Xaverian Robotics Team – Programming Department

How to Robot

Programming explained by a "professional"

Written by: Michael Lachut (Class of 2025) – Co-Captain and Director of Software

Publish Date: October 2023

Contents

Introduction

What is this guide?

This guide is my attempt at teaching people how to write some ***working*** code for the robot. It will go over what software you will need, what languages you will learn, and more importantly, you will learn how to be a mildly successful programmer like me. My goal is to separate this guide into small bite-sized chapters which can be read in less than 15 minutes.

More specifically, what will I learn?

You will learn the fine art of Android Studio, a software used by programmers world-wide to write code for the billions of Android devices all around the world. You will utilize what the *JVM languages*, or Java and Kotlin, to write, test, and fix code for our robot.

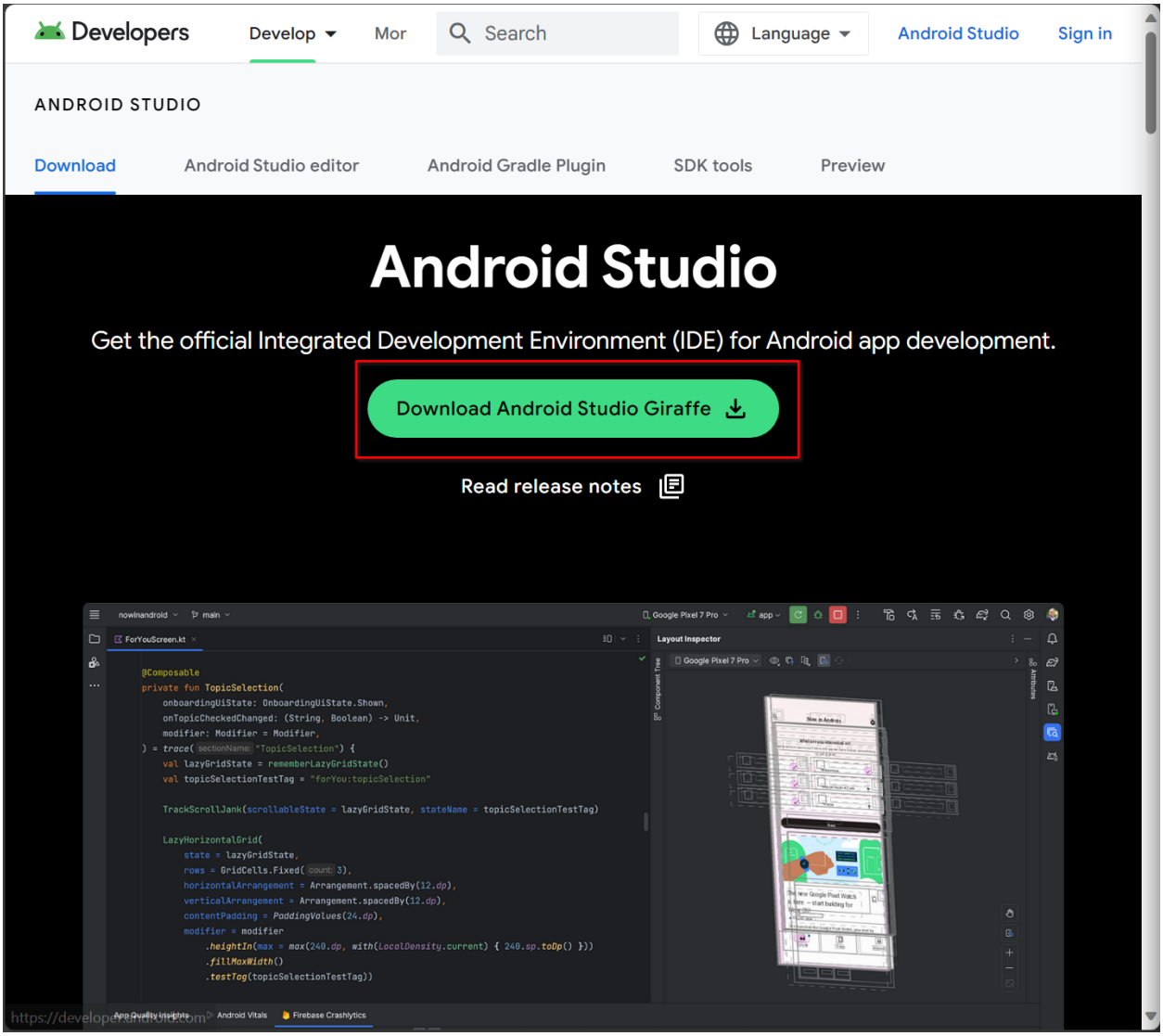
Lastly, what will I need to begin this guide?

You will need…. nothing! The Mac or Windows computer that you use in your day-to-day life will work fine, however you will need to clear up enough storage. On Windows computers, generally 10 GB is fine, but for Macs it *might* be more. You will also need to have basic knowledge on how our robot will work, such as how motors and servos work, how different sensors can be combined, and how the *Control Hubs*, *Driver Stations*, and *Expansion Hubs* interact with each other.

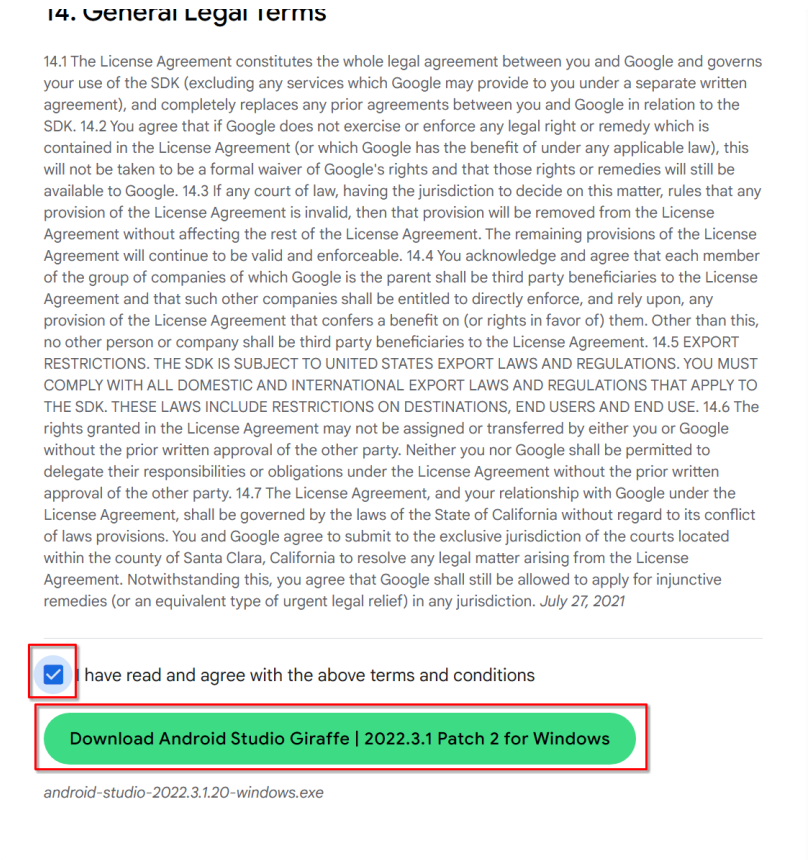
Prerequisites

Installing Android Studio

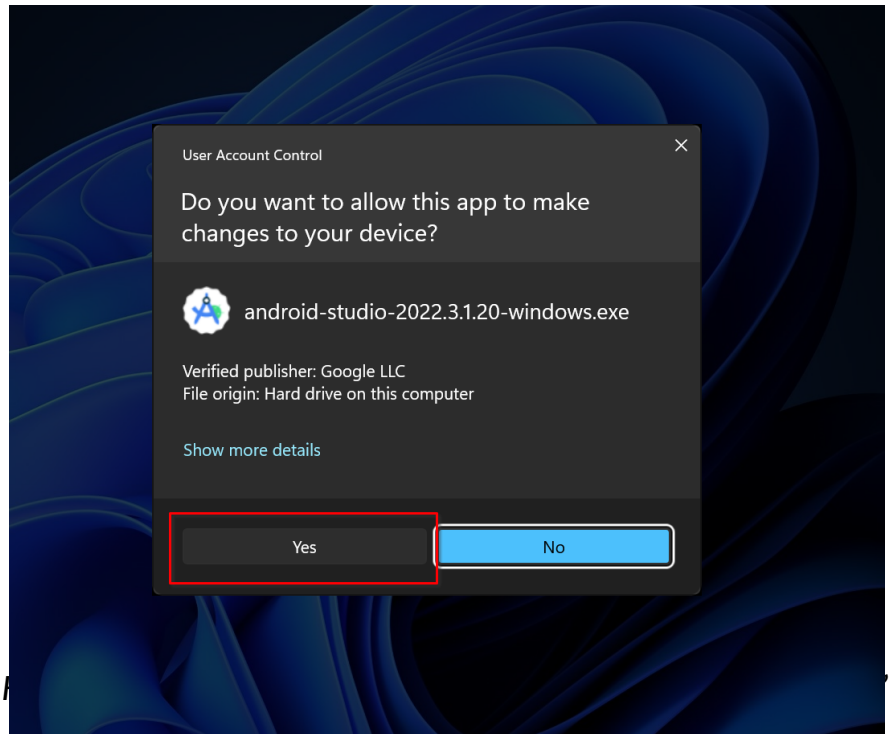
Installing Android Studio changes based on whether you are on a Mac or a Windows computer, and I can't write a guide for both, so instead I will just write a guide for Windows. First thing you have to do is navigate to developer.android.com/studio and click the "Download Android Studio" button.



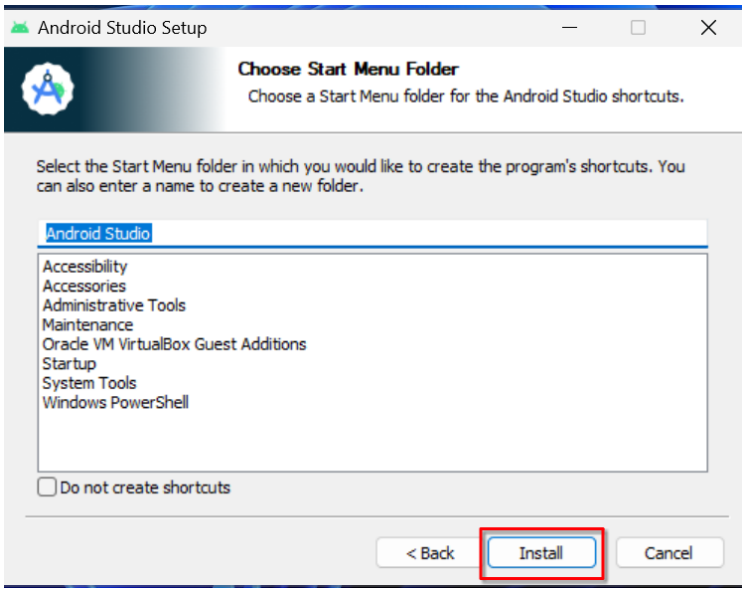
Then, agree to the terms and conditions and click Download.



Then, when the download is done, run the file and click “Yes” on the security prompt.

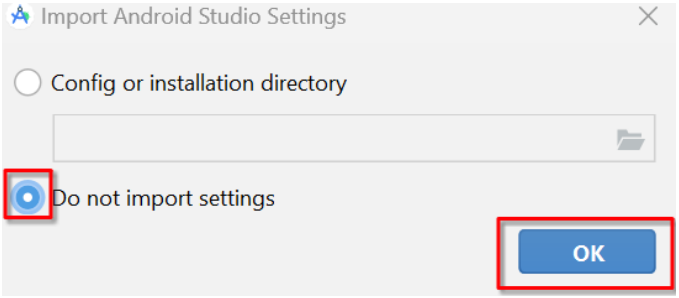


Then click “Next” in the installer, letting Android Studio keep all the default options, until you get to this final page where you click “Install”.

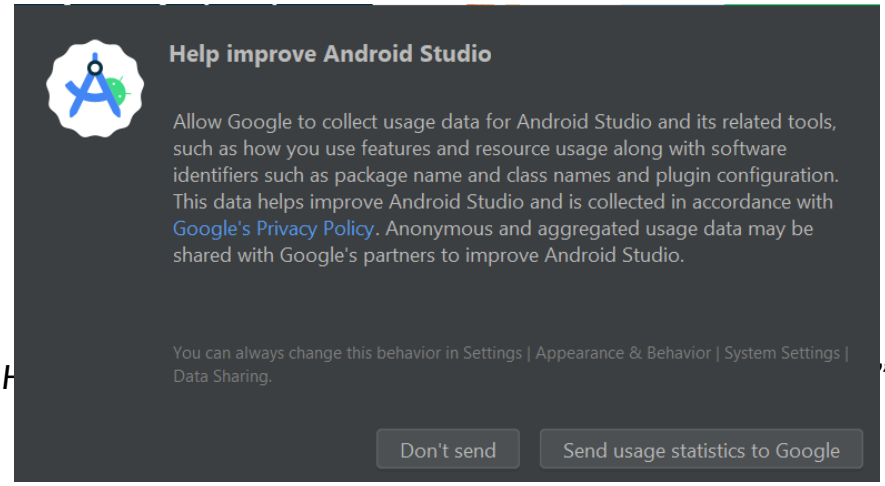


Wait for the installer to finish and click “Next”, then “Finish”.

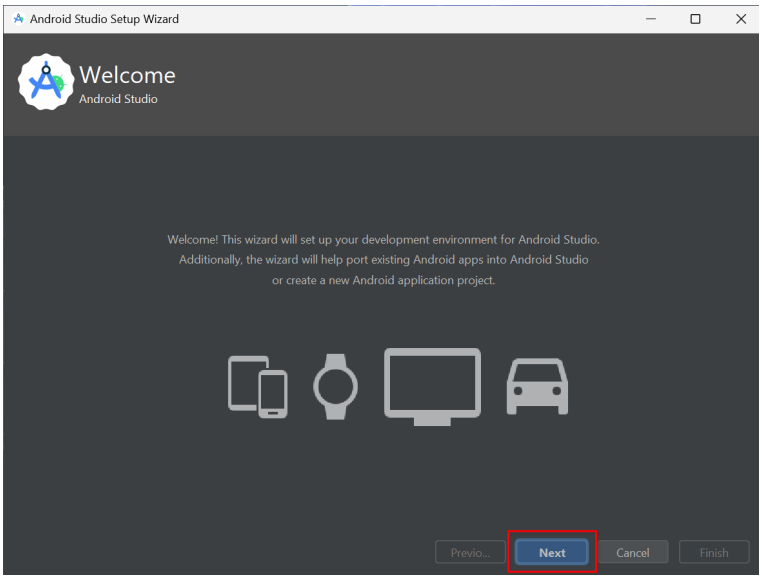
Now we need to do the first-time setup. When you see the menu to import settings, check the “Do not import settings” box and click “OK”.



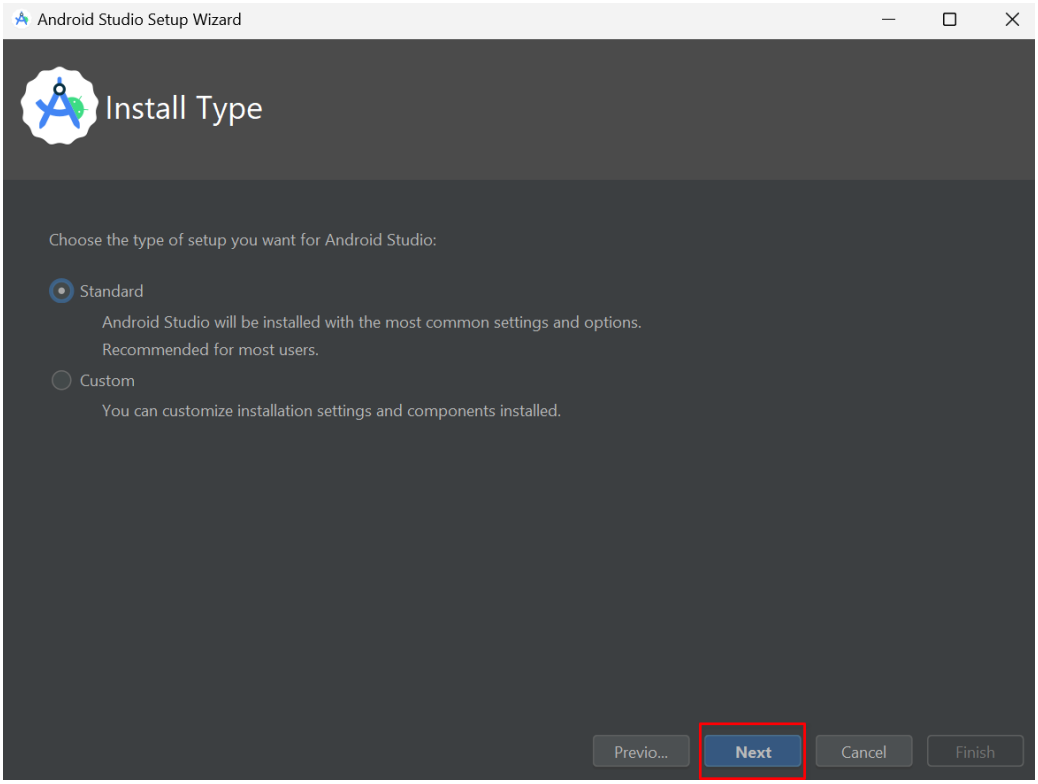
When prompted with the “Help improve Android Studio” popup, I recommend you choose “Don’t send”, but it’s up to you.



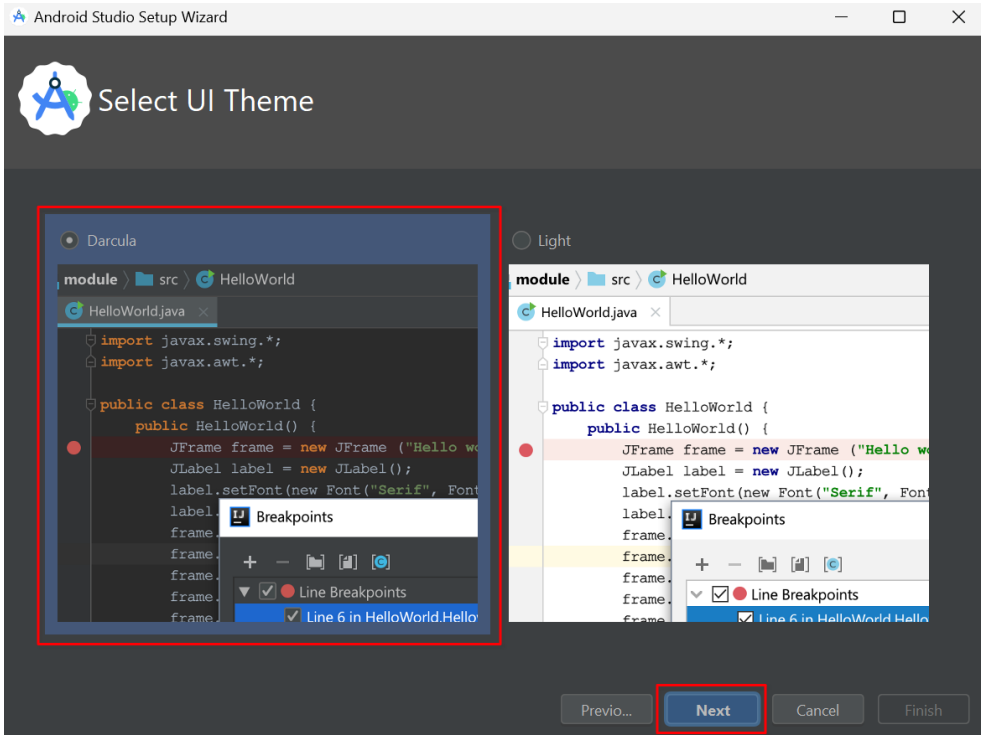
Then, click next to begin choosing your settings.



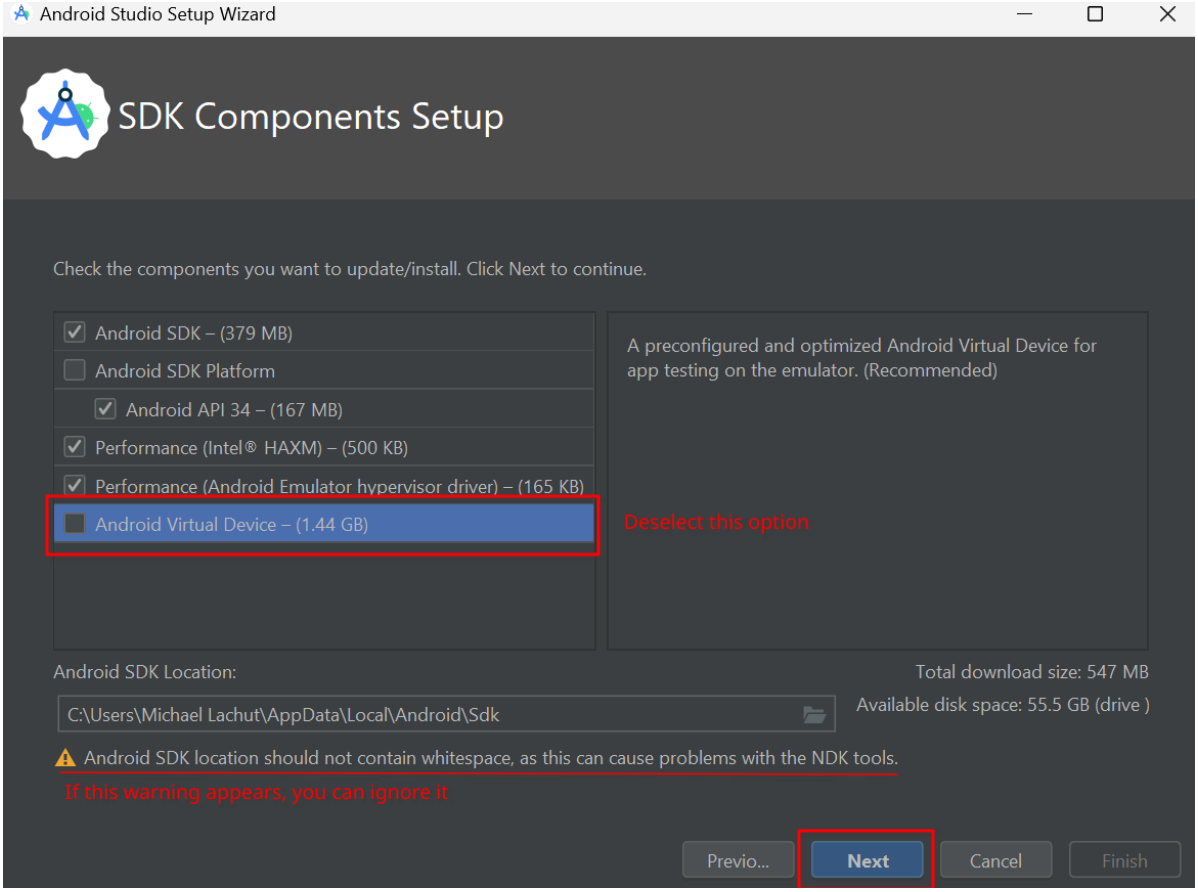
Click the “Next” button again to proceed with the default settings.



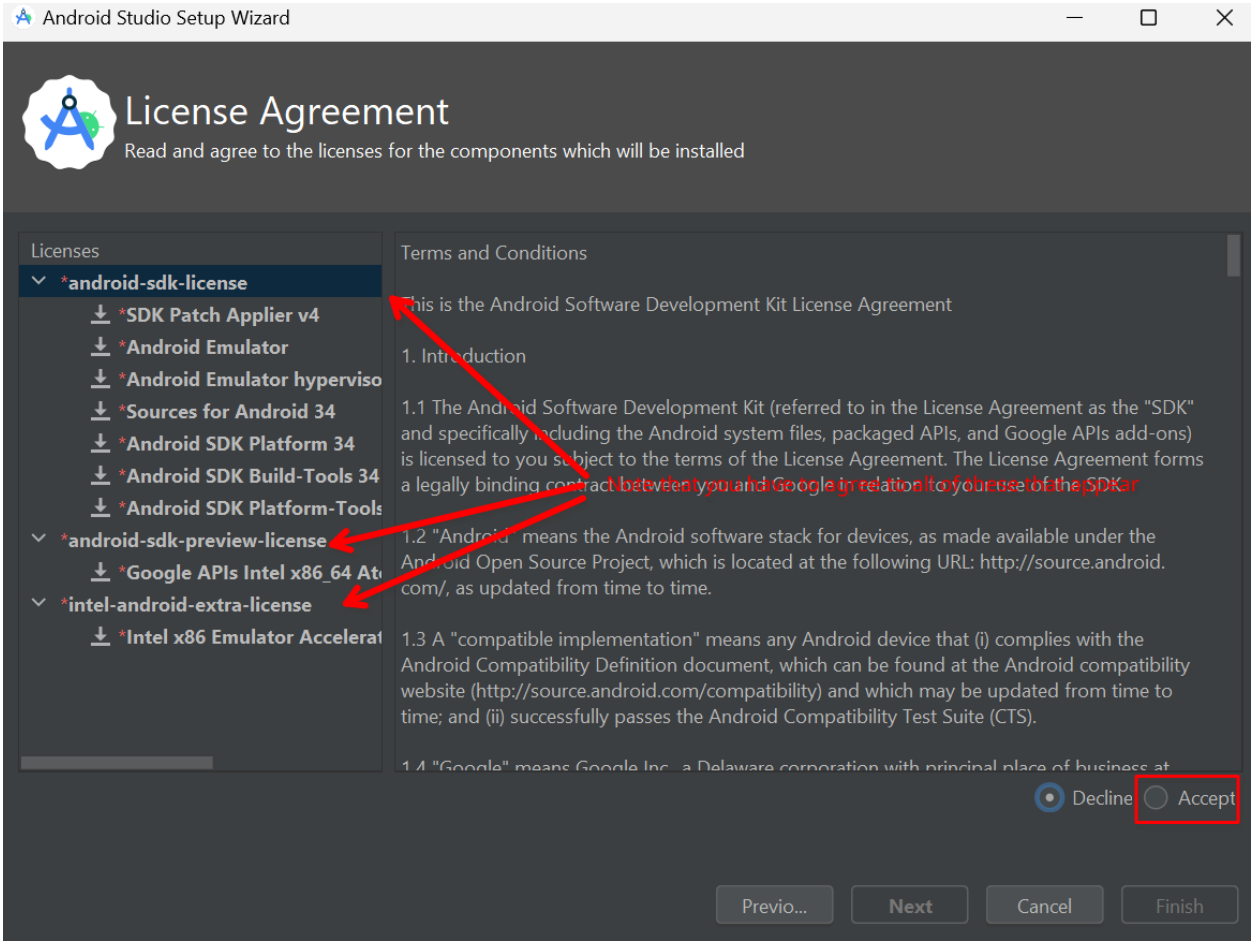
When you are prompted to choose your theme, it all boils down to your personal preference. Some people like light theme, but a common opinion that most programmers share is that light mode can hurt your eyes after extended periods of time. For that we recommend you choose dark theme.



Next it will ask you which SDK Components you want to install. I would recommend you deselect the “Android Virtual Device” option because it tends to waste delicate disk space and tends to use up your battery in the background.



Then click next again in the confirmation screen and then agree to the terms and conditions.



Then click “Finish” once you are done agreeing to the terms and conditions. The setup process will begin and now you can sit back and relax. Click “Finish” when the installer is done.

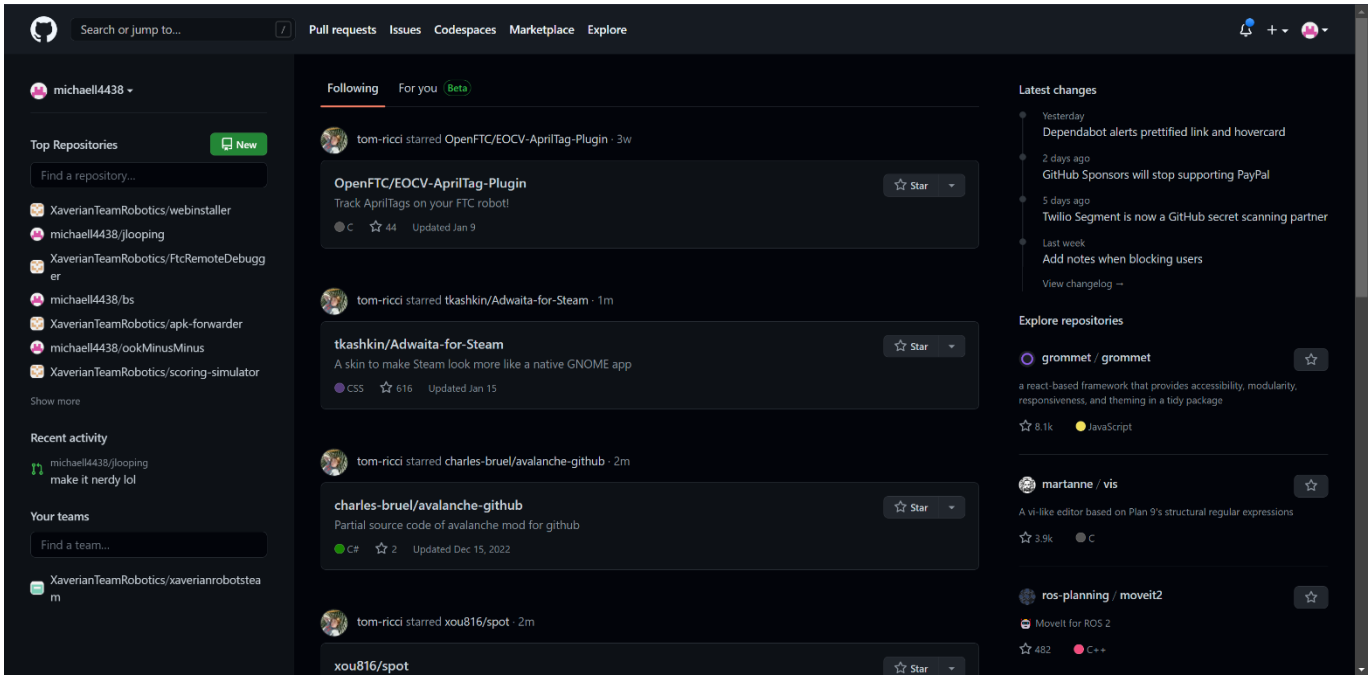
1 – GitHub

What is GitHub?

GitHub.com is a website that we *(and most people)* use to collaborate on code. However, it doesn't work using the same way you may be used to as for something on Google Docs or Slides. Instead of sharing each line of code individually, every time you make some changes, let's say you adjust some servo values, you fix an error, or add a new feature, you do what's called a *commit*. Commits simply say what you changed and have a little message you can add to help other people keep track of what commit changed what. Code is contained in what are called *repositories*. Repositories *(or "repos" for short)* contain a project, and, on your computer, you *clone* a repository to create a copy of it on your computer and periodically you download or upload the commits you or other people make. The version online is called the *upstream repository* and the copy on your computer is the *local repository*. This allows for better collaboration than what happens when you have 10 people on the same Google Doc. Also in repositories are these things called *branches* which are used to separate different versions of the same code so that each person who commits to a branch can be sure that they won't disrupt someone else.

Creating a GitHub Account

Creating a GitHub account is very simple. Go to [this link] and create an account using your preferred e-mail address and an easy-to-remember password. When you are done and you are logged in, you should be at your home page, which will look different for everyone, but will have the same layout.

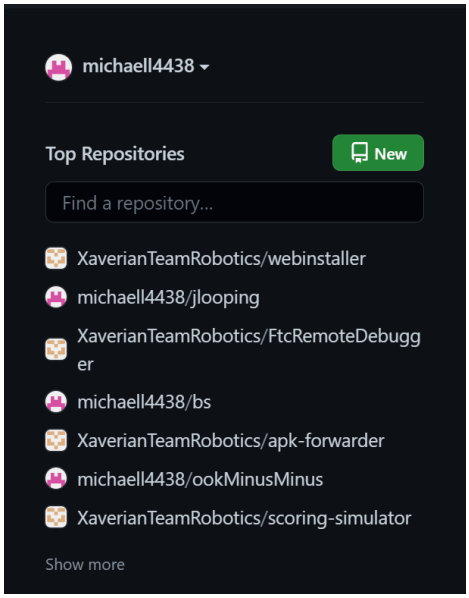


Joining the Organization

To collaborate with the rest of the team, you will have to join the organization. To do this, you will have to ask one of the coaches or another one of the experienced programmers and they can invite you.

Exploring the Organization

Upon joining to the organization, you will see some repositories on the left side of the home page. In the screenshot which I attached, there are some ones that I have made as well. If you click on the *Show more* button then you can see all of your repositories. However, the easiest way to see all repositories for an organization is to visit this link: https://github.com/XaverianTeamRobotics. Either way

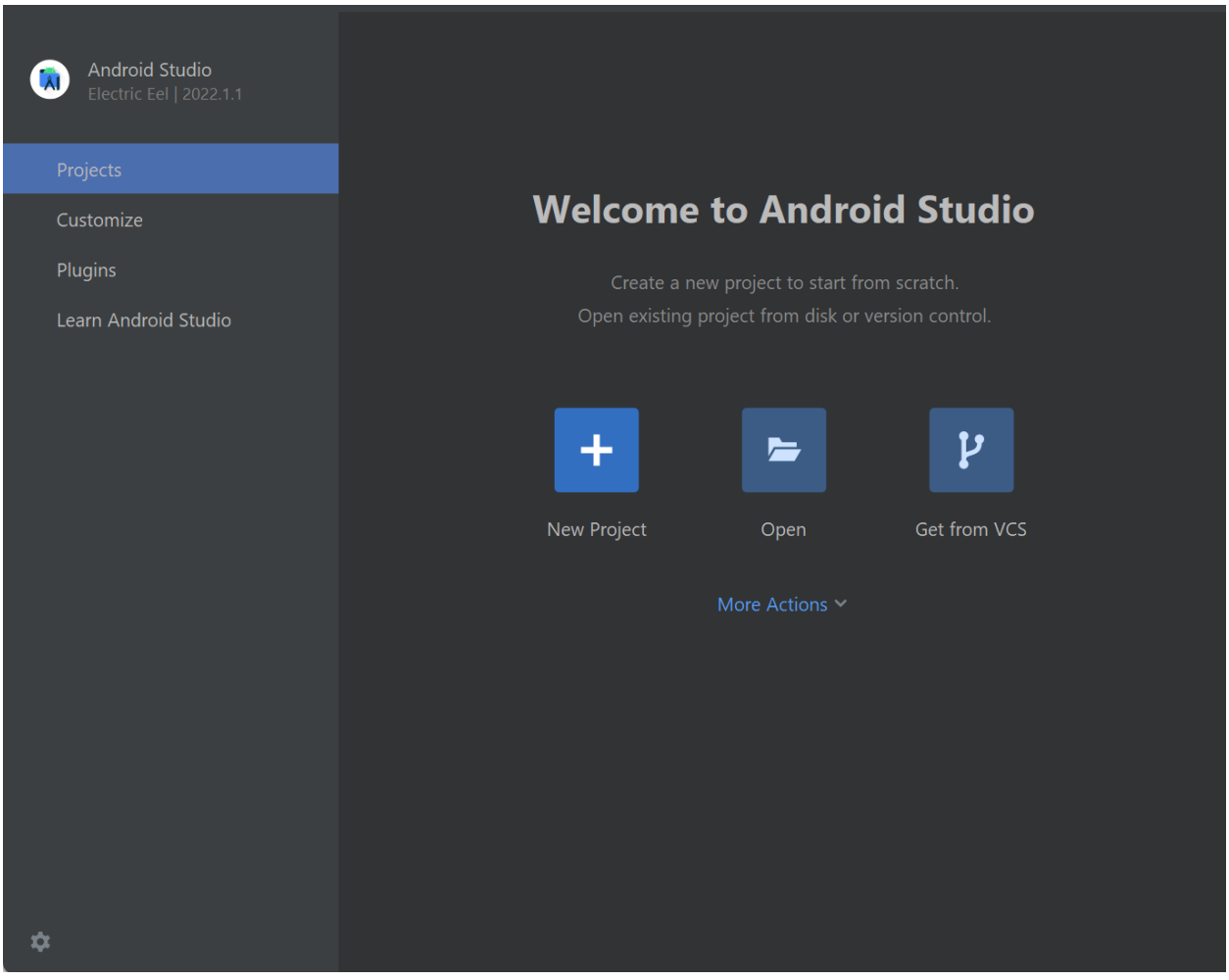


you chose, there is one main repository to focus on, FtcRobotController. This contains the main code which is uploaded onto the *Control Hubs* to run the code that we make.

2 – Android Studio Basics

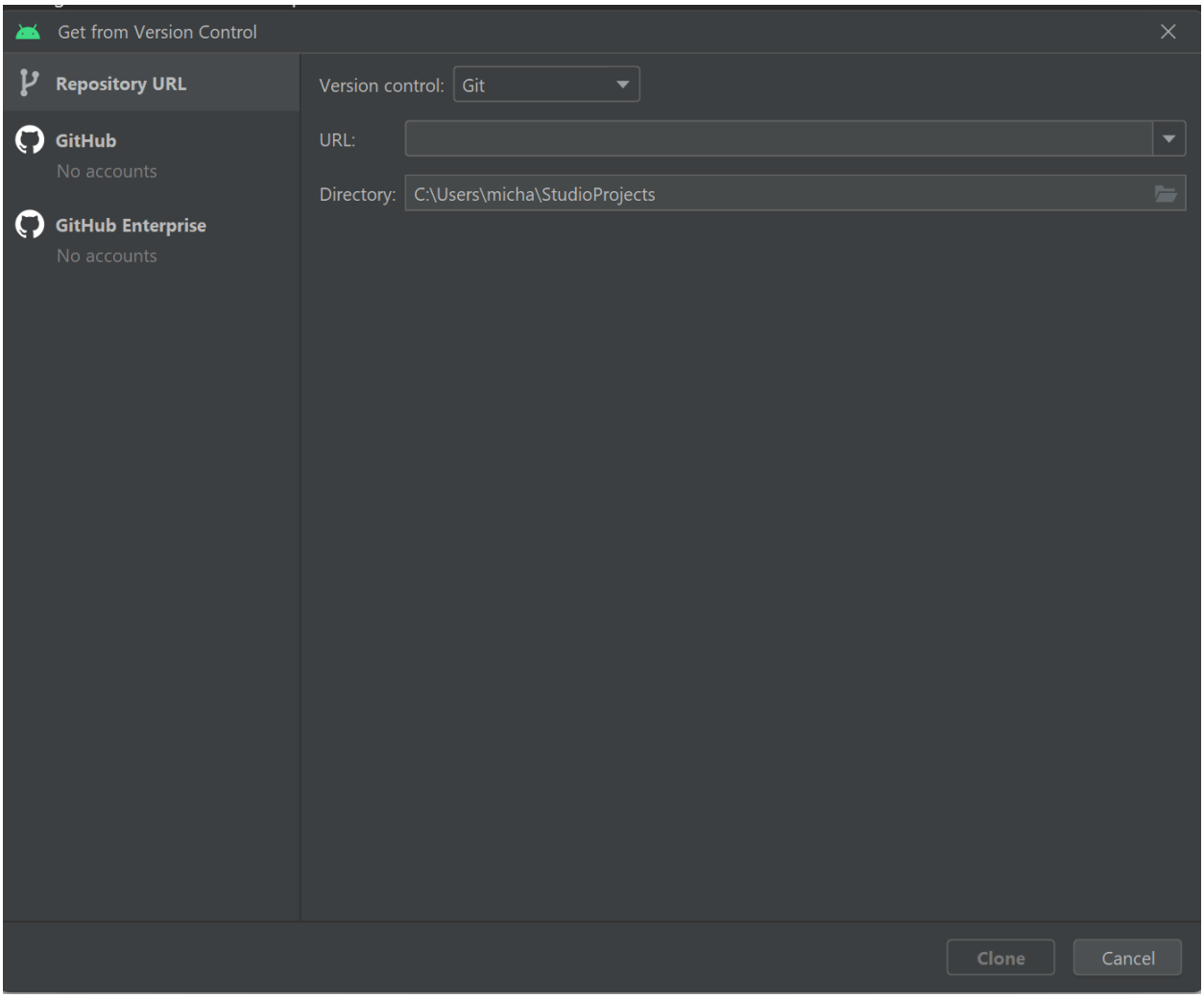
Downloading a Project from GitHub

Upon starting up Android Studio once completing setup, you should get the menu pictured below. If you don't have it, you may have missed a step in the Installing Android Studio section.



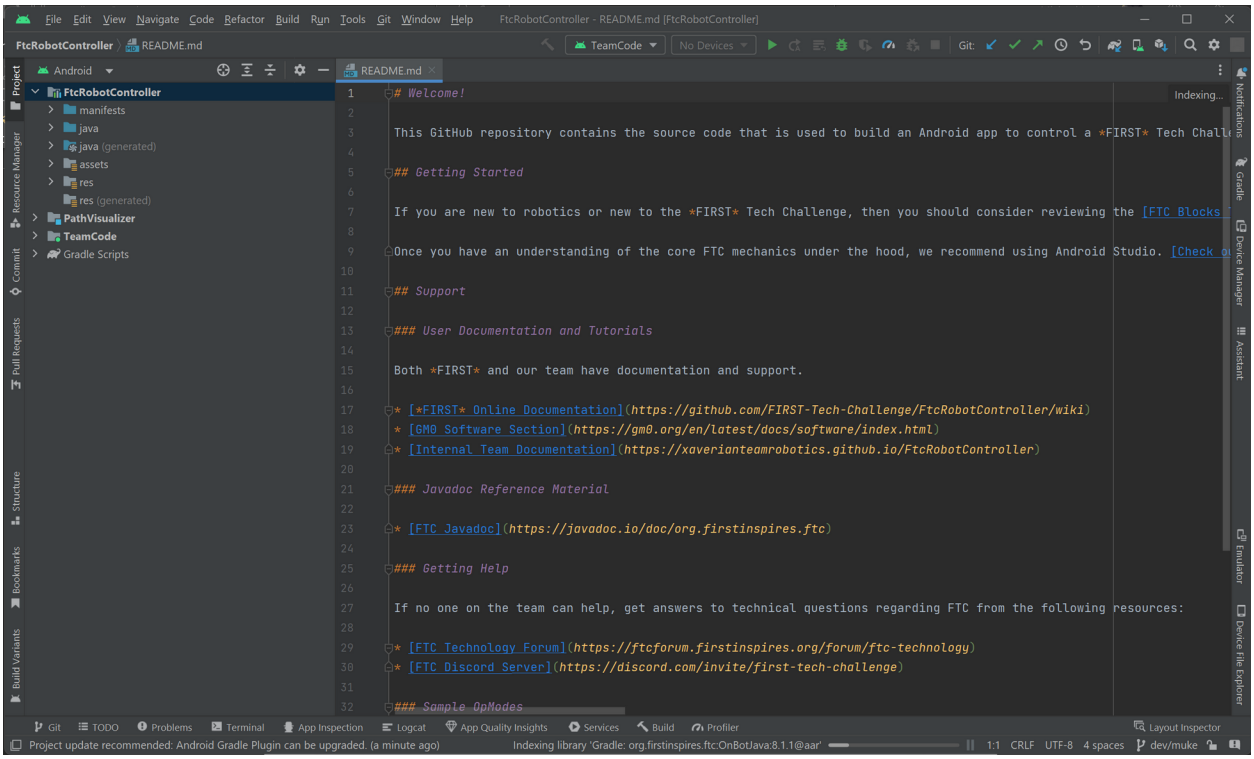
But let's explain this menu. On the left sidebar you can see the name **Android Studio** and the version. At the time of writing, the version is 2022.3.1, which has the code name **"Giraffe"**. Below that are the options *Projects*, *Customize*, *Plugins*, and *Learn Android Studio*. The only menu we care about for downloading a project from GitHub is the *Projects* menu. In the *Projects* menu, you can see three big buttons, *New Project*, *Open*, and *Get from VCS*. It's important to know that

GitHub is what we call a *Version Control System (or VCS)*. Now that I've put that there, you may be able to infer that we want to click the *Get from VCS* option. Once you click that option, you should get this menu.



Here you can put a URL to a special part of the project which is the `.git` file. Every project has a `.git` file. To find it, take the URL of the repository and add `.git` to the end. So, take the link `https://github.com/XaverianTeamRobotics/FtcRobotController` and add `.git` to the end, then put the new link in the `URL` field in Android Studio. Then click *Clone* at the bottom of the screen and Android Studio will download the project for you.

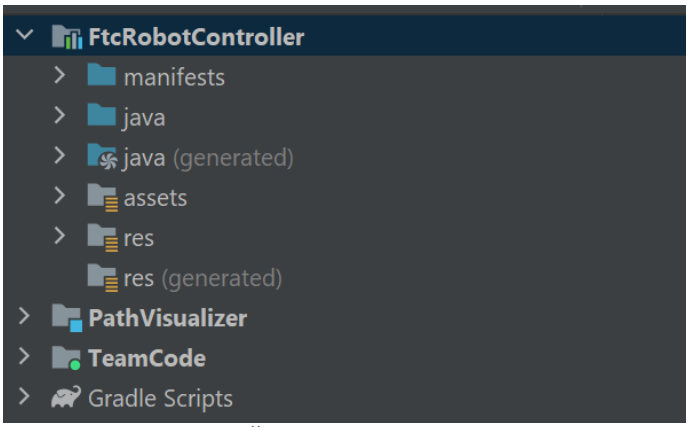
What am I looking at?



Here is the main screen when opening a project for the first time. The middle screen is the code, and the left sidebar has the list of files. Along all the edges are more “sidebars” that you can open to get more functionality. However, for the most part, this layout is all that you will use. At the top of the screen are actions that you can perform, like running the project, rebuilding code, or updating the project from GitHub. At the very bottom, you may see something like `Building Gradle Project` or `Indexing library`. Those are the running background tasks. Whenever you launch the editor, run the project, or really do anything, the background tasks needed will display down below.

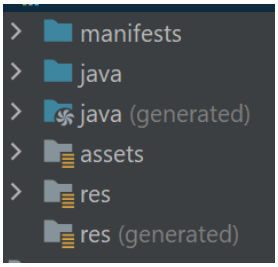
Navigating the File Explorer

To find files, you need to know how our project is organized. So, let’s look at the explorer itself. There are four main folders called *modules*. These three modules are *FtcRobotController*, *PathVisualizer*, and *TeamCode*.



FtcRobotController contains all of the code that you probably will not need to edit. This module contains all of the *internal* code that makes the application actually load the code that we make, but should not be changed under any circumstances unless you know exactly what you are doing. *PathVisualizer* may not have actually appeared for you because on my computer I may have a different version of our code. The third and most important is *TeamCode*. *TeamCode* is where our *team* stores our *code*, hence the name *TeamCode*.

In the *TeamCode* folder, you will see the following folders. The following folders are named *manifests*, *java*, *java (generated)*, *assets*, *res*, and *res (generated)*. I will quickly describe what these folders contain.

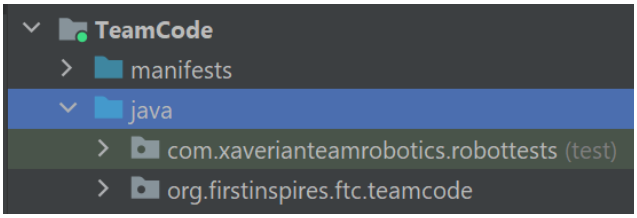


- *Manifests* – This contains technical info so Android knows what the contents of the module are
- *Java* – This folder contains all the Java code that we write
- *Java (generated)* – This folder contains all of the Java code that Android Studio has generated automatically and generally should be left alone
- *Assets*, *res*, *and res (generated)* – These three folders all contain files and other data that the app may need to achieve a desired function, however most of these files are user added.

If some of these are missing, it may be something that just happens on your computer. As always, ask someone else if you believe you messed something up or something looks wrong. The most important folder is the *java* folder, so it is absolutely critical to have it.

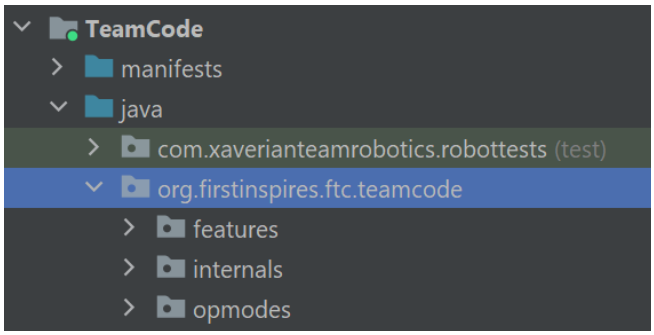
Looking at Our Code

Now that you know where our code is located, on Android Studio, let's explore how we keep our code clean and organized. Open the *java* folder.



Once it is open, you will see two more folders. One is *com.xaverianteamrobotics.robottests* and the other is *org.firstinspires.ftc.teamcode*. The first folder contains tests that allow us to

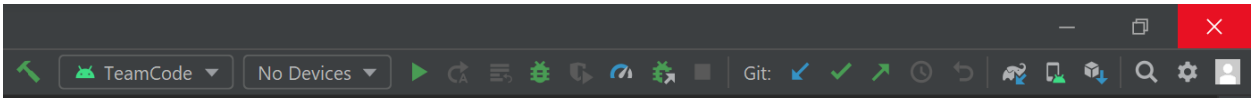
automatically ensure that a certain algorithm behaves as intended. The second folder contains our actual code. Exploring the *org.firstinspires.ftc.teamcode* folder shows us this:



Now we're getting somewhere! Believe it or not, we have entered the space where we can find actual code! It is organized into three sections: *features*, *internal*, and *opmodes*. The *opmodes* folder contains files called *OpModes*. For those who are new to the FTC competition, OpModes contain the code that actually makes the robot do things. The *features* folder contains different functions that can be recycled between *OpModes*. Examples of a *feature* include code for a drive train, an arm, or a grabber. The *internal* folder contains all of the code that we made that make everything happen behind the scenes.

Launching Code

Now that we know how to find code, let's launch some sample code. At the top right of Android Studio, you can find the "*Action Bar*". It contains, well, actions.



Here is a quick breakdown from left to right:

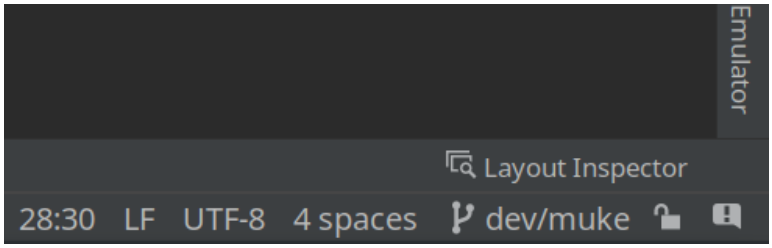
- Run Actions
- The hammer icon: Build – Click to build the code, but don't actually run it
- First drop-down: Action to run – By default it is *TeamCode*, meaning we want to run the *TeamCode* module.
- Second drop-down: Target device – The device to run the code on
- Play Button: Run – Run the selected action on the selected device
- Replay Icon (grayed out): Re-run – Re-run the selected action on the selected device
- The next five icons () are not necessary
- Stop Icon (grayed out): Stop the running code
- Git Actions

- Blue arrow: Update project – Download changes from the repository
- Green Check-mark: Commit – Commit any changes to the local copy of the repository
- Green Arrow: Push – Upload any commits that are not on the upstream repository
- Clock (grayed out): View history – See the history of the repository
- Backwards-facing arrow (grayed out): Rollback – Undo the changes to the last commit
- Settings
- You can disregard the other options () as they don't really matter that much.

Now that we know how the action bar works, if you connect a *Robot Controller* and make sure that it is powered and turned on, once the light turns green you will see it in the target device drop-down. Once you do, click the play button, and at the bottom right you should see the following message:

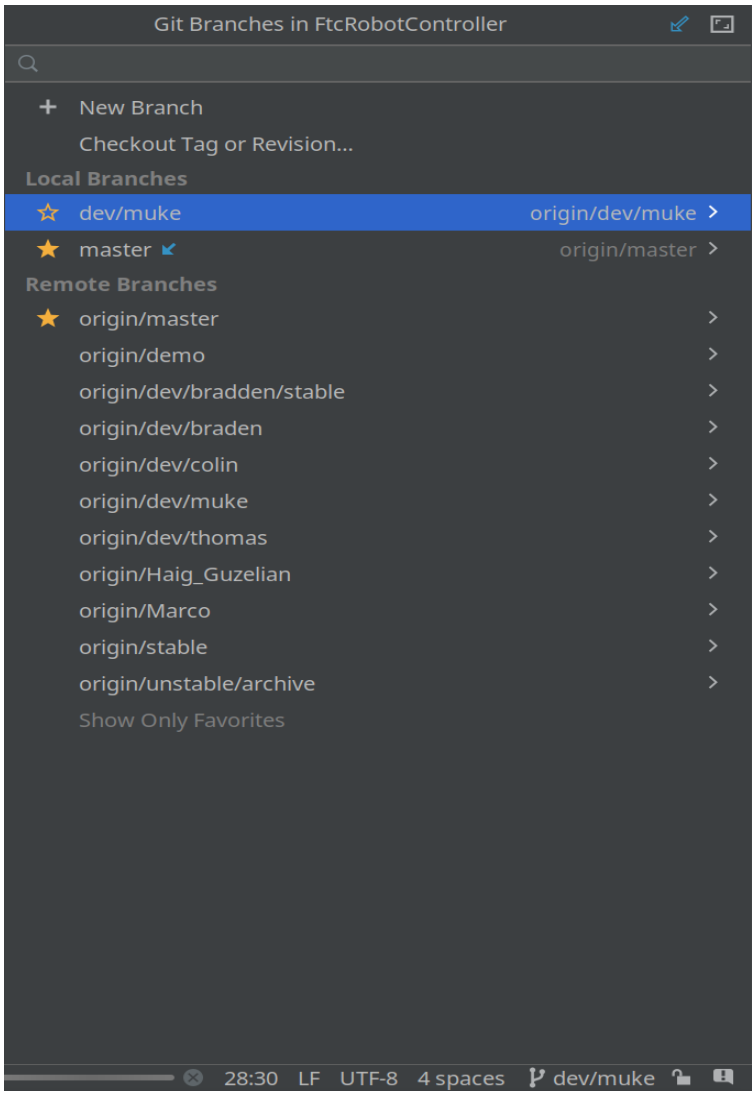
This message means that your computer is now building the project to get it ready. It may take a while, sometimes up to 2 minutes. After it's done, it will change to "*Install*", and after 30 seconds you should see a notification saying *"Launch Succeeded"*. Once you see that, the launch has….*succeeded*.

Selecting Branches

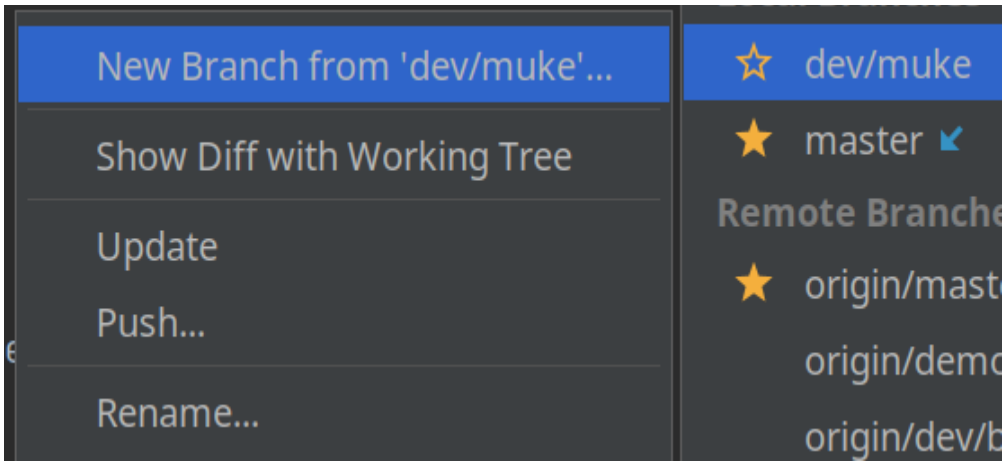


At the bottom right of Android Studio, you can see your selected branch. For me it says "dev/muke" but for you it should say "master". Click on it to open up the branch menu.

The branch menu displays each of the branches, separated by two categories: *Local Branches* and *Remote Branches*. Remote Branches do not exist on your computer, instead only existing on GitHub, which is the "remote" server in this case. Local Branches exist on your computer, and most often times correspond to a Remote Branch. For example, the local branch "master" corresponds to the remote branch "origin/master".



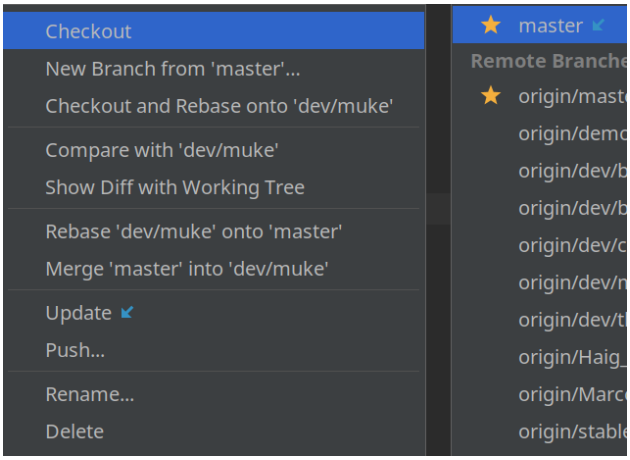
Upon clicking on any of the currently active branch, you get these options:



Here are the options explained:

- New Branch from '…': Create a new local branch using the current branch as the "template". The new branch will be an exact copy of the current.
- Show Diff with Working Tree: Don't worry about this
- Update: Receive any commits from the corresponding remote branch, if one exists. A blue arrow facing downwards next to this button indicates there is something to update.
- Push: Upload any commits to the corresponding remote branch, or create a remote branch if one doesn't exist. A green arrow facing upwards next to this button indicates that commits are ready to be pushed.
- Rename: Rename the branches

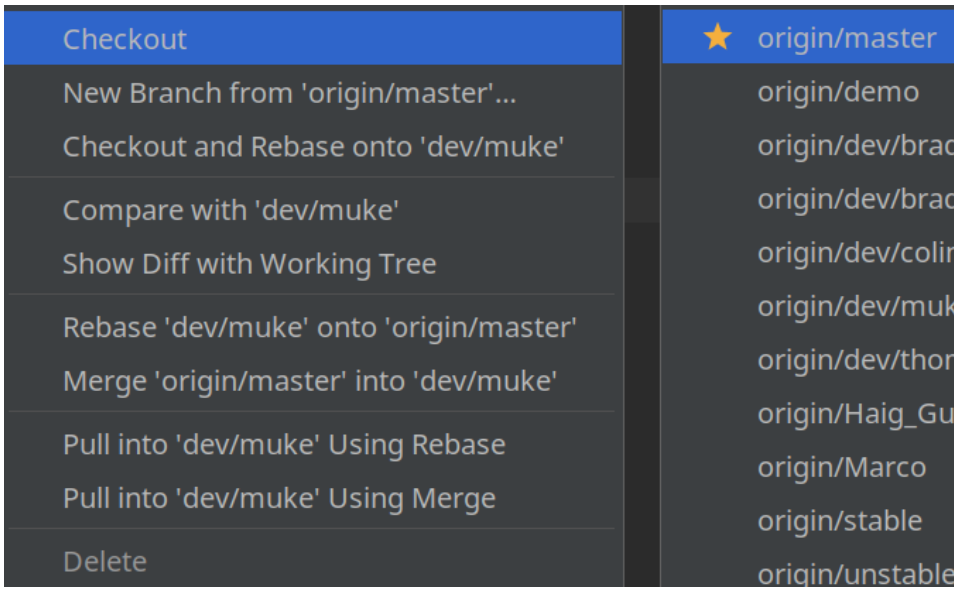
The set of options are different if you select a local branch different to the one you are working on.



- Checkout: Set the branch to be the currently active branch. Note that any changes made in one branch will not carry over to another
- Checkout and Rebase onto '…': Do not worry about this
- Compare with '…': Show the difference in commits between the currently active branch and the selected branches
- Rebase '…' onto '…': Do not worry about this
- Merge '…' into '…': Copy any commits from the currently active branch to the selected branch.

- Delete: Deletes the branch

The options for a remote branch have many more options. However, the only option you should use is “Checkout” to reduce the possibility of messing something up. Note that “Checkout” in this instance would create a copy of the branch as a Local Branch before setting it as your active branch.



So what you should do is create a new branch from master, naming it “dev/” and your name, so “dev/tom” or “dev/colin” are some examples.

3 – Java Basics

Introduction

You may have many questions, like *"What is a Java?"*, "*How do I Java?"*, or "*How does this guy know all the questions I'm asking?"*. Java is ~~not that~~ simple. Java is based around the mentality that everything is an *object*, or a philosophy known as *Object-Oriented Programming*. Think of an object like your phone. Literally, your phone is an object. Your phone has different apps. Your phone also has different properties like a phone number and a serial number. In Java, we could consider your phone as a *class*, which is a type of *object*. If your phone is a *class*, then the apps on it are *methods* and your phone number would be a *field*. *Methods*, like the apps on your phone, contain pieces of code that you can run. *Fields* are another way of saying *variable*, or some number or value that is necessary for a *method*. For those math guys, *methods* would be like $f(x)=x^2$, whereas a *field* would be $x=5$. Also, when writing, at the end of every sentence you type, you must put a period at the end. In Java, it's similar. You have to put a semicolon (;) at the end of every line otherwise Java will think that you never ended your "sentence".

Data Types

Java is not just about numbers. To separate numbers from letters, variables have things called *types*. Whenever you declare a variable, you need to declare what *type* it is. This prevents you from doing something adding a number to a letter, or doing other things that could cause an error. A *type* can either be what is called a *primitive* or it can be a *class*. *Primitives* are just a fancy way of saying a *class* that Java has built in for us. Some examples are:

- int – An integer (any number between -∞ and +∞ without a decimal point)
- float – Any number with a decimal point
- double – Same as float, but can do basic arithmetic better
- boolean – True or False
- String – A string of letters

We can also use our custom classes as data types, and use *instances* of those classes as the value for our variables. If you use the phone analogy from the

previous section, think of the model of phone you have as a type. For example, I have a Google Pixel 7. If Google Pixel 7 is the "type" of our "variable" then my specific phone would be in *instance* of that *type*. Let's also say that you gave me an iPhone 14 and told me that it was my phone. I would give you an error because I can't have an iPhone 14 because I only want a Google Pixel 7.

Creating a Variable

With that explanation out of the way, now would be a good time to show you how to create a variable. There are two ways we can do it. First, we can declare the variable and assign it a value all in one line, or we can just declare it and assign a value later. When we create a variable, we need to think about three things: 1) What will the type of my variable be? 2) What will I name my variable? 3) Do I want to assign it a value right away? Once you know the answers to all these three questions, we can create a variable. I asked those questions in a specific order, because let me answer it like this *"I want to create a variable with the type of **double** that is named **myVariable** and set it **equal to** a value of **11.7**"*. Notice what I put in bold. If we put them together, we get how to declare a variable:

```
double myVariable = 11.7;
```

That was easy! Now, let's create another one, but these time, we won't assign a value.

```
double myVariable;
```

Creating variable is very easy and is one of the basic building blocks to learn Java, so it's important to understand the process early on.

Methods – Creating and Using Them

Methods (sometimes called *functions*) allow us to create blocks of code which can be reused in many places without having to copy and paste. This allows us to change some problematic code in one place. *Methods* have 4 different properties you can change, which are the *access modifier*, *return type*, *name*, and *parameters*. The *access modifier* tells us where we can access this function. It can either be *public* or *private*. *Public* means we can access this function anywhere.

Private prevents anything outside of the current class from accessing this method. The *return type* allows a method to *return* a *variable*. For example, say we had a function that added 2 and 1. The *return type* would be *int*, because 2 and 1 are whole numbers. If we don't want to return anything, then we can put *void* as the *return type*. *Parameters* are like variables. They allow a method to take in a value. So, we can say that we want to create a function that triples a parameter. Let's create an example of that function below:

```
public int triple(int x) {
     return x * 3;
}
```

Let's go over that. First, we set the *access modifier* to *public*. Next, we specified a *return type* of *int*. Then we decided to name it *triple*. We gave it the parameter *x* with type *int*, and then we return triple of x. But how can we use this function? It is very simple, all we must know is the name, what parameters it needs, and what it returns.

```
int myVariable = triple(3);
```

We can also give a variable as a parameter:

```
int myNumber = 3;
int numberTripled = triple(myNumber);
```

In this case, *numberTripled* will be 9, because 3 tripled is 9.

Let's make a function that has no *return type*:

```
public void doNothing() {


}
```

Now this function may seem useless, but functions that return *void* can be used to do things like save something in a file, print out a message, or drive a motor to a certain power.

Combining our Knowledge – Classes

Classes are the last thing that you need to learn before we can make our first *OpMode*. Classes are like methods. We can provide them with an *access modifier* and a name, but in the place of a *return type* we just put the word *"class"*. Here is how we define one.

```
public class MyClass {

}
```

What we've done is we created a class that is *public* (we can use it anywhere) and named it *MyClass*. It is important to note two things about naming classes. First, we must make sure that the name of the *class* matches the name of the *file*. Also, classes cannot *begin* with a number, math operator (+, -, =, etc.), hyphen (-), or underscore (_). But right now are class does absolutely nothing besides take up space on our computers. But let's make add one method to our class, specifically the *triple* method we made earlier. To add it to our class, we simply add the method within the two curly braces, like so.

```
public class MyClass {
     public int triple(int x) {
          return x * 3;
     }
}
```

To use this method, we have to do two things. First, create an *instance* of the class, then call the method.

Using Classes

Image you are trying to call someone on a phone. Let's image a phone as a class with one method: *call*. You can't call someone on a phone that doesn't exist, so you need to get a phone. In Java, we can create an *instance* of a class using the keyword *new*. So in our example class we made above, we could create a new *instance* with this snippet of code:

```
new MyClass();
```

What we did here is say we want to create a new *instance* of MyClass. We will explain later why the two parentheses at the end are needed.

But wait! How do we store this class in a variable? Easy. MyClass can be used as the type of a variable, so we could have this variable to store our precious instance of the class.

```
MyClass myClass = new MyClass();
```

Here Java breaks this statement down into 3 basic parts:

- We create a variable named *myClass*
- This variable has a type of *MyClass*
- We want to store a new instance of *MyClass* in the variable.

We can then call our function like this:

```
int tripledNumber = myClass.triple(3);
```

4 – OpModes

Excerpts taken from online documentation

What are OpModes?

OpModes, short for *OperationModes*, are the starting points of the program. There can be as many as one desires, and are selected from the driver station. They are defined in the TeamCode module in the opmodes folder. They are defined as a class that extends *OperationMode* and either implements *TeleOperation* or *AutonomousOperation*, depending on whether or not the code should be run in teleop or autonomous.

OperationModes consist of two main functions: *construct()* and *run()*. The *construct()* function is called once when the OperationMode is selected and the user pressed *Init* button the Driver Station.

The *run()* function is called repeatedly once the user presses the *Play* button until the user presses the *Stop* button.

See a blank sample on our online documentation

The *Construct* Function

The *construct* function in an OpMode serves as a "getting ready" phase. Here you might want to set any servos to their starting position, set the values of some variables, start the camera's visual processing, and much more. The idea is it should contain anything you want to do once before the game actually starts.

The *Run* Function

The *run* function in an OpMode is the actual "meat" of the OpMode. The *run* function should follow a "short but sweet" mentality in that it shouldn't take too long to run. If the function never quits, then background tasks can't be run and

the app will crash. To avoid this, we avoid adding any loops that would last forever.

EXERCISE – Creating a blank OpMode

Your task is to create a blank OpMode named MyOpMode, based off of the sample above. The answer will be on the next page if you are stuck.

----------SOLUTION ON NEXT PAGE----------

EXERCISE ANSWER

First we need to write the first line of the class, often called the header.

```
public class MyOpMode extends OperationMode implements
     TeleOperation {
```

This indicates we want to create a class named MyOperationMode which extends from OperationMode and is a TeleOperation.

Next we need to add our construct and run functions.

```
    public void construct() {


    }


    public void run() {


    }
```

Here is the whole thing.

```
public class MyOperationMode extends OperationMode implements
     TeleOperation {
    public void construct() {


    }


    public void run() {


    }
}
```

5 – Accessing Hardware Devices

What Hardware Devices are Available?

The hardware devices available include but are not limited to:

- Motors
- Servos
- Controllers
- Distance Sensors
- Cameras
- Color Sensors
- etc.

How do we access these?

There is a central class called *Devices* that can be used to access these devices.

Check out the following sections on our online documentation.

- https://robotics.xbhs.net/apis/Gamepads
- https://robotics.xbhs.net/apis/Servos
- https://robotics.xbhs.net/apis/motors

----------WORK IN PROGRESS----------

VISIT robotics.xbhs.net FOR FURTHER READING