.ftc.teamcode.hardware” package. This isn’t ideal because it means any other unrelated classes can control the motors and can break or crash the code. To prevent such errors, we put the private modifier whenever possible.

Next, we type in the following code:

```
public DifferentialDrive(HardwareMap hwMap) {

    // Define and initialize motors

    leftDrive = hwMap.get(DcMotor.class,  deviceName: "left_drive");
    rightDrive = hwMap.get(DcMotor.class,  deviceName: "right_drive");
    leftDrive.setDirection(DcMotor.Direction.FORWARD);
    rightDrive.setDirection(DcMotor.Direction.REVERSE);

    leftDrive.setPower(0);
    rightDrive.setPower(0);

    leftDrive.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
    rightDrive.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);

}
```

*Set the DcMotor directions to whatever that works for your robot

This function is called a constructor function of the class and is called every time a new object of this class is created (if you don't entirely understand this concept, that's ok, we will talk more about it later). If you are familiar with OnBot Java, you should know that this piece of code sets up the motors. You can change the deviceName to whatever that suits you; just remember to change it in the configuration in your DS phone. Also, we will talk more about "RUN_WITHOUT_ENCODER" later in the "Encoders" section.

By this point, you probably realized something is going on at the top of your code…:

```
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.HardwareMap;
```

(if this didn't appear at the top of your file, then manually add them, but the process *should* be automatic)

These lines are importing code into your file, making the classes DcMotor and HardwareMap accessible on your DifferentialDrive file. And guess what "com.qualcomm.robotcore.hardware.DcMotor" is… Java Packages! Mind blown! (don't worry if you're not, you'll get it as you get through this book).

Now type this:

```
public void drive(double leftStickY, double rightStickX) {



}
```

You should realize that this is a function/method belonging to the DifferentialDrive class. In this method, we will include the necessary code for the robot to move during TeleOp with the two parameters: leftStickY and rightStickX.

Also, notice how this one says public and not private. This is because you want this method to be accessible from *anywhere* in your code (e.g. your TeleOp code)

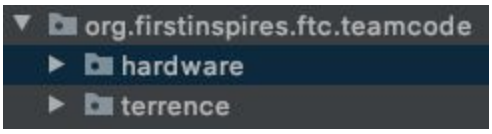
Inside the “drive” method, include:

```
double drive = -leftStickY;
double turn = rightStickX;

double left = drive + turn;
double right = drive - turn;

double max = Math.max(Math.abs(left), Math.abs(right));
if (max > 1.0) {
    left /= max;
    right /= max;
}

leftDrive.setPower(left);
rightDrive.setPower(right);
```

This is a very simple arcade drive code. The left joystick controls the forward-backward motion of the robot and the right joystick controls the turning motion of the robot. The “left” variable value increases and the “right” variable value decreases as “turn” (right joystick X input) increases. The “max” variable finds the greater value of the absolute value of “left” and “right” and scales them down. Finally, in the last two lines of code, the motors are driven.

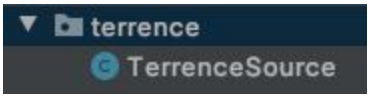
And… that's it for this class! Now let's create the TeleOp code.

Inside "org.firstinspires.ftc.teamcode", add another package with your robot name:



(our differential drive robot is named Terrence)

Inside that package, create another Java class. This will be your TeleOp code:



Inside your file, type:

```
@TeleOp(name="Terrence: TeLeop", group="Terrence")
// @Disabled
public class TerrenceSource extends OpMode {

    private DifferentialDrive drive;

    @Override
    public void init() {


    }

    @Override
    public void loop() {


    }

}
```

(whenever you don't need this to show up on your DS anymore, just uncomment @Disabled)

The private DifferentialDrive is *the* differential drive class you just made.

To prove that, at the beginning of the file, there is:

```
import org.firstinspires.ftc.teamcode.hardware.drive.DifferentialDrive;
```

Which imports the DifferentialDrive class just like you import anything else (e.g. DcMotor). You need to import them because this file is in a different package. If this file was in the "org.firstinspires.ftc.teamcode.hardware.drive" package, there would have been no need to import.

Now all you have to write is:

```
@Override
public void init() {

    drive = new DifferentialDrive(hardwareMap);

}

@Override
public void loop() {

    drive.drive(gamepad1.left_stick_y, gamepad1.right_stick_x);

}
```

That's all? Yep.

Looks so simple and organized! This is why it is necessary to organize code. It makes the code much easier to read for yourself and your team members.

Why is it so simple? Because you put all the necessary calculations and hardware into your DifferentialDrive file. In init(), you typed "new DifferentialDrive(hardwareMap)" which calls the constructor function of DifferentialDrive, which sets up the DcMotors. The "drive.drive" function is also already defined in the DifferentialDrive class. All this combined makes a nice-looking piece of code.

Now, you're ready to use your DifferentialDrive TeleOp! Deploy the code onto your RC and test your code.

Using a Custom Controller Class:

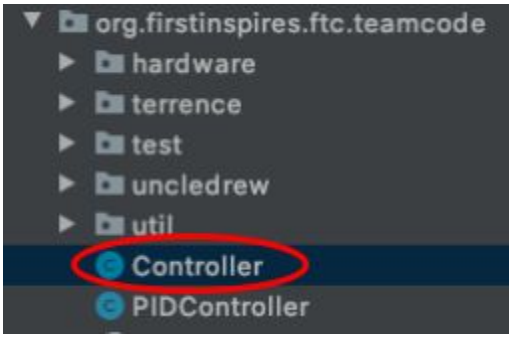
We also recommend using a custom controller class, rather than the predefined one.

```
drive.drive(gamepad1.left_stick_y, gamepad1.right_stick_x);
```

Here, you call gamepad1 to get the inputs. Although this is adequate for this purpose, gamepad1 (and 2) misses a key feature: the ability to tell whether the button was only clicked *once*.

This key feature is extremely useful when toggling modes or when making the robot do something after you hit a certain button. In the next section, we will see a specific example of this.

Inside "org.firstinspires.ftc.teamcode", add a new Java class called Controller.



Inside the Controller class, copy and paste the code from here:
https://github.com/FTC-16660/FtcRobotController/blob/master/TeamCode/src/main/java/org/firstinspires/ftc/teamcode/Controller.java

Instead of using the predefined "gamepad"s, you should use this class from now on.

Improving Your Differential Drive Class:

Let's make use of the new Controller class you just created!

Go to your TeleOp Java file:

```
@Override
public void init() {

    drive = new DifferentialDrive(hardwareMap);

}

@Override
public void loop() {

    drive.drive(gamepad1.left_stick_y, gamepad1.right_stick_x);

}
```

Notice how we are still using “gamepad1” for controlling the robot. Let’s use our new Controller class instead:

```
public class TerrenceSource extends OpMode {

    private DifferentialDrive drive;

    private Controller controller1;

    @Override
    public void init() {

        drive = new DifferentialDrive(hardwareMap);

        controller1 = new Controller(gamepad1);

    }

    @Override
    public void loop() {

        controller1.update();

        drive.drive(controller1.left_stick_y, controller1.right_stick_x);

    }

}
```

Here, after you declare “private Controller controller1;”, you create a new Controller object connected to the input of gamepad1 with “controller1 = new Controller(gamepad1);”. Now every time you want to use your controller, you should use the new controller1 object you created. This is why in drive.drive, we are passing controller1.left_stick_y and controller1.right_stick_x rather than those of gamepad1. In addition to that, we are also calling controller1.update() in the loop() function. This is necessary for the code to function correctly.

So what was the point of using controller1? Well... with just this code... nothing. So let’s use the new feature for the controller class: the once() functions.

The two common ways to control a differential drive robot are arcade drive and tank drive. Arcade drive is what you just programmed: left joystick-y for forward-backward motion, right joystick-x for turning. On the other hand, a robot with tank drive is controlled by the y-axis of both the left and the right joystick.

Since currently, the only way to control our differential drive is by arcade drive, let’s change it so that we can press a button to toggle arcade drive and tank drive.

Go to the DifferentialDrive class and add the line

```
private boolean isArcade = true;
```

Before the constructor function. Then, let’s add a new function to the class that will toggle “isArcade” back and forth:

```
// Toggle arcade and tank drive
public void toggleArcade(boolean button) {

    if (button)
        isArcade = !isArcade;

}
```

Pretty simple. When “button” is true (button is pressed), “isArcade” toggles its value;

Let's also modify our "drive" method as well:

```
// Arcade and tank drive
public void drive(double leftStickY, double rightStickY, double rightStickX) {

    double left;
    double right;

    if (isArcade) {

        // Arcade drive

        double drive = -leftStickY;
        double turn = rightStickX;

        left = drive + turn;
        right = drive - turn;

        double max = Math.max(Math.abs(left), Math.abs(right));
        if (max > 1.0) {
            left /= max;
            right /= max;
        }

    } else {

        // Tank drive
        left = -leftStickY;
        right = -rightStickY;

    }

    leftDrive.setPower(left);
    rightDrive.setPower(right);

}
```

Well, many things changed here. The first thing is the parameters. It now takes three inputs rather than just two. This is since tank drive requires the right joystick-y value as well. Then, rather than simply assigning the values to "left" and "right", we simply declare "left" and "right" with no values. This is to make the code look nice and clean: since both arcade drive and tank drive require these variables, we can just declare these variables beforehand.

Then, we have an if-else statement. Inside the "if" part, we have our arcade drive code, and inside the "else" part, we have our tank drive code. So if "isArcade" is true, it runs arcade drive code, and if it is false, it runs tank drive code.

Finally, the powers of the motors are set at the end, no matter arcade drive or tank drive.

Now let's modify our TeleOp code again! (one last time I promise!)

```
@Override
public void loop() {

    controller1.update();

    drive.toggleArcade(controller1.backOnce());
    drive.drive(controller1.left_stick_y, controller1.right_stick_y, controller1.right_stick_x);

}
```

A very simple modification. Since the drive function has three parameters now, we add in the new “controller1.right_stick_y” variable.

And here’s the important line: “drive.toggleArcade(controller1.backOnce())”. This is where we make use of the new Controller class. The method “controller1.backOnce()” returns true ONLY the moment you press the button (in this case, the back button on the controller). On the other hand, just typing “gamepad1.back” returns true CONTINUOUSLY when you press the back button. This is very important because it means that during every single loop in the loop() function, the “isArcade” property of “drive” is getting toggled and therefore you only have a 50-50 chance of actually toggling the “isArcade” property correctly (when releasing the button).

Because we are using the “backOnce()” method instead, when we run the code and control our robot, we can reliably switch between arcade drive and tank drive anytime by pressing the back button on our controller!

Getting Sensor Inputs

The next step to improving your robot is using sensors! This can significantly improve the reliability of your robot - mainly during autonomous but during TeleOp as well!

Color/Distance Sensors

Let’s first talk about color/distance sensors. Color/distance sensors can... well, sense colors and find the distance between the sensor and the object in front of it. In our Skystone robot, we used a color sensor to detect whether the object in front of the robot

is a Skystone (a block with a black picture in front; worth more points) or a normal stone (a yellow block). We have also seen other teams using color sensors to stop the robot in the correct place during autonomous.

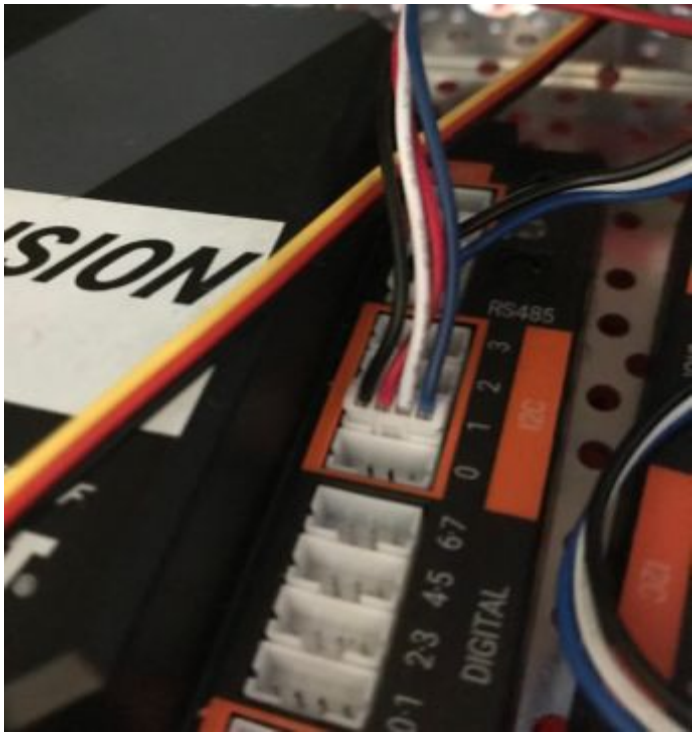
We recommend using the rev robotics color sensor which also has a built-in distance sensor as well: https://www.revrobotics.com/rev-31-1557/.

Let's use a color sensor. Create a new TeleOp file somewhere that includes this piece of code:

```
@TeleOp(name = "ColorDistanceSensorTest")
public class ColorDistanceSensorTest extends OpMode {

    private ColorSensor colorSensor;
    private DistanceSensor distanceSensor;

    @Override
    public void init() {

        colorSensor = hardwareMap.get(ColorSensor.class, deviceName: "color");
        distanceSensor = hardwareMap.get(DistanceSensor.class, deviceName: "color");

    }

    @Override
    public void loop() {

        telemetry.addData( caption: "Red: ", colorSensor.red());
        telemetry.addData( caption: "Green: ", colorSensor.green());
        telemetry.addData( caption: "Blue: ", colorSensor.blue());
        telemetry.addData( caption: "DistanceCM: ", distanceSensor.getDistance(DistanceUnit.CM));
        telemetry.addData( caption: "DistanceINCH: ", distanceSensor.getDistance(DistanceUnit.INCH));

    }

}
```

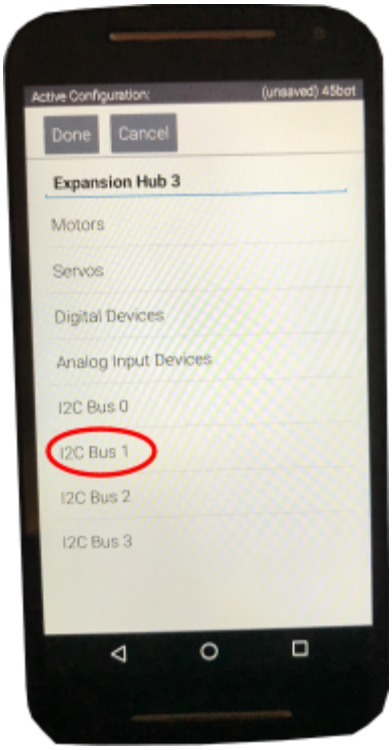
The code here is simple. It creates a color sensor and a distance sensor object with the configuration name "color". Then, in loop(), it adds the data corresponding to the caption to the telemetry.

Now, connect the color/distance sensor to the control hub/expansion hub's I2C port:

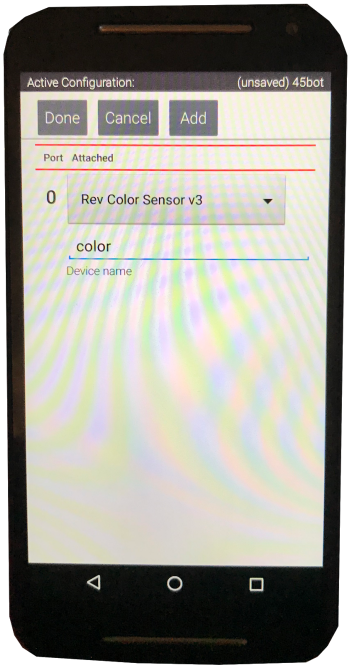


It doesn't matter which port you put into as long as the configuration is correct on the DS.

In your DS, go to your hub configuration and click on the I2C port you plugged the color/distance sensor into (in the picture, it's bus 1):



Then, configure your color/distance sensor as Rev Color Sensor v3 (at least that is what we recommend) and set its name (if you remember, we put in the name “color” in the code).



Now, if you run “ColorDistanceSensorTest”, you should be able to see the color and the distance in inches and centimeters.

When you run the code, you would probably realize that the color values change depending on the distance of the object from the sensor. Although this *could* be an issue if you are trying to detect the exact color of an object, usually in FTC games, you only have to distinguish between different colors. For example, if you want to distinguish between red and blue, you can simply use the code:

```
if (colorSensor.red() > colorSensor.blue()) {

    // Do stuff if red

} else {

    // Do stuff if blue

}
```

This works reliably because the color red is redder than blue (has a higher red value than blue).

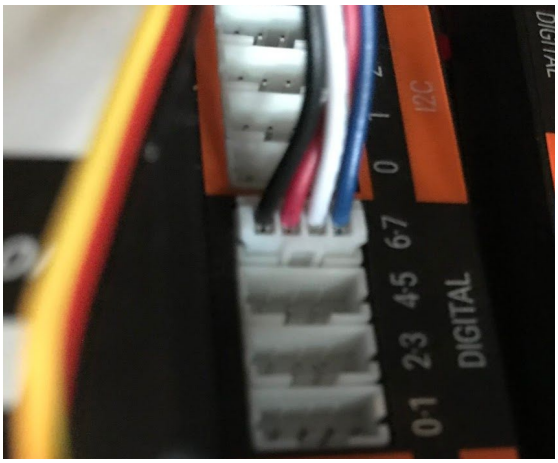
Touch Sensors

Another type of sensor that you might find useful is the touch sensor. A touch sensor, as the name suggests, detects a touch. This can be useful for user input, pushing buttons, resetting values when a part of the robot touches the touch sensor, etc. Touch sensors can be bought here: https://www.revrobotics.com/rev-31-1425/.

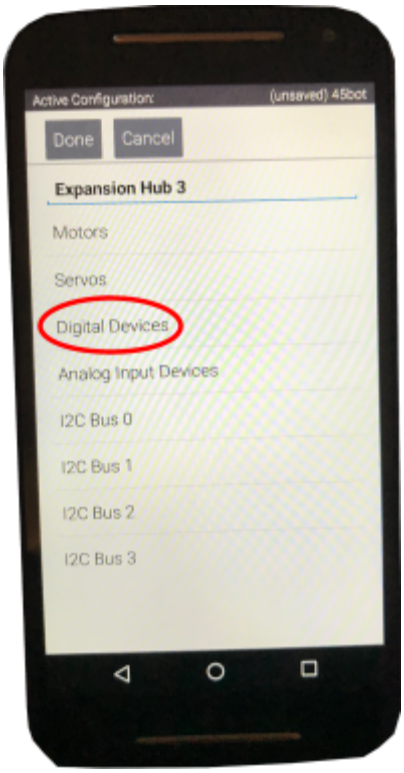
Touch sensors are easy to use. For testing, use the following code:

```
@TeleOp(name = "TouchTest")
public class TouchTest extends OpMode {

    private TouchSensor touchSensor;

    @Override
    public void init() {

        touchSensor = hardwareMap.get(TouchSensor.class, deviceName: "touch");

    }

    @Override
    public void loop() {

        telemetry.addData( caption: "isPressed:", touchSensor.isPressed());

    }

}
```

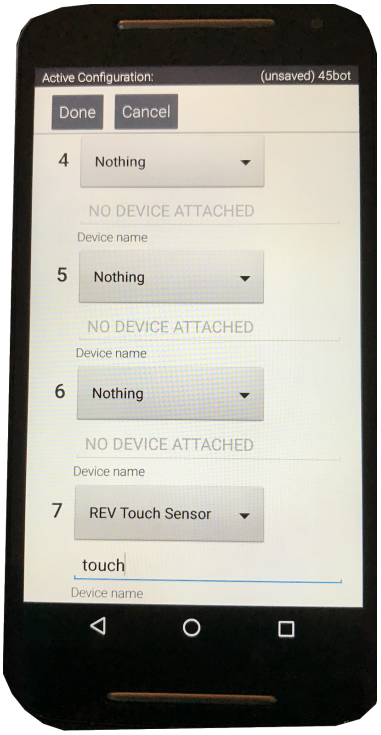
The configuration steps are also similar. However, a touch sensor is a digital device:



So, in your DS, you go to Digital Devices:



Then type the configuration name (“touch” in our case) along with the device name (we recommended the REV Touch Sensor):



And that's it! Touch sensors are fairly easy to use. Test it out using the TouchTest TeleOp code!

Gyro Sensors

Gyro sensors are usually necessary for a consistent autonomous code. They sense angular velocity, meaning they calculate the rate of speed you are rotating, but more importantly, they give you the angle that you are facing towards. For accurate turning, you *need* a gyro sensor. If you are using a holonomic drive (discussed later) with PID Control (also discussed later), gyro sensors are also used in TeleOp for accurate driving.

In this section, we will only cover the basics of using a gyro; we will talk about applying this code in future sections.

First, create a Gyro class in the package org.firstinspires.ftc.teamcode.hardware. Before the constructor, type in the following code:

```
private BNO055IMU imu;

private double globalAngle;
private Orientation lastAngles;
```

The BNO055IMU (inertial measurement unit) is built-in for both the REV expansion hub and the REV control hub, and functions similarly to a gyro sensor. The “globalAngle” and “lastAngles” variables will carry data about the robot angle.

Next, type in the following constructor function:

```
public Gyro(HardwareMap hwMap) {

    // IMU parameters
    BNO055IMU.Parameters parameters = new BNO055IMU.Parameters();
    parameters.mode = BNO055IMU.SensorMode.IMU;
    parameters.angleUnit = BNO055IMU.AngleUnit.DEGREES;
    parameters.accelUnit = BNO055IMU.AccelUnit.METERS_PERSEC_PERSEC;
    parameters.loggingEnabled = false;

    // Define and initialize IMU
    imu = hwMap.get(BNO055IMU.class, deviceName: "imu");
    imu.initialize(parameters);

    // Calibrate gyro
    while (!imu.isGyroCalibrated()) {}

    reset();

}
```

Okay let’s talk about this constructor a little. First:

```
    // IMU parameters
    BNO055IMU.Parameters parameters = new BNO055IMU.Parameters();
    parameters.mode = BNO055IMU.SensorMode.IMU;
    parameters.angleUnit = BNO055IMU.AngleUnit.DEGREES;
    parameters.accelUnit = BNO055IMU.AccelUnit.METERS_PERSEC_PERSEC;
    parameters.loggingEnabled = false;
```

These are the settings of the imu. If you want the angles to be returned in radians, you can use AngleUnit.RADIANS instead.

Next:

```
// Define and initialize IMU
imu = hwMap.get(BNO055IMU.class, deviceName: "imu");
imu.initialize(parameters);
```

This gets the imu from the REV hub (more on this later) and pushes the setting values you just configured in the lines before.

Finally:

```
// Calibrate gyro
while (!imu.isGyroCalibrated()) {}

reset();
```

As the comment says, the first line allows time for the gyro to be calibrated for accurate measurement. Also, don't worry if Android Studio doesn't like the reset() function! We'll just add that now:

```
public void reset() {

    lastAngles = imu.getAngularOrientation(AxesReference.INTRINSIC, AxesOrder.ZYX, AngleUnit.DEGREES);
    globalAngle = 0;

}
```

When you call the reset() function, the globalAngle (the angle that the robot is facing) gets reset (equals zero) and the lastAngles (the way we get our imu value).

To get the angle the robot is facing, you need to add this piece of code:

```
public double getAngleDegrees() {

    Orientation angles = imu.getAngularOrientation(AxesReference.INTRINSIC, AxesOrder.ZYX, AngleUnit.DEGREES);

    double deltaAngle = angles.firstAngle - lastAngles.firstAngle;

    if (deltaAngle < -180)
        deltaAngle += 360;
    else if (deltaAngle > 180)
        deltaAngle -= 360;

    globalAngle += deltaAngle;

    lastAngles = angles;

    return globalAngle;

}
```

And this, if you want it in radians as well:

```
public double getAngleRadians() { return Math.toRadians(getAngleDegrees()); }
```

Again, let's walk through the code.

First, you may realize that the first line is actually the exact same as the way we got our lastAngles value in the reset() function:

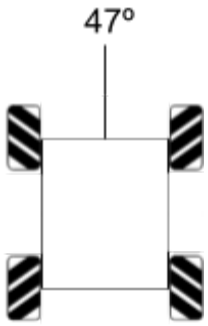
```
Orientation angles = imu.getAngularOrientation(AxesReference.INTRINSIC, AxesOrder.ZYX, AngleUnit.DEGREES);
```

This line assigns the imu value of the robot angle to the variable angles.

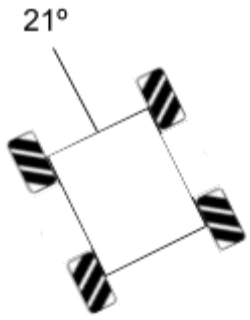
In the next line, we calculate the difference between the "angles" variable and the "lastAngles" variable to get the "deltaAngle" variable:

```
double deltaAngle = angles.firstAngle - lastAngles.firstAngle;
```

For example, let's say that your robot is facing upwards like so and the imu heading (lastAngles.firstAngle) tells you that it's at 47°:



Then, let's say you rotated by certain degrees and now it says it's at 21° (angles.firstAngle):



How much did the robot turn? 21 - 47 = -26. 26° to the left. This is what deltaAngle signifies.

Then you have this if-statement block:

```
if (deltaAngle < -180)
    deltaAngle += 360;
else if (deltaAngle > 180)
    deltaAngle -= 360;
```

This basically makes deltaAngle in the range [-180, 180]. For example, if deltaAngle was -230°, the first if statement will run and change deltaAngle to 130°. Both angles are equivalent since any angle +- 360 equals the same angle.

Finally you have these three lines of code:

```
globalAngle += deltaAngle;

lastAngles = angles;

return globalAngle;
```

The first line of code adds deltaAngle to globalAngle. Since globalAngle tracks the angle of the robot, adding deltaAngle (how much your robot turned by) to it will update the robot angle. In the second line of code, we set lastAngles to angles. This is so that the next time this function runs, you can get the previous angle to calculate how much your robot turned since then with angles.firstAngle - lastAngles.firstAngle. Finally, you return globalAngle, which is the angle of the direction of your robot since the last reset of the gyro!

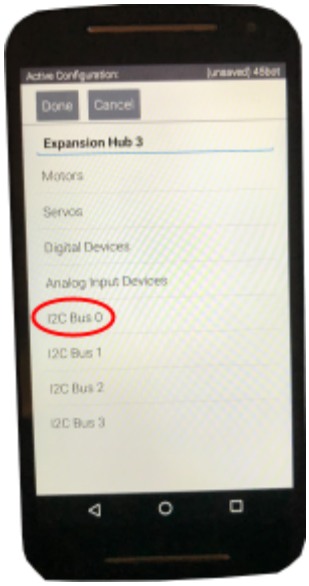
That was the logistics of how we calculate the current angle. Let's get the numbers now.

Create a new TeleOp file and add this piece of code:

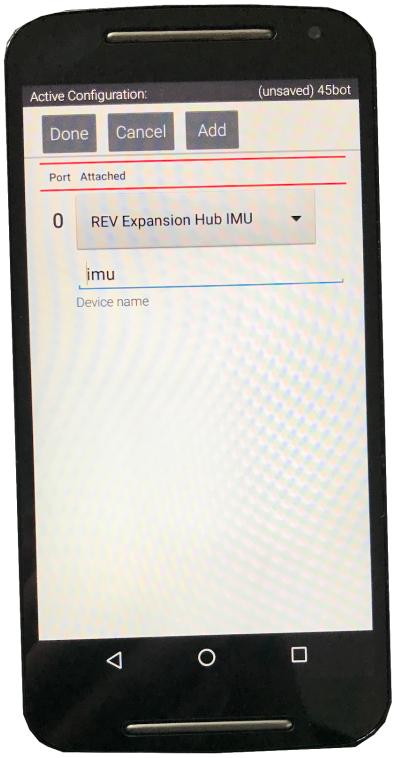
```
@TeleOp(name = "Test Gyro")
public class TestGyro extends OpMode {

    private Gyro gyro;

    @Override
    public void init() {

        gyro = new Gyro(hardwareMap);

    }

    @Override
    public void loop() {

        telemetry.addData( caption: "angle: ", gyro.getAngleDegrees());

    }

}
```

The code is pretty self-explanatory; it creates a new Gyro object and adds the angle to the telemetry.

In your DS robot configuration, go to I2C Bus 0:



Then select REV Expansion Hub IMU and name it “imu”.



Now your imu should be ready to use!

Try running your TestGyro code in your DS and turn the REV Hub to see the angle value change!

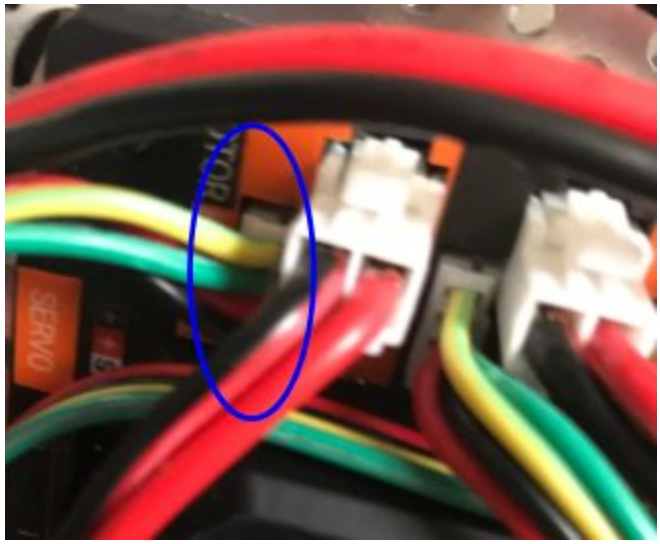
Using Encoders

Another sensor that is extremely useful for autonomous is an encoder. An encoder is a device that returns feedback of how much the motor shaft has rotated. This is useful because it allows us to precisely move our robot for a certain amount of distance.

Why not simply use timers? Because making a robot move using timers is not precise AT ALL. For example, let’s say you programmed a robot to move forward for 3 seconds. Run that code once, then another time after you play around with the robot for a while. You’ll realize that the robot will move a smaller distance the second time. Why? Because you already depleted the battery power when you played around with the robot, therefore providing less power to the motors. And this is why encoders are *necessary* for a good autonomous program.

Luckily, most FTC legal motors have a built-in encoder. Just need an additional cable. For example, for a NeveRest Motor (https://www.andymark.com/products/neverest-classic-40-gearmotor), you just need to attach a REV Hub Encoder Cable (https://www.andymark.com/products/encoder-cable-for-neverest-motor-to-rev-expansion-hub-version-a) for the encoder to be working. We recommend checking the documentation of your motors to see which cables work for you, as different brands might have different requirements.

Let's use your differential drive robot. When you have your encoder cables, plug in one side of the cable to your motor and the other to your REV Hub (next to where you plugged the motors in):



Now you're ready to use encoders!

Fortunately, the code for using encoders are all the same. First, let's try one thing you can do now with an encoder: changing the motor settings to RUN_USING_ENCODER. In your TeleOp differential drive file, change the lines:

```
leftDrive.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
rightDrive.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
```

to:

```
leftDrive.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
rightDrive.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
```

Now, when you move your robot, it should be more stable than before (e.g. go forward in a straight line). The difference might be subtle, but it's there. Although we

recommend setting RunMode to RUN_USING_ENCODER for a differential drive robot, it might not be the best for other uses since you can run into unexpected behaviors.

Another thing you can do is get the encoder values for your motors. In your hardware DifferentialDrive class, add in the following lines:

```
public int[] getEncoderValues() {

    return new int[]{leftDrive.getCurrentPosition(), rightDrive.getCurrentPosition()};

}
```

This function will return an array that has the leftDrive encoder value in index1 and the rightDrive encoder value in index2.

Now go back to your TeleOp DifferentialDrive code and add:

```
telemetry.addData( caption: "Left: ", drive.getEncoderValues()[0]);
telemetry.addData( caption: "Right: ", drive.getEncoderValues()[1]);
```

inside loop().

Now, when you run TeleOp, you should see the left and right encoder values respectively.

What do those numbers even mean? Well, if you go to your motor documentation, you will most likely see a detailed information about your encoders. For example, for a NeveRest 40 Gearmotor, it says:

Ticks Per Revolution: 1120

This means that there are 1120 “ticks” (equivalent of encoder values) in one revolution. Using this information, we can precisely calculate how far the robot moved. For example, let’s say you are using 4” wheels for your robot. If your encoder value reads 22400, 22400 / 1120 = 20, therefore the wheel rotated 20 times exactly (or 20 revolutions). Since you are using 4” wheels, the circumference of the wheels are: 4π” ≈ 12.57”. Because your wheel rotated 20 times, 12.57” * 20 = 251.4”. So your robot moved approximately 251.4”. You can implement this calculation into your

autonomous code to make the robot move a certain distance *precisely* in autonomous mode.

Finally, encoders also allow you to move the motors to a certain “tick” with the RUN_TO_POSITION mode. To test this out, we can make a simple autonomous program:

```
@Autonomous(name="Drive Encoder Test")
public class EncoderTest extends LinearOpMode {

    DcMotor leftDrive;
    DcMotor rightDrive;

    @Override
    public void runOpMode() {

        leftDrive = hardwareMap.get(DcMotor.class, deviceName: "left_drive");
        rightDrive = hardwareMap.get(DcMotor.class, deviceName: "right_drive");
        leftDrive.setDirection(DcMotor.Direction.FORWARD);
        rightDrive.setDirection(DcMotor.Direction.REVERSE);

        leftDrive.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightDrive.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

        leftDrive.setTargetPosition(5000);
        rightDrive.setTargetPosition(5000);

        leftDrive.setMode(DcMotor.RunMode.RUN_TO_POSITION);
        rightDrive.setMode(DcMotor.RunMode.RUN_TO_POSITION);

        telemetry.addData( caption: "Status", value: "Ready to run");
        telemetry.update();

        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        leftDrive.setPower(0.5);
        rightDrive.setPower(0.5);

        while (opModeIsActive() && leftDrive.isBusy()) {

            idle();

        }

        leftDrive.setPower(0.0);
        rightDrive.setPower(0.0);

    }

}
```

These lines of code resets the encoder values:

```
leftDrive.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
rightDrive.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
```

Next, we set the encoder ticks we want the motors to rotate to with setTargetPosition(5000). After that, we set the motors to the RUN_TO_POSITION mode.

```
leftDrive.setTargetPosition(5000);
rightDrive.setTargetPosition(5000);

leftDrive.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightDrive.setMode(DcMotor.RunMode.RUN_TO_POSITION);
```

When we set the power now, the motors will stop automatically when the encoder value reaches 5000. However, we still need to turn the motors off by setting the power to 0 after the movement is done. The way we wait while the motor is running to the position is by using the while loop shown here:

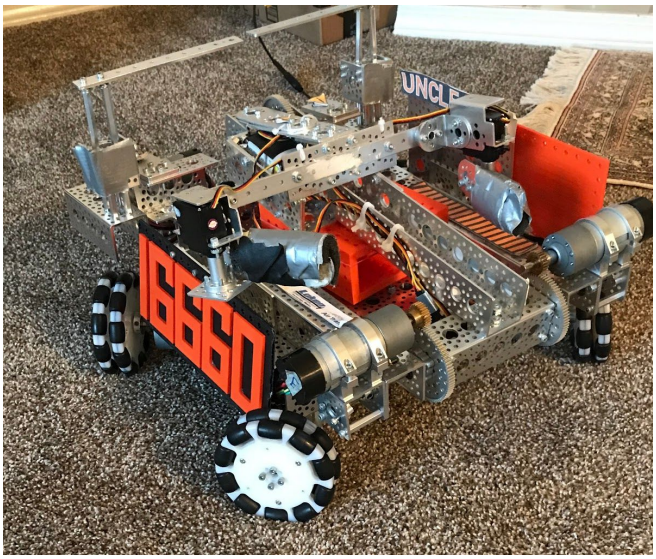
```
leftDrive.setPower(0.5);
rightDrive.setPower(0.5);

while (opModeIsActive() && leftDrive.isBusy()) {

    idle();

}

leftDrive.setPower(0.0);
rightDrive.setPower(0.0);
```

The isBusy() function returns true when the motor (leftDrive in our case) is still running to the position. This will allow us to set the motor powers to 0 after the translation is complete.

Now, when you run this autonomous program, you should travel a certain distance consistently (with 4” wheels and a NeveRest 40, it’ll be about 56”)!

Holonomic Drive

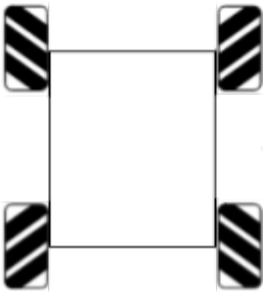
What even *is* a holonomic drive??? Simply said, a holonomic drive allows the robot to move in any direction without rotating. Take our 2019-2020 Skystone robot, Uncle Drew:



Notice how the omni wheels are rotated 45 degrees from their “normal” positions. This special configuration allows the robot to move in any direction. Robots with this configuration are called X-Drives.

Another common type of holonomic drive is the mecanum drive:

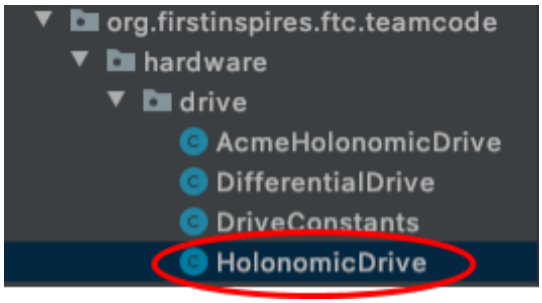
Although this is not an FTC bot, it still has the key distinguishing feature of a mecanum drive: the mecanum wheels. A mecanum drive allows the robot to be moved in any directions using a special type of wheels called mecanum wheels in the following orientation:



If your team is new to the FTC community or has financial constraints, we highly recommend using the X-Drive as the additional vector the robot can move to gives a huge advantage in most FTC games and the omni wheels (~$20) are not as expensive as the mecanum wheels (~$80). However, once finance is less of an issue for the team, we recommend you use the mecanum drive since the mecanum wheels are much more space-efficient, giving the robot additional space for additional subsystems.

Let's make a TeleOp code for a Holonomic Drive (it should work for both X-Drives and mecanum drives.

First, in the org.firstinspires.ftc.teamcode.hardware.drive package, create a new Java class called Holonomic Drive:



Let's start by setting up the motors:

```
public class HolonomicDrive {

    private DcMotor frontLeft;
    private DcMotor frontRight;
    private DcMotor backLeft;
    private DcMotor backRight;

    public HolonomicDrive(HardwareMap hwMap) {

        frontLeft = hwMap.get(DcMotor.class, deviceName: "frontLeft");
        frontLeft.setDirection(DcMotor.Direction.FORWARD);
        frontLeft.setPower(0);

        frontRight = hwMap.get(DcMotor.class, deviceName: "frontRight");
        frontRight.setDirection(DcMotor.Direction.REVERSE);
        frontRight.setPower(0);

        backLeft = hwMap.get(DcMotor.class, deviceName: "backLeft");
        backLeft.setDirection(DcMotor.Direction.FORWARD);
        backLeft.setPower(0);

        backRight = hwMap.get(DcMotor.class, deviceName: "backRight");
        backRight.setDirection(DcMotor.Direction.REVERSE);
        backRight.setPower(0);

        frontLeft.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
        frontRight.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
        backLeft.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
        backRight.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);

    }

}
```

Now, let's make a function that makes it drive (like what we did with differential drive):

```
public void drive(double leftStickX, double leftStickY, double rightStickX) {

    // Holonomic drive calculations
    double r = Math.hypot(leftStickX, leftStickY);
    double robotAngle = Math.atan2(-leftStickY, leftStickX) - Math.PI / 4;
    final double v1 = r * Math.cos(robotAngle) - rightStickX;
    final double v2 = r * Math.sin(robotAngle) + rightStickX;
    final double v3 = r * Math.sin(robotAngle) - rightStickX;
    final double v4 = r * Math.cos(robotAngle) + rightStickX;

    // Set motor powers
    frontLeft.setPower(v1);
    frontRight.setPower(v2);
    backLeft.setPower(v3);
    backRight.setPower(v4);

}
```

Woah woah... Do I see arctangent, sine, and cosine??? I thought this is robotics not trigonometry!

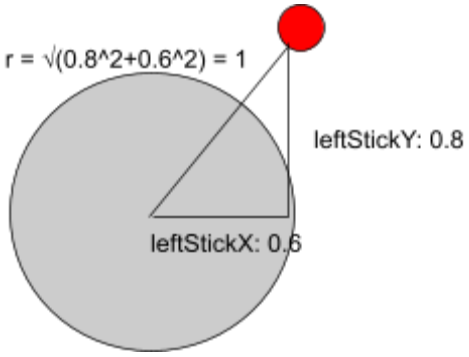
Okay let's digest what is going on.

First, the drive() function takes 3 parameters: leftStickX, leftStickY, and rightStickX. This is because we are going to use the left joystick to control the robot's xy motion and the right stick-x to turn the robot.

The first line calculates how fast the robot should be moving:

```
double r = Math.hypot(leftStickX, leftStickY);
```

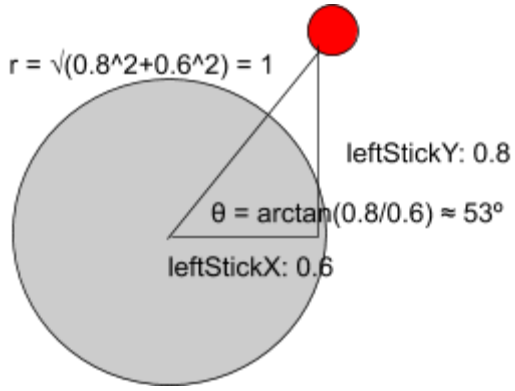
The Math.hypot function returns the hypotenuse of the two legs. For example, in our case, the two legs are leftStickX and leftStickY. Let's say your joystick position is like this:



If leftStickX = 0.6 and leftStickY = 0.8, the hypotenuse of that triangle is 1. Therefore, r = 1.

Next, we calculate the angle the robot should move to. This is done by calculating the arctangent of leftStickY and leftStickX:

```
double robotAngle = Math.atan2(-leftStickY, leftStickX) - Math.PI / 4;
```



So, in this example, you are telling the robot to move in the direction of 53°. leftStickY is inverted because down on the joystick is positive and up is negative. Why the Math.PI / 4 at the end? Because the motors are angled in a 45° fashion (pi / 4 is 90° in radians). Let's look at what we are doing next:

```
final double v1 = r * Math.cos(robotAngle) - rightStickX;
final double v2 = r * Math.sin(robotAngle) + rightStickX;
final double v3 = r * Math.sin(robotAngle) - rightStickX;
final double v4 = r * Math.cos(robotAngle) + rightStickX;
```

Here, we are calculating the values to assign to the motors. Let's ignore the rightStickX because all that does is make the robot turn. Which leaves us with this:

```
final double v1 = r * Math.cos(robotAngle)
final double v2 = r * Math.sin(robotAngle)
final double v3 = r * Math.sin(robotAngle)
final double v4 = r * Math.cos(robotAngle)
```

Take the previous example, if we are telling the robot to move in the direction of 53°, what has to happen is the angle of the vector of all wheels combined has to equal 53°.

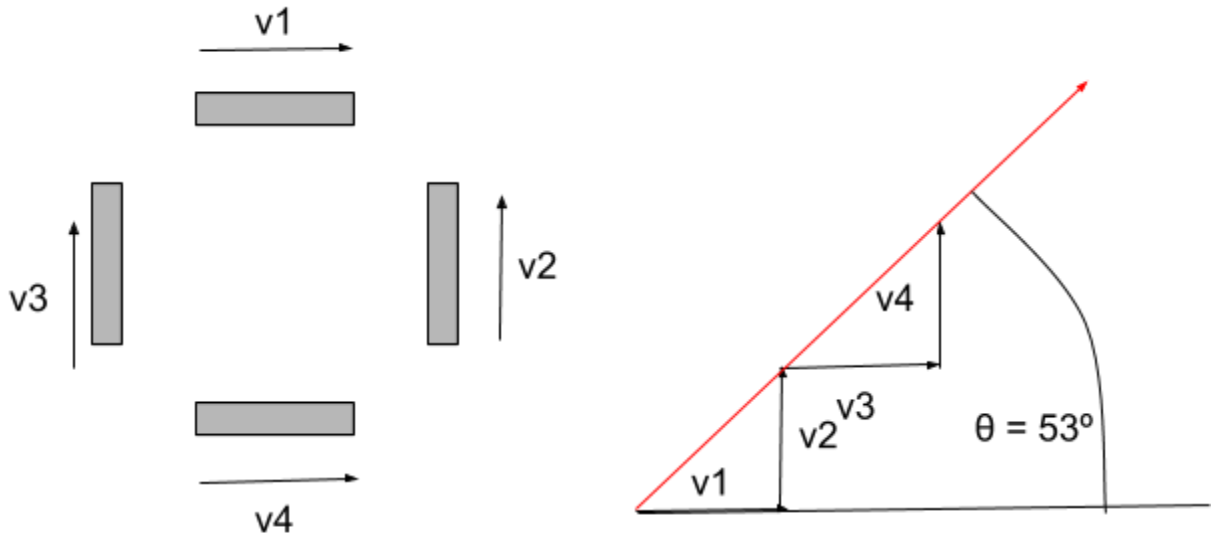


Figure not to scale

In this drawing, the rectangles are the wheels of an X-Drive rotated 45° (we have to compensate for this, hence - Math.PI / 4 in the line that is calculating the robot angle). v1, v2, v3, v4 are the vectors. The angle of all the vectors combined must equal robotAngle (53° in our case). The easiest way to calculate two vectors that add up to a certain angle is by using sine and cosine. In a unit circle, the cosine of the angle signifies the x-value (v1, v4) and the sine of the angle signifies the y-value (v2, v3), therefore the cosine of robotAngle = v1, v4 and the sine of robotAngle = v2, v3:

```
final double v1 = r * Math.cos(robotAngle)
final double v2 = r * Math.sin(robotAngle)
final double v3 = r * Math.sin(robotAngle)
final double v4 = r * Math.cos(robotAngle)
```

We multiply this number by “r” because that is how fast we want the robot to move (we scale it up by a factor of “r”).

Finally, we send these values to the motors:

```
// Set motor powers
frontLeft.setPower(v1);
frontRight.setPower(v2);
backLeft.setPower(v3);
backRight.setPower(v4);
```

Phew that was a lot of math! But with all that logistics explained, we can go ahead and move the robot!

Create a new TeleOp class called HolonomicTest and include the following:

```
@TeleOp(name = "Holonomic Test")
public class HolonomicTest extends OpMode {

    private HolonomicDrive drive;
    private Controller controller;

    @Override
    public void init() {

        controller = new Controller(gamepad1);
        drive = new HolonomicDrive(hardwareMap);

    }

    @Override
    public void loop() {

        controller.update();

        // Drive the robot
        double powerY = controller.left_stick_y;
        double powerX = controller.left_stick_x;
        double turnX = controller.right_stick_x / 1.5;
        drive.drive(powerX, powerY, turnX);

    }

}
```

Once you build an X-Drive or mecanum drive robot, you are ready to test this code!

... did it not work?

If your motors seemed to move in a random direction, see if you need to flip the directions of some of the motors. Changing DcMotor.Direction in the HolonomicDrive class will do that for you. If your robot was turning in the opposite direction, flip all the plus and minus signs here:

```
final double v1 = r * Math.cos(robotAngle) - rightStickX;
final double v2 = r * Math.sin(robotAngle) + rightStickX;
final double v3 = r * Math.sin(robotAngle) - rightStickX;
final double v4 = r * Math.cos(robotAngle) + rightStickX;
```

Once you set it all up correctly, your robot should move in any direction! But if you want it to move more precisely... let's move on to the next section.

PID Control

PID Control stands for Proportional Integral Derivative Control. Basically, it is a fancy way of saying "we use calculus to make a certain number accurately reach a certain number" (don't worry, no need to understand calculus).

PID Control is used everywhere around you!!! For example, let's say you heated an oven up. Well congratulations, you just used PID. Yes, an oven uses PID to make a certain number (the actual temperature of the oven) reach a certain number (the temperature you set up the oven to heat up to).

Well let's use this in robotics. You probably realized that your holonomic drive robot turns more and more as you strafe on either side. This is something we want to avoid because it really isn't a great experience for the drivers. So, we can use PID Control and a Gyro Sensor to make a certain number (the actual robot angle) reach a certain number (the desired robot angle) to make the robot strafe straight.

First, copy and paste this code into your org.firstinspires.ftc.teamcode package: https://github.com/FTC-16660/FtcRobotController/blob/master/TeamCode/src/main/java/org/firstinspires/ftc/teamcode/PIDController.java

This will allow you to easily use PID Controllers in your robot.

Now go to the HolonomicDrive class and include the following lines before the constructor:

```
private Gyro gyro;

// For PID corrections
private PIDController pidDrive = new PIDController( Kp: 0,  Ki: 0,  Kd: 0);
private double prevAngle;
private double correction;
```

We need a gyro for getting the angle of the robot and a PIDController for driving straight. For now, all PID values are set to 0 but we will change this later. The prevAngle variable is used to get the value we want to reach (known as the setpoint) and the correction variable is how we will compensate for the error in the robot angle.

Inside your constructor, add the lines:

```
gyro = new Gyro(hwMap);
prevAngle = gyro.getAngleDegrees();
```

This will initialize the gyro sensor and set prevAngle to the angle the robot is facing.

We will significantly modify drive() as well.

Add the following lines at the beginning of the drive() function:

```
// pid calculation
pidDrive.setSetpoint(prevAngle);
correction = pidDrive.performPID(gyro.getAngleDegrees());
```

pidDrive.setSetpoint(prevAngle) will tell our PIDController object that the value we want to reach is the prevAngle. Then, we performPID with the gyro.getAngleDegrees() function, meaning to tell our code how much we have to “correct” the robot angle by and give that value to the “correction” variable.

Let’s also modify the values we send to the motors:

```
double v1 = r * Math.cos(robotAngle) - rightStickX + correction;
double v2 = r * Math.sin(robotAngle) + rightStickX - correction;
double v3 = r * Math.sin(robotAngle) - rightStickX + correction;
double v4 = r * Math.cos(robotAngle) + rightStickX - correction;
```

You probably realized that we now have the +- correction at the end of the lines. This is to add the additional correction that has to be done to the motors.

Between the holonomic drive calculations and the pid calculations, add the following lines of code:

```
// If turning or not moving then don't correct
if (Math.abs(rightStickX) != 0 ||
    (leftStickX == 0 && leftStickY == 0)) {

    pidDrive.disable();
    pidDrive.reset();
    prevAngle = gyro.getAngleDegrees();
    correction = 0;

} else
    pidDrive.enable();
```

As the comment says, we don't want to correct the robot when it is turning. I mean, we want the robot to turn and not freak out because it wants to correct itself to the "correct" position! The if statement says "if the absolute value of rightStickX is not equal to 0 (meaning it is turning) or leftStickX and leftStickY are both 0 (meaning it is not moving), then disable and reset pidDrive and correction as well as set the prevAngle to the current robot angle. We want to set prevAngle to the current robot angle because that's what you want your setpoint as when you are driving: the angle of the robot from the last time you either turned or stopped moving the robot. When the robot is either driving straight or strafing, the else statement runs and enables pidDrive.

The end result is this:

```
// Drive calculations with PID
public void drive(double leftStickX, double leftStickY, double rightStickX) {

    // pid calculation
    pidDrive.setSetpoint(prevAngle);
    correction = pidDrive.performPID(gyro.getAngleDegrees());

    // If turning or not moving then don't correct
    if (Math.abs(rightStickX) != 0 ||
        (leftStickX == 0 && leftStickY == 0)) {

        pidDrive.disable();
        pidDrive.reset();
        prevAngle = gyro.getAngleDegrees();
        correction = 0;

    } else
        pidDrive.enable();

    // Holonomic drive calculations
    double r = Math.hypot(leftStickX, leftStickY);
    double robotAngle = Math.atan2(-leftStickY, leftStickX) - Math.PI / 4;
    double v1 = r * Math.cos(robotAngle) - rightStickX + correction;
    double v2 = r * Math.sin(robotAngle) + rightStickX - correction;
    double v3 = r * Math.sin(robotAngle) - rightStickX + correction;
    double v4 = r * Math.cos(robotAngle) + rightStickX - correction;

    // Set motor powers
    frontLeft.setPower(v1);
    frontRight.setPower(v2);
    backLeft.setPower(v3);
    backRight.setPower(v4);

}
```

Now, when you run your HolonomicTest on your DS, your robot should... do nothing different. This is because you still haven't set the PID values yet!

Here is a basic graph of what changing each of the PID values will affect your robot (or anything using PID, really):

	Rise Time	Overshoot	Settling Time	Steady-State Error	Stability
Increasing P	Decrease	Increase	Small Increase	Decrease	Degrade
Increasing I	Small Decrease	Increase	Increase	Large Decrease	Degrade
Increasing D	Small Decrease	Decrease	Decrease	Minor Change	Improve

You can set the PID values in this line of code at the beginning of the HolonomicDrive class:

```
// For PID corrections
private PIDController pidDrive = new PIDController( Kp: 0,  Ki: 0,  Kd: 0);
```

Usually, you would start first by setting up a P value (at around 0.1 should work for this). If the robot starts oscillating too much, reduce the value. If it seems to have not much affect, increase the value. If the robot is going crazy and rotating around in a circle (yes, that does happen), try switching up the plus and minus signs before “correction” here:

```
double v1 = r * Math.cos(robotAngle) - rightStickX + correction;
double v2 = r * Math.sin(robotAngle) + rightStickX - correction;
double v3 = r * Math.sin(robotAngle) - rightStickX + correction;
double v4 = r * Math.cos(robotAngle) + rightStickX - correction;
```

Once the robot is oscillating around the desired angle, increase the D value until the robot doesn’t oscillate much but still actively correcting itself. Increasing D, as seen on the table, will increase stability.

For this, you probably won’t need to use the I value. This is used in a future section. However, what the I value mainly does is decrease the steady-state error, meaning it helps when the robot is not reaching its setpoint for a long period of time. Although this is useful in some cases, in this case, we won’t use it since it simply reduces stability.

Once you get the tuning right, your robot should move flawlessly!

Automated Holonomic Drive

First, I should congratulate you for making it to the final section of this book! This section won't cover anything new. But it will provide you with useful code for autonomous. We will use encoders, PID controllers, and a gyro sensor to accurately move your holonomic drive robot automatically wherever you want it to go!

First go to your HolonomicDrive class and add two more PIDControllers before the constructor class and name them pidSpeed and pidTurn.

```
private PIDController pidSpeed = new PIDController( Kp: 0,  Ki: 0,  Kd: 0);
private PIDController pidTurn = new PIDController( Kp: 0,  Ki: 0,  Kd: 0);
```

As the name name suggests, one of them is going to be used for controlling the speed of the robot and the other one for turning the robot.

You should also type in the following line before the constructor as well:

```
private LinearOpMode opmode = null;
```

Since some autonomous functions such as the sleep() and opModeIsActive() are only available for LinearOpModes, we are going to create a new LinearOpMode object and use the function from that object.

Let's also clean up some code in the constructor function. First, cut (control or command + X) all the motor setup code inside the constructor function and place it in a new function called setupMotors() like so (also added STOP_AND_RESET_ENCODER to reset encoder values):

```
private void setupMotors(HardwareMap hwMap) {

    frontLeft = hwMap.get(DcMotor.class, deviceName: "frontLeft");
    frontLeft.setDirection(DcMotor.Direction.FORWARD);
    frontLeft.setPower(0);

    frontRight = hwMap.get(DcMotor.class, deviceName: "frontRight");
    frontRight.setDirection(DcMotor.Direction.REVERSE);
    frontRight.setPower(0);

    backLeft = hwMap.get(DcMotor.class, deviceName: "backLeft");
    backLeft.setDirection(DcMotor.Direction.FORWARD);
    backLeft.setPower(0);

    backRight = hwMap.get(DcMotor.class, deviceName: "backRight");
    backRight.setDirection(DcMotor.Direction.REVERSE);
    backRight.setPower(0);

    frontLeft.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    frontRight.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    backLeft.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    backRight.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

    frontLeft.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
    frontRight.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
    backLeft.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
    backRight.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);

}
```

Now cut the encoder codes (the last 8 lines) and place it in another function called setupEncoders():

```
private void setupEncoders() {

    frontLeft.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    frontRight.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    backLeft.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    backRight.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

    frontLeft.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
    frontRight.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
    backLeft.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
    backRight.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);

}
```

Call setupEncoders() at the end of the setupMotors() function, and call setupMotors() at the end of the constructor.

A few steps before, we created a new LinearOpMode object called “opmode” and assigned “null” to use functions specific to the LinearOpMode class. But here’s the problem: for TeleOp, we use the class OpMode which is different from LinearOpMode, which means it will cause errors in the code if we use the “opmode” object for TeleOp and vice versa. So we need a way for the code to be able to distinguish between TeleOp code and Autonomous code, and we do this by creating *another* constructor class.

Here is the second constructor:

```
public HolonomicDrive(LinearOpMode opmode, HardwareMap hwMap) {

    this.opmode = opmode;

    gyro = new Gyro(hwMap);
    prevAngle = gyro.getAngleDegrees();

    setupMotors(hwMap);

}
```

Looks almost exactly the same as the other one except it has an additional parameter and an additional line. If we are going to use the HolonomicDrive class for autonomous, we are going to use this newly created constructor, and for TeleOp, we are still going to use the other constructor. Since this.opmode = opmode (we use “this.” to distinguish between the opmode created in the parameter and the opmode created before the constructor) assigns a value that is not “null”, we know that if opmode has a value of “null”, it is being used for TeleOp and otherwise it is being used for autonomous.

At the end of the setupEncoders() method, add the following lines of code:

```
if (opmode == null) {

    frontLeft.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.FLOAT);
    frontRight.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.FLOAT);
    backLeft.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.FLOAT);
    backRight.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.FLOAT);

} else {

    frontLeft.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);
    frontRight.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);
    backLeft.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);
    backRight.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);

}
```

This tells the robot that if it is running a TeleOp code (opmode == null), set the zero power behavior of the motors to FLOAT and if it is running an autonomous code, to BRAKE. What's the difference? If it is set to FLOAT (default mode), the motors will run freely when power is set to 0, but if it is set to BRAKE, the motors won't move freely. In autonomous, we want it in BRAKE because having the motors run freely may result in imprecise movement. These options only take effect when encoders are installed, and we are going to use more encoders later, so make sure the encoder cables are installed.

Finally, we should create a new method called resetMotors() to... well, reset motors:

```
public void resetMotors() {

    frontLeft.setPower(0);
    frontRight.setPower(0);
    backLeft.setPower(0);
    backRight.setPower(0);

    frontLeft.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    frontRight.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    backLeft.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    backRight.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

    frontLeft.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
    frontRight.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
    backLeft.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
    backRight.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);

}
```

The code itself is pretty self-explanatory. Set the powers to 0, and reset the encoders.

Cool! We have all the setups complete now, let's move on to the code that moves the robot.

There are three important things a holonomic drive robot can perform: moving forwards/backwards, moving sideways, and turning. Let's start with moving forwards/backwards.

First create a new function called moveForward() that takes in two parameters: speed and position:

```
public void moveForward(double speed, int position) {



}
```

"speed" will take in a value from -1 to 1 and tells the motors how fast and in which direction the robot should move. "position" will take in any integer value and tells the robot how far it should move.

Then, add:

```
gyro.reset();
resetMotors();
opmode.sleep( milliseconds: 100);

pidDrive.reset();
pidDrive.setSetpoint(0);
pidDrive.enable();

pidSpeed.reset();
pidSpeed.setSetpoint(position);
pidSpeed.setInputRange(0, position);
pidSpeed.setOutputRange(0, speed);
pidSpeed.setTolerance(1);
pidSpeed.enable();
```

This will do all the setup. Reset the gyro sensor, reset the motors, and reset the PIDs. It will also give the robot some time to just take a breath and stay calm for 100 milliseconds using opmode.sleep() (this will ensure a more accurate and precise motion). Something to take note of here are these three lines:

```
pidSpeed.setInputRange(0, position);
pidSpeed.setOutputRange(0, speed);
pidSpeed.setTolerance(1);
```

For the first line, you put in the range of inputs the PID controller can expect to get. Since (*spoiler alert) your pidSpeed value will take the encoder value as an input, we can say that input range is expected to be between 0 and whatever number "position" is. For the second line, you put in the range of outputs the PID controller can expect to output. Since you want to control the speed of the robot using pidSpeed and our maximum speed is the speed we put for the "speed" parameter, we can say that the range of outputs is between 0 and "speed". For the third line, we put the tolerance percentage for the PID controller. This stops the PID corrections once it reaches the setpoint +- the tolerance percentage of the setpoint. For example, if your setpoint is the value 5000 and you set the tolerance percentage to 1, the PID controller will stop its corrections once the input reaches the range [5000 - 5000 * 1%, 5000 + 5000 * 1%] = [4950, 5050]. This is necessary because without the tolerance, the PID controller will continue to correct itself until it reaches the setpoint *exactly* (e.g. 5000) which isn't ideal because the robot will continue to oscillate itself.

Next, we can put the while loop that moves the robot:

```
do {

    correction = pidDrive.performPID(gyro.getAngleDegrees());

    double averagePosition = (frontLeft.getCurrentPosition() + frontRight.getCurrentPosition() +
                              backLeft.getCurrentPosition() + backRight.getCurrentPosition()) / 4.0;
    speed = pidSpeed.performPID(averagePosition);

    frontLeft.setPower(speed - correction);
    frontRight.setPower(speed + correction);
    backLeft.setPower(speed - correction);
    backRight.setPower(speed + correction);

} while (opmode.opModeIsActive() && !pidSpeed.onTarget());
```

What's the condition? opmode.opModeIsActive() returns "true" if the opmode is active. More importantly, pidSpeed.onTarget() returns true if pidSpeed has reached the tolerated setpoint (why setting the tolerance is important). Since it has an

exclamation mark at the front, the while loop continues to run while the pidSpeed controller has not reached its desired setpoint.

```
while (opmode.opModeIsActive() && !pidSpeed.onTarget());
```

The “correction” variable has the same job here as before in the drive() function for TeleOp - it makes the robot drive as straight as possible using the gyro input.

```
correction = pidDrive.performPID(gyro.getAngleDegrees());
```

The “averagePosition” variable, as the name suggests, calculates the average encoder value of all the motors:

```
double averagePosition = (frontLeft.getCurrentPosition() + frontRight.getCurrentPosition() +
                          backLeft.getCurrentPosition() + backRight.getCurrentPosition()) / 4.0;
```

The “speed” variable (or I guess parameter) contains the PID correction of pidSpeed. This will be the speed of your robot.

Finally, we put speed +- the pidDrive correction as our parameter for setPower(). Again, if the robot is going crazy, try flipping all plus and minus signs here:

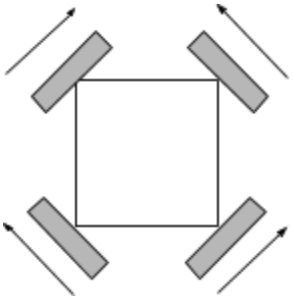
```
frontLeft.setPower(speed - correction);
frontRight.setPower(speed + correction);
backLeft.setPower(speed - correction);
backRight.setPower(speed + correction);
```

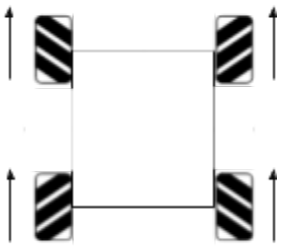
That's it for moving forwards and backwards! Let's move on to moving sideways:

```
public void moveSide(double speed, int position) {

    gyro.reset();
    resetMotors();
    opmode.sleep( milliseconds: 100);

    pidDrive.reset();
    pidDrive.setSetpoint(0);
    pidDrive.enable();

    pidSpeed.reset();
    pidSpeed.setSetpoint(position);
    pidSpeed.setInputRange(0, position);
    pidSpeed.setOutputRange(0, speed);
    pidSpeed.setTolerance(1);
    pidSpeed.enable();

    do {
        correction = pidDrive.performPID(gyro.getAngleDegrees());

        double averagePosition = (frontLeft.getCurrentPosition() - frontRight.getCurrentPosition() -
                backLeft.getCurrentPosition() + backRight.getCurrentPosition()) / 4.0;
        speed = pidSpeed.performPID(averagePosition);

        frontLeft.setPower(speed - correction);
        frontRight.setPower(-speed + correction);
        backLeft.setPower(-speed - correction);
        backRight.setPower(speed + correction);

    } while (opmode.opModeIsActive() && !pidSpeed.onTarget());

}
```

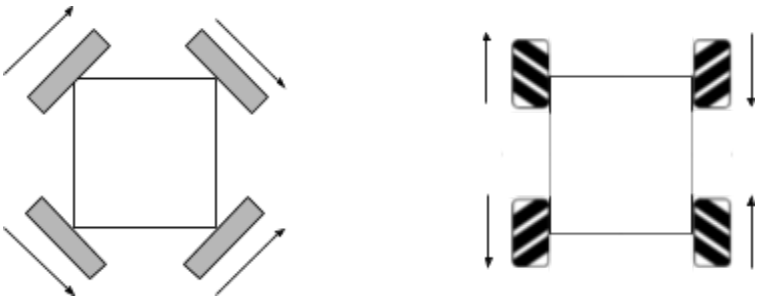
You might think that this looks almost exactly the same as the other one. Well, that's because it *is* almost exactly the same. The only parts that are slightly different are in the do-while loop.

First, the way we calculate the average position is slightly different. Instead of adding all values, we are adding some while subtracting some other. Think about it, we move the robot forward by moving the motors like so:





We move it sideways by moving the motors like this:



Notice how only the front-right and the back-left motors have switched directions. This is why we swap the operations for frontRight and backLeft:

```
double averagePosition = (frontLeft.getCurrentPosition() - frontRight.getCurrentPosition() -
        backLeft.getCurrentPosition() + backRight.getCurrentPosition()) / 4.0;
```

Since those motors are moving in the opposite direction, the values are switched, so we flip the operators as well to prevent values from canceling each other.

Then of course, the values to assign to the motors are a tiny bit different as well with the additional negative values:

```
frontLeft.setPower(speed - correction);
frontRight.setPower(-speed + correction);
backLeft.setPower(-speed - correction);
backRight.setPower(speed + correction);
```

Last one! The robot still has to turn.

Copy these lines of code into your HolonomicDrive class:

```
public void turn(int angle, double speed) {

    gyro.reset();
    resetMotors();
    opmode.sleep( milliseconds: 100);

    if (Math.abs(angle) > 359) angle = (int) Math.copySign(359, angle);

    pidTurn.reset();
    pidTurn.setSetpoint(-angle);
    pidTurn.setInputRange(0, -angle);
    pidTurn.setOutputRange(0, speed);
    pidTurn.setTolerance(1);
    pidTurn.enable();

    do {

        speed = pidTurn.performPID(gyro.getAngleDegrees());

        frontLeft.setPower(speed);
        frontRight.setPower(-speed);
        backLeft.setPower(speed);
        backRight.setPower(-speed);

    } while (opmode.opModeIsActive() && !pidTurn.onTarget());

}
```

The format is actually quite similar to moving the robot forwards/backwards/sideways.

The first few lines reset the gyro and the motors and makes the robot wait for 100 milliseconds:

```
gyro.reset();
resetMotors();
opmode.sleep( milliseconds: 100);
```

Then, we cap the angle in the range of [-359, 359] in this line:

```
if (Math.abs(angle) > 359) angle = (int) Math.copySign(359, angle);
```

After that, we reset and set input range, output range, and tolerance of pidTurn which we will be using for turning the robot.

```
pidTurn.reset();
pidTurn.setSetpoint(-angle);
pidTurn.setInputRange(0, -angle);
pidTurn.setOutputRange(0, speed);
pidTurn.setTolerance(1);
pidTurn.enable();
```

Finally, like what we did in moveSide() and moveForward(), we have a do-while loop that keeps the motors moving until the setpoint is reached:

```
do {

    speed = pidTurn.performPID(gyro.getAngleDegrees());

    frontLeft.setPower(speed);
    frontRight.setPower(-speed);
    backLeft.setPower(speed);
    backRight.setPower(-speed);

} while (opmode.opModeIsActive() && !pidTurn.onTarget());
```

Phew! Lot's of coding right there. Hang on a little more though! We'll actually start writing the autonomous code now.

Luckily, this is the easiest part of the code because you did all the hard calculations and programming in the hardware file! Now the autonomous code can be super streamlined and easy to understand.

Let's just make our holonomic drive robot move straight.

Here is the code:

```
@Autonomous(name="Test Autonomous")
public class TestAutonomous extends LinearOpMode {

    private FortyFiveDrive robot;

    @Override
    public void runOpMode() {

        robot = new FortyFiveDrive( opmode: this, hardwareMap);

        telemetry.addData( caption: "Status",  value: "Ready to run");
        telemetry.update();

        waitForStart();

        // Move the robot forwards
        robot.moveForward( speed: 1,  position: 2000);

        telemetry.addData( caption: "Path",  value: "Complete");
        telemetry.update();

    }

}
```

Super simple! Really the only line of code that makes the robot move is “robot.moveForward(1,2000);” which will move the robot forwards with max speed until it reaches the encoder position of 2000.

Let's add more things:

```
@Override
public void runOpMode() {

    robot = new FortyFiveDrive( opmode: this, hardwareMap);

    telemetry.addData( caption: "Status",  value: "Ready to run");
    telemetry.update();

    waitForStart();

    // Make the robot do stuff
    robot.moveForward( speed: 1,  position: 1000);
    robot.moveSide( speed: 0.7,  position: 1000);
    robot.turn( angle: -90,  speed: 0.6);
    robot.moveSide( speed: 0.3,  position: -500);
    robot.turn( angle: -45,  speed: 0.8);

    telemetry.addData( caption: "Path",  value: "Complete");
    telemetry.update();

}
```

This will make the robot move forwards, sideways, turn, move sideways to the other side, and turn to the other direction. Although some PID tuning is required, this piece of code is extremely easy to read, and works very well in moving the robot for autonomous.

When you run this code, the robot should move according to the commands you wrote!