
GirlsOfSteelDocs

Release 0.0.1

Feb 21, 2022

Helpful links:

1	WPI Documentation	3
2	wpilib Source Code	5
3	CTRE Documentation	7
4	REV Documentation	9
5	Electronics Status Lights	11
6	SnobotSim Documentation	13
7	Latest Robot Code	15
8	FRC Programming Environment	17
9	IntelliJ Setup	21
9.1	Installing IntelliJ	21
9.2	Installing Plugins	21
9.3	Navigating IntelliJ	21
9.4	FRC Plugin	23
10	Style Guide	25
10.1	Class Names	25
10.2	Function Names	25
10.3	Variable Names	25
10.4	Braces and Spacing	26
11	Direction Conventions	27
11.1	Motor Conventions	27
11.2	Digital Input Conventions	27
11.3	Field Coordinate System	27
12	General Guidelines	29
13	Git-Flow Introduction	31
13.1	Scenario: I want to add a new feature	31

14 Code Review	33
14.1 What To Look for	33
14.2 Requesting changes	34
14.3 Merging Changes	34
15 Github Issue	35
16 Git	37
16.1 What is git?	39
16.2 Git Command Line	39
17 Installers	43
17.1 Software Installers Kept on USB Drives	43
17.2 Team Laptop Setup	43
17.3 Upgrading Firmware and Setting Up Electronics	44
17.4 Changing the Team Number	46
18 Sensors	47
18.1 LIDAR-Lite	47
18.2 Navigation Sensors	49
18.3 Tuning PID and Related Parameters	50
19 Code Labs	53
19.1 Gitflow Lab	53
19.2 Simulator Lab	54
19.3 Custom Shuffleboard Widget Lab	57
19.4 Calculator Lab	59
20 Monorepo intro	65
20.1 Creating a New Robot Project	65
20.2 Create New Shuffleboard Plugin	71
21 Indices and tables	73

Follow the steps in *Environment Setup* to install all of the required tools, then read through the rest of the documentation for details about our coding standard and workflow practices.

The links below are commonly used FRC programming references.

CHAPTER 1

WPI Documentation

CHAPTER 2

wpilib Source Code

CHAPTER 3

CTRE Documentation

CHAPTER 4

REV Documentation

CHAPTER 5

Electronics Status Lights

CHAPTER 6

SnobotSim Documentation

CHAPTER 7

Latest Robot Code

FRC Programming Environment

Joseph Jackson

The official FIRST documentation for installing the FRC programming tools is a bit convoluted and time-consuming. I've created a streamlined process for Girls of Steel by:

- downloading and unpacking all the files ahead of time,
- focusing only on Java and the third-party libraries we rely on,
- writing scripts to automate many of the steps on the Mac, and
- adding steps for using GitHub to share code.

1. Obtain a USB drive with all the installers

- See Joe or the Programming Lead to borrow one. The contents of the drive are documented on the [[Software installers kept on USB Drive]] page.

2. Follow the steps specific to your platform of choice

1. macOS (the red pill)

1. Open the `Mac` folder on the USB drive

- There's no need to make a copy of the folder unless others are waiting for a flash drive

2. Drag `Visual Studio Code` to your `Applications` folder

- As a shortcut, you can drop it on the `Applications alias` found in the `Mac` folder
- If prompted, choose to `Replace` any older version

3. Drag `GRIP` to your `Applications` folder, same as above

4. Launch (double-click) the `WPILib-Installer.command` script

- A Terminal window will open to show the progress as the script runs
- Confirm there aren't any error messages; If so, show them to Joe

- Close the Terminal window after you see the [Process completed] message
 - If you see an error “cannot be opened because the developer cannot be verified”, take these steps:
 - Open System Preferences from the Apple menu
 - Go to the Security & Privacy preference pane
 - Switch to the General tab
 - Click the Open Anyway button, if present
 - Back in the error dialog, click Cancel
 - Let the installer script finish, but be sure to run it again to complete any canceled steps
5. Launch (double-click) the CTRE-Installer.command script
 - A Terminal window will open to show the progress as the script runs
 - Confirm there aren’t any error messages; If so, show them to Joe
 - Close the Terminal window after you see the [Process completed] message
2. Windows (the blue pill)
 1. Open the Windows folder on the USB drive
 - There’s no need to make a copy of the folder unless others are waiting for a flash drive
 2. Run (double-click) the WPI library installer: WPILibInstaller_Windows64-*.exe
 - It is recommended to install for All Users when prompted
 - In the main installer window, click on Select/Download VS Code, then Select Existing Download
 - Select the OfflineVsCodeFiles-*.zip from the Windows folder of the USB drive
 - Make sure the checkbox for Visual Studio Code is checked
 - Click Execute Install and wait for the installer to finish
 3. Run the National Instruments FRC Game Tools installer
 - Open the ni-frc-2020-game folder found in the Windows folder of the USB drive
 - Run install.exe to begin the installation
 - Review and accept the license agreements to continue
 - Take all the default installation options by clicking Next buttons
 - When prompted to log into your NI User Account, use the Create account link, if necessary
 - **NOTE:** When the NI Licensing Wizard prompts you to activate the Vision Development Module, use the X button in the upper right to cancel out of the window
 - Reboot when the installer offers the Reboot Now option
 4. Run the FRC Radio Configuration installer: FRC_Radio_Configuration-*.exe
 - Accept any default installation options
 - You will be prompted to install a network packet utility as subtask of this installer
 - Be sure to choose I want to reboot later, as instructed by the installer
 5. Run the Git installer: Git-*.exe
 - There are many, many pages of options presented to customize your Git environment

- Just accept all defaults and click through the pages to complete the installation
6. Run the GRIP installer: `GRIP-*.exe`
 - Accept any default installation options
 - Note that the GRIP tool is automatically launched after installation
 - Quit out of GRIP before continuing
 7. Run the CTRE Phoenix installer: `CTRE Phoenix Framework *.exe`
 - De-select the Hero C# and LabView options, leaving the others enabled
 - This provides support for the Talon SRX motor controllers
 8. Run the Spark Max installer: `spark-max-client-setup-*.exe`
 - Includes a utility and libraries for the Spark Max motor controller
 9. (Optional) Install the Limelight firmware flashing tools
 - You can skip this for personal laptops (unless you work with the Limelight often and want it)
 - All team laptops should have this installed
 - Run the USB Flash Driver installer (it's for the embedded Raspberry Pi): `rpiboot_setup.exe`
 - Copy the Balena Etcher imaging tool to the laptop: `balenaEtcher-Portable-*.exe`
 - You can leave it in your Downloads folder or create a new folder such as: `C:\Program Files\Limelight Image Tool\`
 - Right-click on *your copy of* the `balenaEtcher*` file and create a shortcut to it on your Desktop
 - Copy the latest image file to the same location on your laptop as the imaging tool: `LL*_RELEASE.zip`
3. [Mac & Windows] Essential VS Code settings
 1. Start Visual Studio Code if it's not already running
 - On the Mac, it is in Applications
 - On Windows, you **must** use the desktop icon `FRC VS Code 2020` to start the correct version of VS Code
 2. Bring up the Command Palette by pressing:
 - On the Mac, Command-Shift-P
 - On Windows, Control-Shift-P
 3. In the palette, enter the command `WPILib: Set VS Code Java to FRC Home` and press return or select from the list of matching commands
 4. First-time import (clone) of projects from GitHub
 1. Run through this process once per season to establish access to this year's GitHub code repository
 - If you've done this during the preseason, there's no need to do it again
 2. Start Visual Studio Code if it's not already running
 - On the Mac, it is in Applications
 - On Windows, you **must** use the desktop icon `FRC VS Code 2019` to start the correct version of VS Code

3. Bring up the Command Palette by pressing:
 - On the Mac, Command-Shift-P
 - On Windows, Control-Shift-P
 4. In the palette, enter the command `Git: Clone` and press return or select from the list of matching commands
 5. Enter in the URL for this year's GitHub repository:
 - `https://github.com/GirlsOfSteelRobotics/2020GirlsofSteel.git`
 - (Adjust the URL above for the current season, if necessary)
 6. Select a location to create a new folder for the repository
 - Your `Documents` folder is a reasonable place to put it
 7. If prompted to open the repository, cancel out with the `X` button
 - **Only open individual VS Code projects** found inside the repo instead of opening the entire repo!
 8. To open a project, use the `Open Folder` option from the VS Code Welcome page or from the `File` menu
5. Create a personal GitHub account
- <http://github.com/>
 - If you already have one for school, there's no need to create another
 - Ask Joe or the Programming Lead to invite your GitHub account to the Contributors team for Girls of Steel
 - See your e-mail to accept the invitation
 - When finished, you can push changes to GitHub to share them with the rest of the team
 - VS Code will prompt you for your GitHub username and password the first time you push code
6. Installing Third-Party Libraries When using third-party hardware on the robot, the project needs to have the associated libraries installed.

IntelliJ is an IDE specifically built for writing Java software. Because it is a first class Java editor, it offers much better support than the versions of VS Code that WPI provides us.

9.1 Installing IntelliJ

The easiest way to install and update IntelliJ is by download Jet Brains [Toolbox](#) application. This will alert you whenever there is a new version, and allows you to easily download their other IDEs if you ever want to dabble in Python or Web Development

Once Toolbox has been installed, open it and select the “IntelliJ Community Edition”, and run the installer

9.2 Installing Plugins

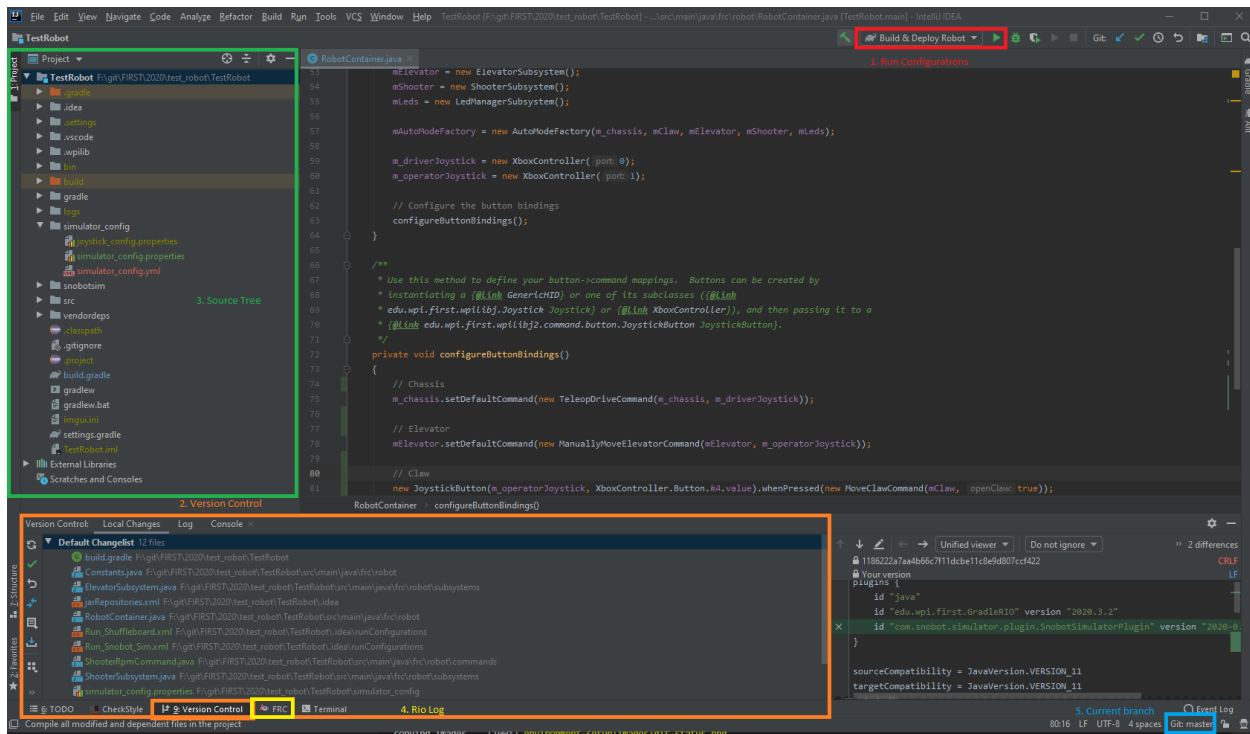
To get the best experience out of IntelliJ, we need to install some additional plugins.

To install a plugin, go to “File -> Settings -> Plugins”. Select the “Marketplace” for the following plugins, and install them. Once they are all done, you will need to restart IntelliJ

- FRC
- Checkstyle-IDEA

9.3 Navigating IntelliJ

When you open IntelliJ, you will see a screen that looks like this.



9.3.1 1. Run Configurations

The options in this dropdown allow you to do all of the important things we could do in VS code like deploying code to the robot, opening the RioLog, as well as allowing you to run the simulator.

Note: unlike VS code, you have to select what you want to run AND THEN press the green play button, they don't auto run when selected like in VS Code.

9.3.2 2. Version Control

The version control tab allows you to see your current changes, easily look at diffs at a quick glance, and kick off the commit/push process. There are a lot more options available in the "VCS -> Git" toolbar on the top.

9.3.3 3. Source Tree

This is where we can see all of the files for our project. Unlike VS Code, you have to manually open folders that have only one sub-folder in them which can be a little bit more tedious. Our robot code is located under the "src/main/java/frc.robot" folders

9.3.4 4. FRC Tab / RioLog

When you are connected and communicating with the robot, the RioLog will be displayed here

9.4 FRC Plugin

The IntelliJ FRC plugin is an open source project to provide a lot of helpful features for us when we are developing software for FRC robots. It eases the creation of Subsystems and Commands by filling in a lot of the boiler plate code for use.

9.4.1 Adding a Subsystem

Right click on the package where you want to put the subsystem (almost certainly `frc.robot.subsystems`), and click “New -> FRC -> Subsystem”

The subsystem it creates is very sparse. At the very least, you will probably want to add a `periodic` function

9.4.2 Adding a Command

Right click on the package where you want to put the command, and click “New -> FRC -> Command”. You will be given the option to select what subsystems this command requires to save you some typing.

Our styleguide is enforced by a [Checkstyle](#) configuration file. This ensures that we all follow the same conventions for variable names, class names, spacing, brace location, and many other things.

While the styleguide enforces many more rules than outlined below, these are some of the major ones to be aware of.

10.1 Class Names

Class names should be UpperCamelCase, i.e. `ShooterSubsystem`

10.2 Function Names

Function names should be lowerCamelCase, i.e. `void setSpeedAndSteer(double speed, double steer)`

10.3 Variable Names

- Member variables (variables that belong to a class) should start with the *m_* prefix, then be lowerCamelCase after that, i.e. `TalonSRX m_leftDriveMotor`
- Local variables and function arguments should be lowerCamelCase
- Constants should be `static final`, and be named `UPPER_SNAKE_CASE`, i.e. `public static final double DEFAULT_SHOOTER_RPM = 5000;`

10.4 Braces and Spacing

- All keywords (i.e `if`, `switch`, `for`, etc) and math operands (`+`, `-`, `<`, etc) should be separated by a space

```
if (x > 5) // Good  
if(x>5)  // Bad!
```

- Indentation should be done with 4 spaces. Never 2, never with Tabs
- We use [K&R Braces](#), meaning that all opening braces start on the same line.

```
String myGoodFunction(int x) {  
    if (x > 5) {  
        return "Greater Than 5"  
    }  
    else if (x == 5) {  
        return "Equals 5";  
    }  
    else {  
        return "Less than 5";  
    }  
}  
  
String badFunctionButPJLikesItMore(int x)  
{  
    if (x > 5)  
    {  
        return "Greater Than 5"  
    }  
    else if (x == 5)  
    {  
        return "Equals 5";  
    }  
    else  
    {  
        return "Less than 5";  
    }  
}
```

Direction Conventions

11.1 Motor Conventions

Motors should be set up such that moving in the “positive direction” should result in a green LED, and be set with a positive speed value. This may require inverting the polarity of the motor controller. This is especially evident for our tank drive set up, since one side is mechanically inverted.

The “positive direction” is defined such that it causes the mechanism to move in the direction that leads to us scoring or something of similar nature.

Examples Include:

- Driving the robot forwards
- Moving an elevator up to lift a game piece
- Spinning a shooter wheel in the direction that ejects the game piece

11.2 Digital Input Conventions

Any getters in the code or things sent to the dashboard should return `true` when the sensor is tripped. This is often the opposite of how the sensor behaves natively, so it must be inverted in the code.

Examples:

- `boolean getBallSensor()` should return `true` when a ball is detected, and `false` otherwise

11.3 Field Coordinate System

The field coordinate system matches the standard mathematical definition (which sucks).

Imagine the field is laid out such that the long dimension goes along the x axis, and the short dimension goes along the y axis. The origin point (0, 0) is located at the top left, while the bottom right is defined as (long dim, -short dim).

(0, 0)



Angles are measured from the X-Axis, increasing counter-clockwise.

→ = 0 degrees
 ↑ = -90 degrees
 ↓ = 90 degrees
 ← = 180 degrees

CHAPTER 12

General Guidelines

- In general, there should be a 1:1 match between mechanical subsystems and our software subsystems.
- Base commands should be broken down into extremely simple tasks. More complicated commands can be strung together with `ParallelCommandGroup` or `SequentialCommandGroup`, which should be represented in their own class
- Fully fledged autonomous modes can be represented as a command group, but should live in a different package than the rest of the commands
- Constants that need to be tuned such as PIDF values should use our *Properties* library. This allows us to quickly tune the value on Shuffleboard, and easily lock them once we are satisfied with the values. This dramatically reduces debugging time, since you don't have to rebuild and redeploy the code during tuning.
- There should be no "Magic Numbers" in the code. All motor/sensor ID's should be defined in *Constants.java*, and all other constants should be defined inside the class that is using them
- Always test your code with the simulator or the real robot before you declare your task "done" and ready for review.
- ALWAYS commit your code before you leave a meeting.

Git-Flow Introduction

git was invented to solve the problem of many different people working on a project at the same time. Frequently we are working on tasks that we cannot finish in a single meeting, that we want to share with our fellow teammates. However, we do not want to commit something to master that will cause things to break. We should strive to keep master in a 100% working, ready to deploy state.

One way to do this is to use a modified version of [gitflow](#), a very popular workflow used by many professional engineers and companies. In essence, whenever we need to implement a new feature, or fix a bug, or really make any kind of change, you create a branch off of master, do your work, then submit a “Pull Request” so we can pull your changes back into master.

Doing this allows each student / group of students working on a task to work in their own little sandbox. They can commit and push their changes so they can bounce between computers and meetings, all without affecting all of the people working on other tasks.

Using Pull Requests gives us the opportunity to review a change before it gets into master, to make sure it follows all of our coding conventions, the logic makes sense, and helps keep everyone knowledgeable about all of the things that are going on in our software.

13.1 Scenario: I want to add a new feature

13.1.1 1. Get Latest

Go to the toolbar and selecting “VCS -> Git -> Pull”

13.1.2 2. Create and Checkout a new branch

Click on the the git indicator on the bottom right. Click on “origin/master” under the remotes area, and select “New branch from selected”. Enter your branch name in the popup, and hit “Checkout”

13.1.3 3. Start working!

You can now start adding / modifying / deleting files.

4. Commit work your work Open the “Version Control” tab on the bottom of the screen. Navigate to the “Local Changes” tab. If there is anything under “Unversioned Files”, drag that to “Default Changelist”. Hit the green check mark, and type a commit message in the popup. Finally hit the “Commit” button. Alternatively, you could change it to button to “Commit and Push”

Note: You can commit and push as frequently as you want. You should ALWAYS push at the end of the meeting, even if you haven’t finished. This allows other students at the next meeting to continue working even if you aren’t there.

13.1.4 5. Push your final commit

When you are done with your feature, and have committed locally, you need to push it to GitHub. Depending on how you ran step 4, you may have been automatically pushing. To run a push, go the the VCS menu and run “git -> push”

13.1.5 6. Create a Pull Request

There are many ways to create a pull request, the two easiest being from GitHub’s website, or through intelliJ.

To create a PR on Github, open the repository in a web browser. Click on the branches tab, find your branch, and click the “Create Pull Request” button. In the new window you can write an additional message for the PR if you want, and then click the “Create Review” button.

To create a PR from intelliJ, go to “VCS -> git -> Create Pull Request”. Create a title with a quick synopsis of your feature, add any additional information in the description, and click OK

You should usually reference your *Github Issue*. in your description.

13.1.6 7. Wait for Review and Approval

A *Code Review* will happen for your code, which may require you making additional changes before it gets put into master. You can follow steps 3-5 to just as you did before to create the branch. Whenever you run a push, the code review will automatically get updated.

13.1.7 8. Run “Squash and Merge”

Once the PR has been approved, one of the leaders will run a “Squash and Merge” to get your changes into master. After the PR has been merged, your branch should be deleted.

13.1.8 9. Cleanup

It is easy to make mistakes when you are are starting your next feature, so in order to avoid some of the confusion we should clean up our now dead branch.

- Switch to master, and run a pull.
- Delete your local copy of your completed branch by going to it in the git indicator, clicking your branch under the “local” section, and selecting “Delete”
- Delete your remote copy by clicking on “origin/<your branch name” under the “Remotes” section

You are now on master, with the latest code, and no traces of the completed branch.

Note: In the video, I could not request changes because it was my own review. When you are reviewing someone else's code and find something they should fix, click that instead of "Comment"

Also note that the mac build hadn't finished, but you should wait for all of the builds to be green before running the merge.

Pull requests and code reviews are a chance for the team to look over your changes and make sure they are good and ready to be merged into master. Github will also attempt to build your code and make sure it can be merged into master, so we can mitigate mistakes and see exactly what the changes you are proposing are before before we give it the OK.

It is everyone's responsibility to review other peoples code, so we can increase the code quality, keep informed about all the new features going into the robot, and make sure nothing gross gets in.

14.1 What To Look for

When you are reviewing someone else's code, the first thing to thing you should do is look through it and make sure it makes sense to you.

- Are the class / variable names good? Will you be able to remember what `public IntakeCommand(boolean move) / new IntakeCommand(true)` means in a month when you are creating autonomous modes?
- If there is complicated logic, is it commented well enough so you someone else can come back to it later and quickly pick up what is going on?
- Is this change even necessary? Would it be better to modify an existing class to add more functionality instead? It is easy to get out of sync and duplicate logic which can be super confusing later.
- Are there any spelling errors or style guide errors?
- Note, not all comments have to be negative. You can always give props during reviews if someone did something really cool or in a new and innovative way.

14.2 Requesting changes

When you look at a review on GitHub, you can add comments, and if you think something should be changed, you block the pull request until those comments are addressed. In the “Files Changed” tab of the PR, you can click on a line and leave a comment, which will start your review. When you are done, you can click the green button, and optionally request that they fix the problems, or you can “Approve” the review.

14.3 Merging Changes

Once the review has been finished, all the CI builds pass, and one of the software mentors or student lead have given approval, the PR can be merged. We use the “Squash and Merge” approach, which takes all of your commits (if you have multiple), and combine it into one commit before merging it into master. This allows us to have a nice and neat linear history, which makes it easier to rollback changes if a bug pops up.

CHAPTER 15

Github Issue

We can use Issues on Github to track our task backlog. This is a good way to keep our TODO list in one place and assign tasks to the students that are working on them. We can also link issues to pull requests, so we can have traceability to what this PR is for and vice versa.

[Code](#) [Issues 2](#) [Pull requests 1](#) [Actions](#) [Projects 1](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

Label issues and pull requests for new contributors

Dismiss

Now, GitHub will help potential first-time contributors [discover issues](#) labeled with [good first issue](#)

Filters

Labels 9

Milestones 0

New issue

<input type="checkbox"/>	2 Open ✓ 0 Closed	Author	Label	Projects	Milestones	Assignee	Sort
<input type="checkbox"/>	<div><div>Make wrappers around CTRE / Rev libraries</div><div>enhancement</div><div>#30 opened 6 minutes ago by pjreiniger</div></div>						1
<input type="checkbox"/>	<div><div>Convert Odometry logic to use meters</div><div>enhancement</div><div>#29 opened 7 minutes ago by pjreiniger</div></div>						

We can use a KanBan board to visualize which tasks are not being worked on / in progress / in review / done.

GirlsOfSteelRobotics / 2020GirlsofSteel

<> Code Issues 2 Pull requests 1 Actions Projects 1 Wiki Security Insights Settings

Example Project
Updated 6 minutes ago Filter cards

To do

Convert Odometry logic to use meters

#29 opened by pjreiniger

enhancement

In progress

Make wrappers around CTRE / Rev libraries

#30 opened by pjreiniger

Review in progress

Smart speed controller

✓ #22 opened by 22avigada

Changes requested

Reviewer approved

Do

CHAPTER 16

Git



16.1 What is git?

(keywords mostly italicized.)

Git helps keep track of changes made to source code (well, anything, really, but people mostly use it for source code). A *repository* stores code files and all the changes (the *history*) that have ever been made to them. When you *clone* a repository, you copy both all of the code and the history to your computer. These two things—the code and the history—are separate.

(there’s also a big concept in git called “branching” that I’m completely ignoring for the purposes of this explanation.)

If you were the only programmer working on a repository, you could just *commit* over and over to save your changes to your own history, and there’s be no problems. The history on your machine would be the only history, and the only reason to *push* those changes and the changed code to github would be to have a copy of your work stored somewhere that’s not your own computer (this is a good idea, by the way—it keeps you from losing work if something happens to your computer). However, when many people are working on the same repository, there needs to be a way to combine everyone’s changes into one place. If I make 10 changes and you make 10 changes, both our code and our histories are now different, and we need to find a way to combine them.

upstream is the place your copy of the repository came from (your copy is *downstream* from it). A copy of a repository that’s stored on the internet or elsewhere on a network from your computer is *remote*. The *master* copy is the central copy with the central history (ideally with everyone’s changes). When you *fetch* from the remote master repository, you’re getting copies of all the new changes you don’t have yet on your computer (but you’re not applying them to the code! You’re just getting the changes in the history. Remember, they’re different). To apply the changes, you need to *merge* the changes/commits you’ve made with the commits that have come in. A *pull* fetches and then merges all in one step.

Merging takes the changes other people have applied and tries to automagically combine them with the changes you’ve applied. Sometimes this is easy. If you change just the first paragraph of this explanation and I make a change to the third paragraph, then the system can just apply them both; there’s no conflict between them. IF we both commit a change to this very line in this paragraph, however, git might not know how to combine them. Someone has to tell it. That’s where manual merging comes in, which will be explained elsewhere (and hopefully Claire will remember to link!).

16.2 Git Command Line

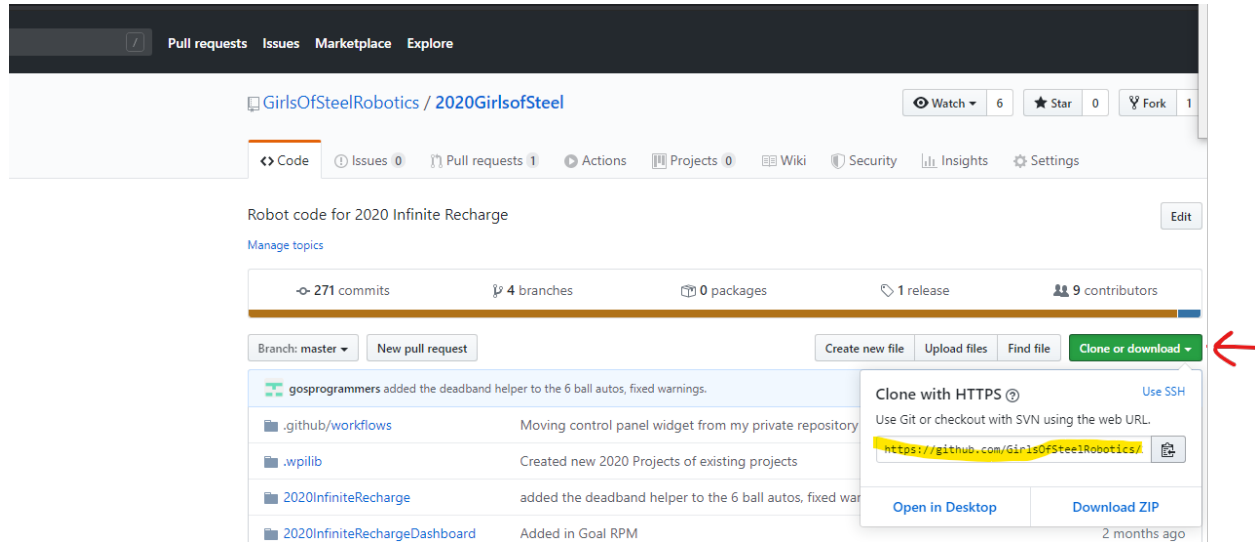
Because *Real Programmers* use the command line

Sometimes there are situations where doing things from the command line are infinitely easier than trying to use the GUI build into an IDE, so it is a good idea to get some of the common commands under your belt.

16.2.1 Clone a repository

`git clone <repository url>` is used to download a repository for the very first time. This command should be run in the directory where you want the code to live

The URL can be retrieved from GitHub, i.e



```
git clone https://github.com/GirlsOfSteelRobotics/2020GirlsofSteel.git
```

16.2.2 Get the latest from GitHub

TL;DR Run a `git pull` to get the latest

There are two ways to get the latest updates from Github, a `git fetch` and a `git pull`.

A `git fetch` will download the latest updates for all remote branches, but it will not update your local branch. This is an advanced feature that you will probably not need.

A `git pull` is syntactic sugar, which under the hood runs a `git fetch` followed by a `git merge`. This means that it will download everything from Github, and if you branch is tracking a remote branch, it will update pull in all of those remote changes on top of yours

16.2.3 Switching Branches

`git checkout <branch name>` will switch from your current branch, to the one you specify. You can use this when you are switching between what features you are working on, or to checkout a branch that already exists on the remote, like if you are picking up from where a teammate left off at the previous meeting.

```
git checkout master
```

Adding the `-b` option will allow you to create a new branch and immediately switch to it. You would do this when you are starting to work on a branch new issue

```
git checkout -b adding_shooter
```

16.2.4 Committing Files

TL;DR, do a commit by

```
git add *
git commit -m "My commit message"
```

There are two steps required when you run a commit. First, you must add your files to the 'index', then you can commit those changes. You can add all of your changes by running `git add *`, or add files individually by running `git add path/to/my/file.java`.

Then, you can run a `git commit`. Running that will bring up open a new window so you can enter in your commit message, or you can specify the commit message right away by running `git commit -m "My Commit Message"`

16.2.5 Pushing code to GitHub

Running a `git push` will publish your changes to GitHub.

If you are on a branch new feature branch which has never been pushed before, you might see an error like this. Simply copy the suggestion they give you, and run that command

```
F:\git\FIRST\2020\3504\2020GirlsOfSteel>git push
fatal: The current branch example_branch has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin example_branch
```

16.2.6 Look at your current status

Running a `git status` will show you all of the new/deleted/changes files you currently have that have not been committed. It will also tell you important information when you are in the middle of a merge that has conflicts.

```
F:\git\FIRST\2020\3504\docs>git status
On branch initial_working
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   source/conf.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   source/index.rst

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    source/coding-standard/
    source/environment-setup/
    source/github-workflow/
    source/rosie.jpg
```

16.2.7 Merging Branches

`git merge <branch name>` will merge all of commits from that branch with yours. This is important when your feature branch is out of date, and you need to get the latest stuff from master. When we merge a Pull Request, GitHub is running this command for us to get your features back into master.

Note that there might be conflicts when you run the merge. It is your responsibility to fix these issues and retest your code before we can approve your PR and merge it into master

```
git merge origin/master
```

When you are in the middle of a merge that has a conflict and decide you want to deal with it later, you can abort the merge by running

```
git merge --abort
```

16.2.8 Resetting

Sometimes in the course of working or merging, you realize that you messed something up, or you only have made debugging changes that you don't want to commit, so you can run a `git reset` to obliterate the changes.

NOTE: When you run a reset, your changes are gone forever, there is no 'undo' button

```
git reset --hard
```

17.1 Software Installers Kept on USB Drives

17.1.1 Required Installers

17.1.2 Helpful Commands

The `rsync` command is a fast way copy files, skipping any files that are already up to date at the destination. This command will update a USB drive mounted on `/Volumes/FRC` from an updated folder at `~/Desktop/FRC`:

```
rsync --itemize-changes --recursive --delete --times --omit-dir-times
--modify-window=1 --specials --links --exclude=.DS_Store --exclude=._.DS_Store
--exclude=/.Spotlight-V100 --exclude=.fseventsd --exclude='/System Volume
Information' --exclude=.Trashes --exclude=.TemporaryItems ~/Desktop/FRC/ /
Volumes/FRC
```

17.2 Team Laptop Setup

We have a number of team laptops that we try to set up in a consistent fashion. Here's a checklist for installation of everything except the FRC-specific programming software. (That's covered on the [\[\[Programming Environment Set Up\]\]](#) page.)

- Get a USB drive with installers
 - Most of the installers mentioned below are available on a USB drive that Joe maintains
 - URLs are provided below in case the USB drive isn't available or is out of date
- Install Windows 10 Enterprise
 - Joe can provide an installation ISO deployed to a USB drive
 - The initial user should be "Girls of Steel" with the standard laptop password and no hint

- Activate Windows (once every 120 days) <http://www.cmu.edu/computing/software/all/windows/activate.html>
 - * Must connect to a CMU wireless network or to a VPN that provides equivalent access
- Set the timezone to US/Eastern
- Set the 3504 background image
- Task bar: Remove Cortana and Store, add Command Prompt (cmd), Settings, Control Panel, and items installed below
- Install Firefox <http://firefox.com/>
 - Add bookmarks to the Bookmarks Toolbar for important resources
 - * GitHub <http://github.com/GirlsOfSteelRobotics/Docs/wiki>
 - * FRC Javadocs <http://first.wpi.edu/FRC/roborio/release/docs/java/>
 - * FRC Screen Steps <https://wpilib.screenstepslive.com/s/4485>
 - * RoboRIO <http://roborio-3504-frc.local/>
- Install Silverlight, needed for the RoboRIO web interface <https://microsoft.com/silverlight/>
- Continue to [[Programming Environment Set Up]] for further directions

17.3 Upgrading Firmware and Setting Up Electronics

This is a quick reference guide to upgrading and setting up the FRC electronics. Many of the utilities mentioned are Windows-only.

17.3.1 RoboRio

1. Upgrade firmware

1. Connect to the roboRIO with USB
2. In Internet Explorer (not Firefox, Chrome, or Edge) visit <http://172.22.11.2/>
3. Login with “admin” and blank password
4. Click `Update Firmware` button
5. Firmware is in “C:\Program Files (x86)\National Instruments\Shared\Firmware\cRIO\76F2”

2. Install the FRC image

1. C:\Program Files (x86)\National Instruments\LabVIEW 2016\project\roboRIO Tool\roboRIO_ImagingTool.exe
2. Assign team number if needed
3. Check the `Format Target` box if the image listed below it is newer than what’s currently installed

3. Add CTRE extensions to the RoboRIO web interface for managing CAN devices

1. The CAN device tab won’t appear in an imaged roboRIO until this step is completed
2. Connect to the roboRIO with USB
3. Launch the LifeBoat utility using the taskbar icon or from C:\Users\Public\Public Documents\Cross The Road Electronics\LifeBoat\HERO LifeBoat.exe

4. Switch to the FRC roboRIO tab
5. Press the Install Phoenix/Web-based Config button
6. Quit out of LifeBoat and restart the roboRIO

17.3.2 PCM, PDP, and CAN Talons

1. Upgrade firmware
 1. Connect to RoboRIO with USB
 2. In Firefox/IE visit <http://172.22.11.2/>
 3. Login with “admin” and blank password
 4. Select device to be checked or upgraded on left side
 5. If upgrade is needed, firmware files are in “C:\Users\Public\Public Documents\FRC”
2. Set CAN ID numbers
 1. Each CAN ID is preset to 0 at the factory
 2. If there are multiple devices of the same type (PCM, PDP, or Talon), each needs a unique ID
 3. Choose one, check the Light Device LED box, and press Save
 4. Locate the unit with the rapidly flashing light
 5. Enter the appropriate CAN ID (usually based on what’s in the code) and press Save

17.3.3 Wi-Fi Radio

After attending a competition, don’t forget to reconfigure the radio for home use.

1. Check for Utility updates
 1. Download and run the installer for the latest version of the FRC Radio Configuration Utility from https://wpilib.screenstepslive.com/s/currentCS/m/getting_started/l/144986-programming-your-radio#download_the_software
2. Configure an OpenMesh OM5P-AN or OM5P-AC radio for home use
 1. Attach to the radio with an ethernet cable and disable your computer’s Wi-Fi
 - If using a USB/Thunderbolt Ethernet adapter, you may need to reboot Windows
 2. Run the Utility located in “C:\Program Files (x86)\FRC Radio Configuration Utility”
 3. Fill in the team number
 4. Set the appropriate radio type
 5. Leave the mode at “2.4GHz Access Point”
 6. If not already done, update to this year’s firmware (otherwise, configuration will fail)
 1. Power off the radio by pulling out the power cord
 2. Press the “Load Firmware” button
 3. When prompted, re-insert the power cord
 7. Fill in a robot name specific to this robot (e.g. PracticeBot)

1. This avoids having multiple robots with the same Wi-Fi name
2. The resulting name is composed of the team number and name (e.g. 3504_PracticeBot)
8. Press the `Configure` button and exit the utility when finished
3. Configure a D-Link radio for home use
 1. **NOTE: D-Link radios are no longer supported for competition** as of the 2017 season
 2. Download and install version 16.4 of the FRC Radio Configuration Utility
 3. Run the Utility located in “C:\Program Files (x86)\FRC Radio Configuration Utility”
 4. Fill in the team number
 5. Set the appropriate radio type
 6. Leave the mode at “2.4GHz Access Point”
 7. Press the `Configure` button and exit the utility when finished
 8. Access the administrative interface on the radio at <http://10.35.04.1/>
 9. Login with username “admin” and a blank password
 10. Click `Wireless Settings` on the left side
 11. Add a robot name to the end of the network name to make it unique (e.g. “3504_PracticeBot”)
 12. Click `Save Settings`

17.4 Changing the Team Number

We use alternate team numbers during summer camps and off-season events, so occasionally have a need to change them. The steps are:

1. Update the FRC plug-in in your IDE:
 - In Eclipse, from the Window or Eclipse menu, choose Preferences. The team number is under “WPILib Preferences”.
 - In VSCode, use Control-P (Command-P on the Mac) (maybe with Shift?), then enter the command “WPILib: Set Team Number”
 - In IntelliJ, the New Project wizard for FRC prompts for a team number and stores it with the project. To update, manually edit the `wpilib_preferences.json` found in the “.wpilib” folder at the top of the project.
2. In the SmartDashboard, the team number is in the Preferences.
3. Use the roboRIO Imaging Tool to update the team number. Formatting is not necessary as long as the image is up to date.
4. Use the FRC Radio Configuration Utility to change the team number. Loading firmware is not necessary.

18.1 LIDAR-Lite

LIDAR-Lite is a low-cost distance sensor with a range and accuracy that makes it ideal for use on FRC robots. For around \$150, it can range over 40 meters with an accuracy of 2.5 cm. It was created by [Pulsed Light](#), now part of [Garmin](#). There have been three versions of the product. The first two were sold by Pulsed Light before the Garmin acquisition. Some useful links:

- A few of our favorite sources for LIDAR-Lite v3 (current model as of January 2018)
 - <https://buy.garmin.com/en-US/US/p/557294>
 - <https://www.sparkfun.com/products/14032>
 - <https://www.andymark.com/product-p/am-3829.htm>
- Documentation on the various versions
 - [Garmin LIDAR Lite v3 Operation Manual and Technical Specifications](#)
 - [AndyMark copy of Operation Manual and Technical Specifications](#)
 - [LIDAR Lite v2 Manual on GitHub](#)
 - [LIDAR Lite v1 Manual on GitHub](#)
 - [Garmin software library on GitHub](#)
 - [Pulsed Light software library on GitHub](#)

18.1.1 Connection options

All versions of the LIDAR-Lite unit support both I2C (a two-wire bus interface) and PWM output.

I2C

The I2C interface is very flexible and easy to use, but has been flaky when connected to the FRC RoboRIO. I suspect the root cause is having pull-up resistors on the clock and data lines of *both* the LIDAR-Lite and the roboRIO, but I haven't fully diagnosed it myself. [Chief Delphi](#) has many discussions and potential workarounds. In the end, none of them worked out for us, and we switched to the PWM connection option.

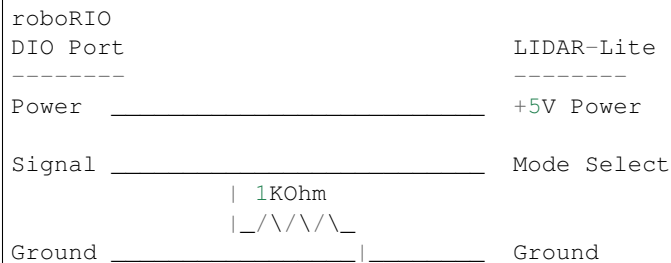
PWM Output

Using an alternate set of pins, the LIDAR-Lite can also produce a pulse-width modulated (PWM) output signal. This just means it raises the logic level of a digital output pin for a time in proportion to the measured distance. Specifically, the output goes high for 10 microseconds per centimeter of measured distance. This is easy to wire to the roboRIO, requiring only a single resistor and one DIO port in the minimal configuration.

Reading the duration of the PWM output is actually very simple on the roboRIO because its FPGA implements the necessary level detectors and timers. Many people on Chief Delphi have suggested converting the signal to analog, but I strongly disagree with that approach since it loses accuracy.

18.1.2 Minimal roboRIO Interface

Here's the simplest way to connect the LIDAR-Lite to the roboRIO using PWM mode. It uses just one DIO port.



I'm not including pin numbers for the LIDAR-Lite cable, only pin names, because it varies between version 1, 2, and 3 of the product.

18.1.3 Source Code

Here's a simple Java class to represent the LIDAR-Lite unit, consisting only of the constructor where the port number is specified and a "get()" method to obtain a reading. This isn't final-quality code in terms of error handling and calibration constants, but should get you started.

```
package org.usfirst.frc.team3504.robot;

import edu.wpi.first.wpilibj.Counter;
import edu.wpi.first.wpilibj.DigitalSource;

public class LidarLitePWM {
    /*
     * Adjust the Calibration Offset to compensate for differences in each unit.
     * We've found this is a reasonably constant value for readings in the 25 cm to 600_
     * ↪cm range.
     * You can also use the offset to zero out the distance between the sensor and edge_
     * ↪of the robot.
     */
}
```

(continues on next page)

(continued from previous page)

```

*/
private static final int CALIBRATION_OFFSET = -18;

private Counter counter;
private int printedWarningCount = 5;

/**
 * Create an object for a LIDAR-Lite attached to some digital input on the roboRIO
 *
 * @param source The DigitalInput or DigitalSource where the LIDAR-Lite is attached
 * → (ex: new DigitalInput(9))
 */
public LidarLitePWM (DigitalSource source) {
    counter = new Counter(source);
    counter.setMaxPeriod(1.0);
    // Configure for measuring rising to falling pulses
    counter.setSemiPeriodMode(true);
    counter.reset();
}

/**
 * Take a measurement and return the distance in cm
 *
 * @return Distance in cm
 */
public double getDistance() {
    double cm;
    /* If we haven't seen the first rising to falling pulse, then we have no
    → measurement.
    * This happens when there is no LIDAR-Lite plugged in, btw.
    */
    if (counter.get() < 1) {
        if (printedWarningCount-- > 0) {
            System.out.println("LidarLitePWM: waiting for distance
    → measurement");
        }
        return 0;
    }
    /* getPeriod returns time in seconds. The hardware resolution is microseconds.
    * The LIDAR-Lite unit sends a high signal for 10 microseconds per cm of
    → distance.
    */
    cm = (counter.getPeriod() * 1000000.0 / 10.0) + CALIBRATION_OFFSET;
    return cm;
}
}

```

18.2 Navigation Sensors

To help track the location of the robot on the field, a device can be used that combines two sensors:

- accelerometer - track the change in velocity over time; integrate twice to find distance traveled
- gyroscope - track the rate of turning; integrate over time to calculate the total angle turned

FRC teams typically use one of these two products as navigation sensors:

18.2.1 Pigeon

- A small board that connects into the CAN bus
- Made by Cross the Road Electronics
- [Pigeon \(Discontinued\) Product Page](#)
- [Pigeon 2 Product Page](#)
- [Software Reference \(for all CTRE products\)](#)

18.2.2 NavX

- A larger board that plugs directly into the large expansion port on the RoboRIO
- Made by Kaili Labs
- [NavX Product Page](#)
- [NavX Programming Reference](#)

18.3 Tuning PID and Related Parameters

These are very brief notes on tuning the various motor control parameters available on the Talon SRX motor controller. Most are related to closed-loop control, but there are a few knobs for open-loop modes (eg: joystick driving during teleop.) The API calls referenced below are methods of a TalonSRX or WPI_TalonSRX object.

18.3.1 Open-Loop Parameters

```
configOpenloopRamp(secondsFromZeroToFullThrottle)
```

We typically want the robot to go from stopped to full speed in something like 0.5 to 2.0 seconds.

18.3.2 Motion Magic Tuning

- Capture the sensor velocity when driving at full speed using the roboRIO CAN interface for a drive Talon
 - $\text{fullspeed} = \frac{\text{sensor positions}}{100 \text{ ms}}$
- Set feed-forward gain (F) so 100% motor output is calculated when requesting the above full speed
 - $F = (100\% * 1023) / \text{fullspeed} = \underline{\hspace{2cm}}$
- The configMotionCruiseVelocity should be less than the full speed, perhaps 75%
 - $\text{cruise} = 75\% * \text{fullspeed} = \underline{\hspace{2cm}}$
- The configMotionAcceleration value sets a limit on the change in velocity per second. If we want to achieve full speed in half a second, the acceleration is twice the full speed value.
 - $\text{secondsFromZeroToFull} = \underline{\hspace{2cm}}$ (maybe 0.5?)
 - $\text{accel} = \text{fullspeed} / \text{secondsFromZeroToFull} = \underline{\hspace{2cm}}$
- Set the proportional gain (P) based on an average error in position seen when running with only the above parameters set. Then decide what percent of throttle you want to use to correct this typical error.
 - $\text{typError} = \underline{\hspace{2cm}}$ sensor positions

- correction = _____ (maybe 10% = 0.1?)
 - $P = (\text{correction} * 1023) / \text{typErr} = \text{_____}$
- To test the calculated value of P, use it to drive to some position. While it's still running, back-drive the motor to see if it promptly responds to additional error. If you're able to move the motor further than you'd like, double P until it is responsive enough.
 - $P = P * 2^n$, where $n=0, 1, 2$, etc. = _____
- Set the derivative gain (D) to correct for any overshoot at the end of travel. If overshooting is observed, start with 10 times the proportional gain.
 - $D = P * 10 = \text{_____}$
- If the mechanism being controlled needs to have very precise positioning, you may need to use the integral gain and integral zone to clean up any residual error. If the above calculations are good enough, skip these values. Because integral values accumulate error over time, they can add up quickly and introduce massive throttle changes at surprising times. Start by observing the typical residual error in positioning. Start with an integral zone (range of error values over which those errors are accumulated into the I term) of two times the typical error.
 - $\text{typResidual} = \text{_____}$ sensor positions
 - $\text{iZone} = 2 * \text{typResidual} = \text{_____}$
 - $I = 1/100 * P = \text{_____}$
- If the residual error is not resolved, you can either increase the integral zone or double the integral gain
 - $\text{iZone} = 3 * \text{typResidual} = \text{_____}$
 - $I = I * 2^n$, where $n=0, 1, 2$, etc. = _____

These code labs walk you through our general workflow process and working with the simulator.

All of the examples work out of the GitHub repository, <https://github.com/GirlsOfSteelRobotics/Git-Workflow-Codelab>

You will need to clone it locally to start working

19.1 Gitflow Lab

This lab will walk you through our workflow, from creating branches through merging a pull request.

Here is the project we will be using <https://github.com/GirlsOfSteelRobotics/Git-Workflow-Codelab>

19.1.1 Part One

1. Checkout a new branch

IMPORTANT: For this code lab, you will base your branch off of something other than master. This is to force a merge conflict later in the lab. 99.9% of the time you are developing real code from the robot, you would start off of master

Base the branch off of origin/codelab_start, and name it something like <your name>_codelab_part1

2. Create a new Subsystem

Create it with the name “<your name>Codelab<year>Part1”, ex. PJCodeLab2020Part1

3. Put a print line in the constructor

Something along the line of `System.out.println("<name> says hello world in <year> part 1");`

4. Create your command

In `RobotContainer`, declare your subsystem.

5. Run SnobotSim

Run it from the run configurations area, and make sure your string gets printed out

6. Commit, Push, Create PR

You will notice that the you cannot merge your branch, because there is a conflict

7. Fix conflict, re-push

After the push, add PJ or Joe as a reviewer, and ping them in Slack to review and approve the PR

19.1.2 Part Two

Part two is meant to make sure you run the cleanup process correctly, and can create your next feature branch.

Re-run steps 1-6, but replace any references to “part1” with “part2”

19.2 Simulator Lab

This lab will get you acquainted with using the simulator, and executing some basic robot controls. Many parts of the robot code have been “stubbed out”, meaning that much of the infrastructure work you would normally have to write already exists, so that you can just focus on the fun part of making the robot do things.

All of the places that need to be updated are marked with TODO’s, which can be seen in the TODO tab IntelliJ. This lab will walk through the easiest to most complex tasks in order, but feel free to jump around if you are comfortable.

This project also contains unit tests. Unit tests run automatically when we build, and test the code and make sure that certain expectations are met. In order to “pass” this codelab, all of the unit tests should pass

19.2.1 Robot Overview

We will be implementing code for a very simple robot.

The robot has the following mechanisms (think: Subsystems): - A Chassis, with 4 speed controllers (2 each side), an encoder for each side, and a gyroscope - A Elevator, with one speed controller, an encoder, and limit switches on the top and bottom - A Punch, which uses a single action solenoid - A Spinning Wheel, with one speed controller and an encoder

Our goal is to give the robot these abilities (think: Commands) - Drive the chassis with the drivers Xbox controller, using Halo drive (aka split arcade, aka left thumb throttle, right thumb rotation) - Drive the elevator with a joystick

on the operator XBox controller - Have buttons which allow the elevator to go to 3 preset heights - Have a button to move the punch piston - Have a button to spin the wheel at a preset RPM - Autonomously drive the robot straight for N seconds - Autonomously drive the robot straight for N feet - Autonomously turn the robot to N degrees

19.2.2 Implement the Punch Subsystem

The punch subsystem uses a pneumatic solenoid to control our piston. Pneumatics can only have two states, extended vs retracted (in vs out). Because of this, we interact with them using the `boolean` values `true` or `false`

If we look at our `PunchSubsystem` class, we can see the member variable `m_punchSolenoid`, and the functions `extend` and `retract`. Inside of these functions, we will want to **SET** the solenoid to to a certain state. Our coding standard states that the extended state should be `true`, and the retracted state should be `false`. We should also fill out the `isExtended` function, by asking for and **GET**ting the current state

Once that is complete, you can try running the `PunchSubsystemTest`, and see if you have implemented things correctly. Once the tests pass, make a commit.

19.2.3 Implement the Elevator Manual Controls

The elevator subsystem uses a speed controller to make the physical motor spin. Motors can go multiple different speeds, and the library provides a nice API where we can tell the motors to go full speed backwards by **SET**ting it to 1.0, and making it go full speed backwards by **SET**ting it to -1.0. We can also do any speed in between, like going half speed forwards by **SET**ting it to 0.5, or making it go backwards at 10% speed by **SET**ting it to -0.1

For manual control, we make a function, called `setSpeed`, and we will give that value directly to the speed controller

The elevator also has an encoder, and we can ask it for its current position. We should fill out the `getHeight` function, and return whatever the value is when we **GET** the **POSITION**

Once that is complete, you can try running the `LiftingSubsystemTest.testManuallyMoveUp` and `LiftingSubsystemTest.testManuallyMoveDown` tests. Once the tests pass, make a commit.

19.2.4 Implement the Chassis Manual Controls

Similar to the elevator, the chassis has a motor controller and an encoder for each side. However, the FIRST library provides a nice helper class, `DifferentialDrive` which does some fancy math to help us implement our arcade drive functionality, so we won't be **SET**ting the motor controllers directly, but rather talking to that classes `arcadeDrive` function. This function takes in a speed (aka throttle, aka how fast we want to drive straight), and a rotation value (how much we want to curve/spin).

We will also want to fill out the `getLeftDistance` and `getRightDistance` functions, like we did with the elevator. You will need to do a little bit of simple math to fill out the `getAverageDistance` function, but it should incorporate the distance of both the left and right sides.

Once that is complete, you can try running the `ChassisSubsystemTest`. Once the tests pass, make a commit

Note: You are off the hook for the gyroscope, because the API is too gross to explain for this lab.

19.2.5 Implementing Joystick Interactions

Up to this point, we added the ability for the chassis and elevator to moe with manual input, and now we want to ask our xbox controllers for what that manual input should be. For example, the more we press the driving joystick forward, the faster we want to make the robot driver. Luckily, the joysticks are also normalized to a [-1.0, 1.0] range, so the mapping is easy.

To do this, we need to implement the functions in `OI`

For `getDriverThrottle` we want to **GET** the **Y** axis for the **LEFT HAND**. For spin, we want to **GET** the **X** axis on the **RIGHT HAND**. For the elevator, we want to **GET** the **Y** on the **RIGHT HAND**

IMPORTANT NOTE For historical reason, due to how people like to fly airplanes, pushing forwards on the Y axis will result in a negative value, and pulling back yields a positive value, so you will probably want to negate those values.

Now that we can read the joysticks, we can implement our manual tele-op commands. To do this, we can go into our `LiftWithJoystick` and `DriveChassisWithJoystick` commands, and call the functions we implemented in the previous steps.

For the elevator, we want to **SET** the **SPEED**, and give it the values from **GETing** the **ELEVATOR JOYSTICK** from the `OI` member variable

For the chassis, we want to **SET** the **SPEED** and **STEERING** values, and give it the values from **GETing** the **DRIVER THROTTLE** and **DRIVER SPIN**.

Once those are hooked up, you can run the `DriveChassisWithJoystickCommandTest` and `LiftWithJoystickCommandTest` tests. Once those pass, make a commit.

19.2.6 Implement Driving with Timers

Most of the time in autonomous, we want finer grained controls than just “drive forwards for a couple seconds and hope we don’t hit a wall”, but it is always a good command to have in our back pocket in case all of our sensors break.

To do this, we can make a command that will drive straight with some speed (either forwards or backwards), and give it an amount of time to run. `WPILib` provides a timer functionality that we can **START** as soon as our command runs, and we can **GET** how many seconds have elapsed each loop, and say our command **IS FINISHED** after the time is bigger than our argument. Each time our command **EXECUTES** we can **SET** our **THROTTLE** to whatever our speed argument is.

Note, it is important that when we **END** our command, we stop driving, otherwise the chassis will keep on trucking after our timer has expired

Once those functions have been filled out, you can run the `AutoDriveStraightTimedCommandTest`. Once the tests pass, make a commit

19.2.7 Implement Driving a Distance

More often, we will want to tell the robot to drive some number of feet in autonomous mode.

So to **EXECUTE** this command, we will want to check where we currently are (based on our **AVERAGE DISTANCE**, and figure out if we need to drive forwards (positive throttle) or backwards (negative throttle). We can say we are **FINISHED** when we are close enough to our goal distance. Like the previous command, it is important that when we **END** our command, we stop driving.

IMPORTANT NOTE We will never hit our goal right on the nose. Doing a `current == goal` check will (pretty much) always fail, because if we are even off by the width of a single atom, we aren’t there. Usually it is good to set up some kind of **ALLOWABLE ERROR**, and check if we are inside of that window. We want to make sure we are within `[-deadband < error < deadband]`. Using something like `Math.abs` makes that check pretty easy.

Once that is complete, you can run the `AutoDriveStraightDistanceTest` tests. Once those pass, make a commit.

19.2.8 Implement Moving the Elevator to a Height

Much like the “drive straight a distance”, we want our elevator to move up or down until we have reached our goal height. When we **EXECUTE** our command, we will figure out if we need to move up or down, and we will **SET** the elevator motors **SPEED** accordingly. Also like the chassis, we want to make sure we stop the motor when we **END** the command.

Once that is complete, you can run the `LiftToPositionCommandTest`. Once those pass, make a commit

19.2.9 Wire Up Commands to Buttons

Now that we have all of our commands implemented and tested, we can hook them up to buttons on the driver and operators xbox controllers.

This takes place in the `OI` class. We want the operator joystick to do the following things: - When B is pressed, have the lift go to a position of 0 inches - When Y is pressed, have the lift go to a position of 40 inches - When X is pressed, have the lift go to a position of 60 inches - When A is pressed, extend the punch. When it is released, retract it - When RB (`kBumperLeft`) is pressed, make the spinning wheel run.

19.3 Custom Shuffleboard Widget Lab

We frequently use Shuffleboard to debug and verify our code, especially in the simulation environment. The text boxes and boolean indicators can help us out a lot, but sometimes it is better to have a more complicated, customized view to look at.

Shuffleboard gives us the ability to create our own visualizations, written in Java, which communicate over the same Network Tables interface used for the rest of shuffleboard. This allows us to do things like plot our robot position on the field, or even make a simplistic view of our robot, where we can watch our mechanisms move just like they do in real life.

This provides us with multiple benefits. When we are developing complicated things like autonomous modes in the simulation environment, it is way easier to look at a “video” of the robot move its elevator up to the correct height, then eject balls than it is to look at a text box with the elevator height rising, and then a light very quickly turn from red to green when the solenoid switches state. It can also help debug wiring problems the first time we bring the software up. For example, “When I hit X the intake motors are supposed to move, and when I hit Y the shooter is supposed to start, but it is backwards. The Widget says it is doing the correct thing, it must be wired incorrectly”

We will be making a custom widget for the robot we created in the previous code lab. Like that codelab, lots of stubs have been created which we will need to fill in.

19.3.1 Introduction into JavaFX

JavaFX is a framework built on top of java, used to create graphics. You can make very complicated GUI (graphical user interfaces) complete with buttons, text fields, dropdown boxes, custom graphics, etc. For our shuffleboard purposes however, we will stick to drawing shapes, typically just `Rectangle` and `Circle` objects for simplicity. When the state of the robot changes, we can change the colors of these shapes, move them around in the image, change their sizes, and even rotate them around an angle.

JavaFX uses a concept called `mvc` (Model - View - Controller), where the View (what shapes we have) are defined in their own file (typically a `*.fxml` file in the resources package), the Model (the data structure definitions) in a separate java file (typically called `*Data.java`), and the Controller (typically in a `*Controller.java`), which takes in changes in the model and updates the view accordingly.

We have written a script that automatically generates the model from a config file, since shuffleboard requires a LOT of boiler plate code, and we will just need to focus on what shapes we need to draw (view) and how draw them to reflect the state of the robot (controller).

19.3.2 Making a View

The project already contains our view (`src/resources/shuffleboard_codelab.sd_widgets.ss.codelab_super_structure.fxml`), but we need to add our shapes to it. We can do it by adding all the rectangles and circles we need inside of the group tags, like so

```
<Group fx:id="m_group">
  <children>
    <Circle fx:id="m_spinningWheel" fill="plum"/>
    <Rectangle fx:id="m_punch" fill="peachPuff"/>
  </children>
</Group>
```

Note the `fx:id="..."`. The ID is used to auto-magically let the controller and the view talk to each other. So the id should be whatever you want the member variable name to be. Remember, our coding standard says member variables should start with `m_`

You also can play around with the `fill`. JavaFX provides a LOT of predefined colors, you can use intellisense to see what options are available. It usually helps debugging the layout if you make each item a different color, so you know which thing you are moving around. Unfortunately, we will usually override these colors with more standard values when we actually connect it to the model.

For this lab, we will want: - Rectangle for the chassis base - Rectangle for the elevator - A small Rectangle for each of the two limit switches for the elevator - Rectangle for the punch solenoid - Circle for the spinning wheel

19.3.3 Setting up the Controller

The first step in writing the controller is defining all the member variables for the shapes you defined in FXML, like so

```
@FXML
public Circle m_spinningWheel;

@FXML
public Rectangle m_punch;
```

Note the special `@FXML` annotation; this is what allows the black magic connection of the things in your FXML file and your java file. You have to be careful and make sure your names match the ids in the file, there will be no compiler errors if you make a mistake, but you will likely get a `NullPointerException` when you try to run it, which can be confusing and hard to track down.

Next, you will want to move the shapes around and make them the appropriate size. There is a special `initialize` function, which auto-magically gets called after the FXML file has been loaded, so you should do your work in there.

For the most part, we will only need to set four attributes for `Rectangle`, the X and Y (top left right corner), and its height and width. For a `Circle`, we need to set its Center X and Y location, and its Radius. For simplicity, we can use inches for these values, and scale the shapes to make it fit inside of our window (the scaling code is provided)

Note: Computer graphics typically put the origin point at the top left corner of your screen. Moving down the screen increases your Y value, moving to the right increases your X value. For example, if you had a screen that was 200x100 pixels wide, the coordinate system would look like below.

This, combined with the fact that we specify the top left coordinates for rectangles can make it tricky to think about how to draw an elevator taller, as you will need to change both the height and the Y location

```
m_spinningWheel.setRadius(2);
m_spinningWheel.setCenterX(10);
m_spinningWheel.setCenterY(20);

m_punch.setWidth(5);
m_punch.setHeight(7);
m_punch.setX(30);
m_punch.setY(25);
```

You should now try to place and size all of the objects until it looks like the sample image (except you can substitute the colors for whatever your heart desires)

19.3.4 Making the Widget Dynamic

Now that we have all of our components in the correct place, we can start making them move to reflect the state of the robot.

You will notice that there are some stubs that get called when the robot state changes, like `updatePunch`. We can ask the function argument **IS* the **PUNCH EXTENDED?** If it is, we can change the punches rectangle, so it looks like it is bigger to reflect the extension.

Similarly, we can change the color of these shapes when they are moving. For example, in the `updateSpinningWheel` function, we can **GET** the **MOTOR SPEED**, and use the following utility to color it based on how fast it is going.

```
m_spinningWheel.setFill(Utils.getMotorColor(spinningWheelData.getMotorSpeed()));
```

We want the widget to do the following things: - Change the color of the elevator box based on the motor speed - Change the size of the elevator rectangle based on the height - Make the limit switches green if they are at their limit, or black if they are not - Make the punch bigger if it is extended, or put it back to its default height if it is retracted - Change the color of the spinning wheel based on the motor speed

19.4 Calculator Lab

In this lab, you will implement and test a command-line calculator app.

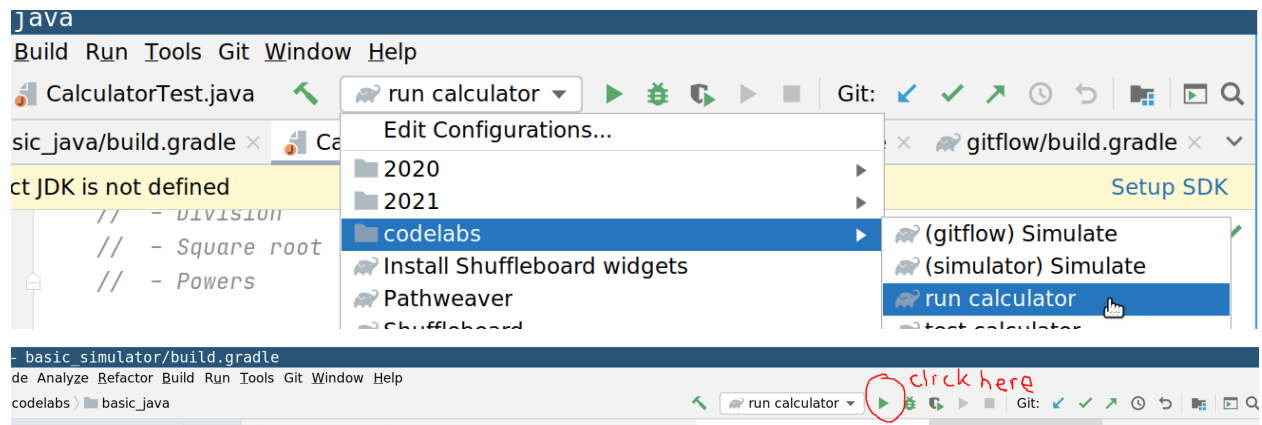
Unlike most of the code you will work with in GoS, you should be able to understand the majority of this code.

The project code is located in the monorepo: <https://github.com/GirlsOfSteelRobotics/GirlsOfSteelMonoRepo> in the `codelabs/basic_java` folder.

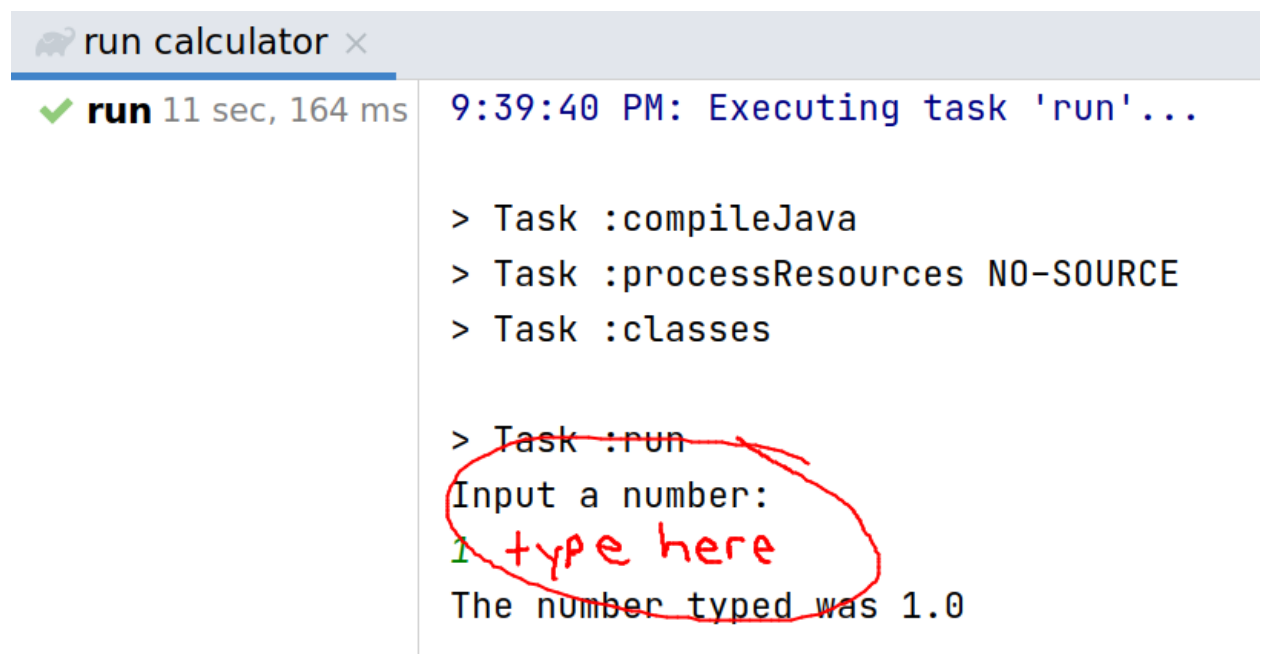
19.4.1 Part One

1. Run the existing calculator

One function (add) has been implemented for you. Select “run calculator” (shown in the menu screenshot below) and click the green play button to run the calculator (also shown in a screenshot below)



At the bottom of the screen, follow the prompts to add 2 numbers and print the result.



What happens?

2. Find the file you just ran

Code files are generally in `src`, though often nested in many layers of folders. You can find this one in `codelabs/basic_java/src/main/java/com/gos/codelabs/basic_java`

Read the file. Look for the main method, which is anything between the curly brackets in `public static void main(String[] args)`. It's okay if you don't know what all of those words mean yet. It basically means the class has a function available outside of it that does not output anything. The image below breaks the components down.


```

public class Calculator {
    public double add(double x, double y) {
        return x+y;
    }
}

```

Handwritten annotations:

- access**: points to `public`
- class name**: underlines `Calculator`
- output**: points to `double` (return type)
- input+types**: underlines `double` (parameter type)
- input+names**: underlines `x, double y` (parameter names)
- output**: points to `x+y` (return value)

Although there are nicer ways of doing this, the simplest way to build a string is to use a + operator between the strings. Example: "The number" + "typed was" Find the line in the code that caused "The number typed was" to be printed to the screen.

3. Edit the line printed.

Change the line in the code to anything else. For example, to print *Alex typed in the number*

4. Create a calculator object

Java uses objects, which are structures that contain data and perform functions. More details: <https://docs.oracle.com/javase/tutorial/java/concepts/index.html>

In Java, the syntax (pattern of typing code that can be read by what builds it) of creating an object is `Object nameOfMyObject = new Object()` The parentheses go around the inputs to the object, which in this example is empty. We have one object being made in the file already. `Scanner input = new Scanner(System.in)`; A scanner is an object that reads in text from a prompt.

The object you will create is a `Calculator`. Follow the pattern above to create a `Calculator` called *myCalculator*.

5. Use the add function of the calculator

Objects can call methods, which are functions that operate on inputs. We call the inputs arguments. To call a function, type a dot after the object name, and then put the inputs in the parentheses. `myCalculator.add(3, 4)`

Try adding 5+6 using `myCalculator`. Print out the result.

6. Find Calculator.java

Find the `Calculator.java` file. Look at the structure. What are the functions it already has? Where is the `add` function?

Let's understand what the functions mean. `public` means the function can be accessed anywhere. It is the least restrictive type of access. The opposite is `private` `double` means a computer's approximation of a real number, with decimals. Ex) 4.563201 is best represented using a double.

7. Add functions to Calculator.java

Following the pattern of `add` add more useful functions to `Calculator.java`. Java operators are similar to what you use on your graphing calculator. Ex:

1. subtract: `x-y`
2. multiply: `x*y`
3. divide: `x/y`

Bonus: Call your functions in RunCalculator.java

This can potentially be a challenging exercise as it involves putting together several concepts. If you find yourself stuck, either ask for help from your fellow students, mentors, or move onto the Part 2 and come back later. Useful concepts:

1. `charAt(0)` outputs the 0th (first) character of a string of characters. You can use it to read a character from the input
2. conditions: a statement that is either true or false. To check equality, Java uses two equal signs (`==`) Ex)
3. `(3 == 4)` a false statement
4. `(4 == 4)` a true statement
5. `('a' == 'b')` a false statement
6. if statement: run the code inside the brackets if the condition in the parentheses is true

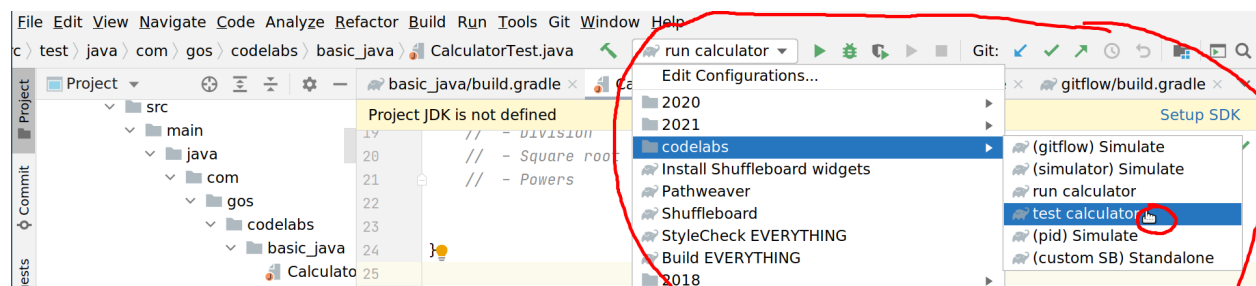
There is some sample code in the `TODO` that might be helpful.

19.4.2 Part Two

Tests are a useful way of making sure the code is working properly. We use `assert` statement that checks whether the condition in the parentheses is true. If it is not, it throws an error. Why might this be useful for checking if our code is working?

1. Find CalculatorTest.java.

You can find it in your IDE. If you prefer, you can also use Windows Explorer or Mac Finder to find this file. Run "test calculator" from the menu above



2. Make a test fail

Look at the result in the bottom window. Try to make the test fail. Then change it back.

3. Test your subtraction method

Follow the pattern in `testAddition` in `testSubtraction` to test your `subtract` method. Make sure it passes.

4. Test your other methods

Write tests for the other functions you wrote.

CHAPTER 20

Monorepo intro

We now use a monorepo, aka we keep all of our code in a single repository instead of creating a new one every year. This allows us to easily reference our old code, and quickly up-convert our previous years robots to the latest version of WPILib without having to copy robot projects between repositories.

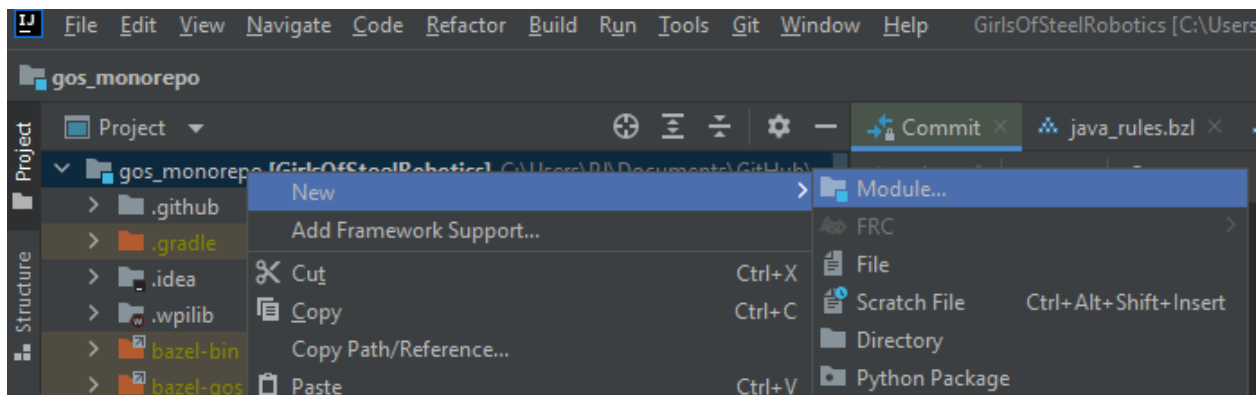
However, that does make some of the “one time” tasks like creating a project a little bit more involved, and we are forced to make sure every project compiles with the latest versions of the external libraries.

20.1 Creating a New Robot Project

Follow these steps to create a new robot project. This assumes that you have the FRC IntelliJ plugin installed

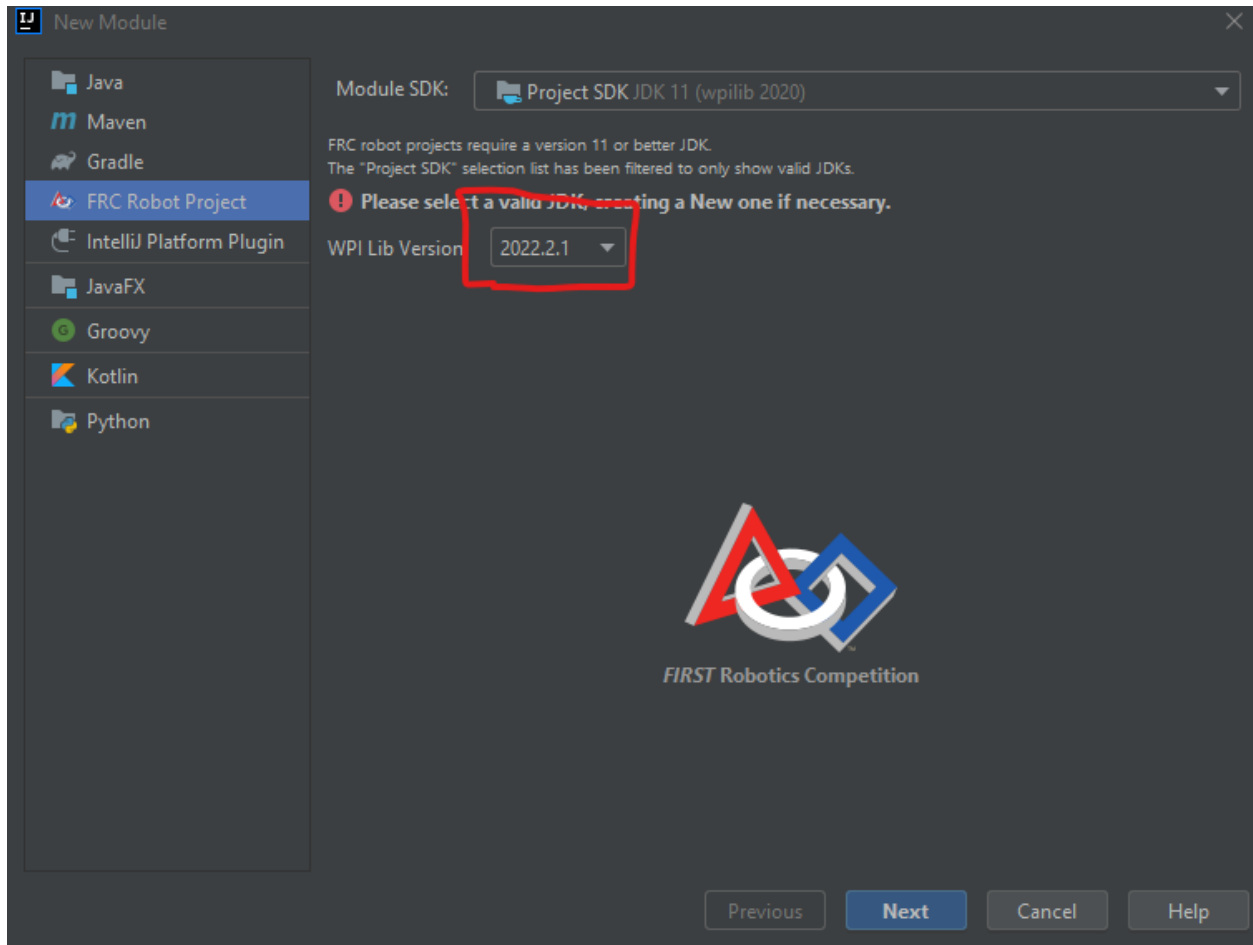
20.1.1 1. Create a new module

From the file menu, do File -> New -> Module

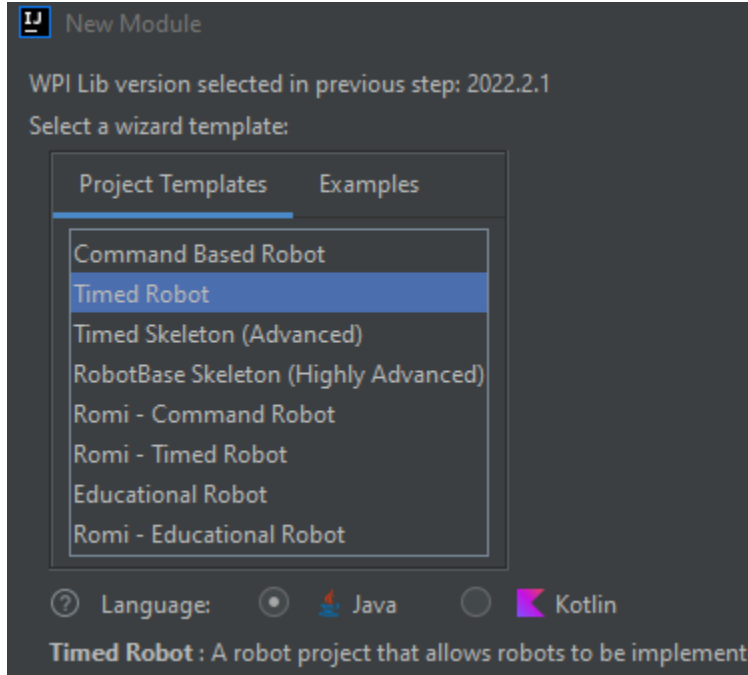


20.1.2 2. Select the FRC Robot Project

Note: , Ensure the most recent version of WPI Lib is selected

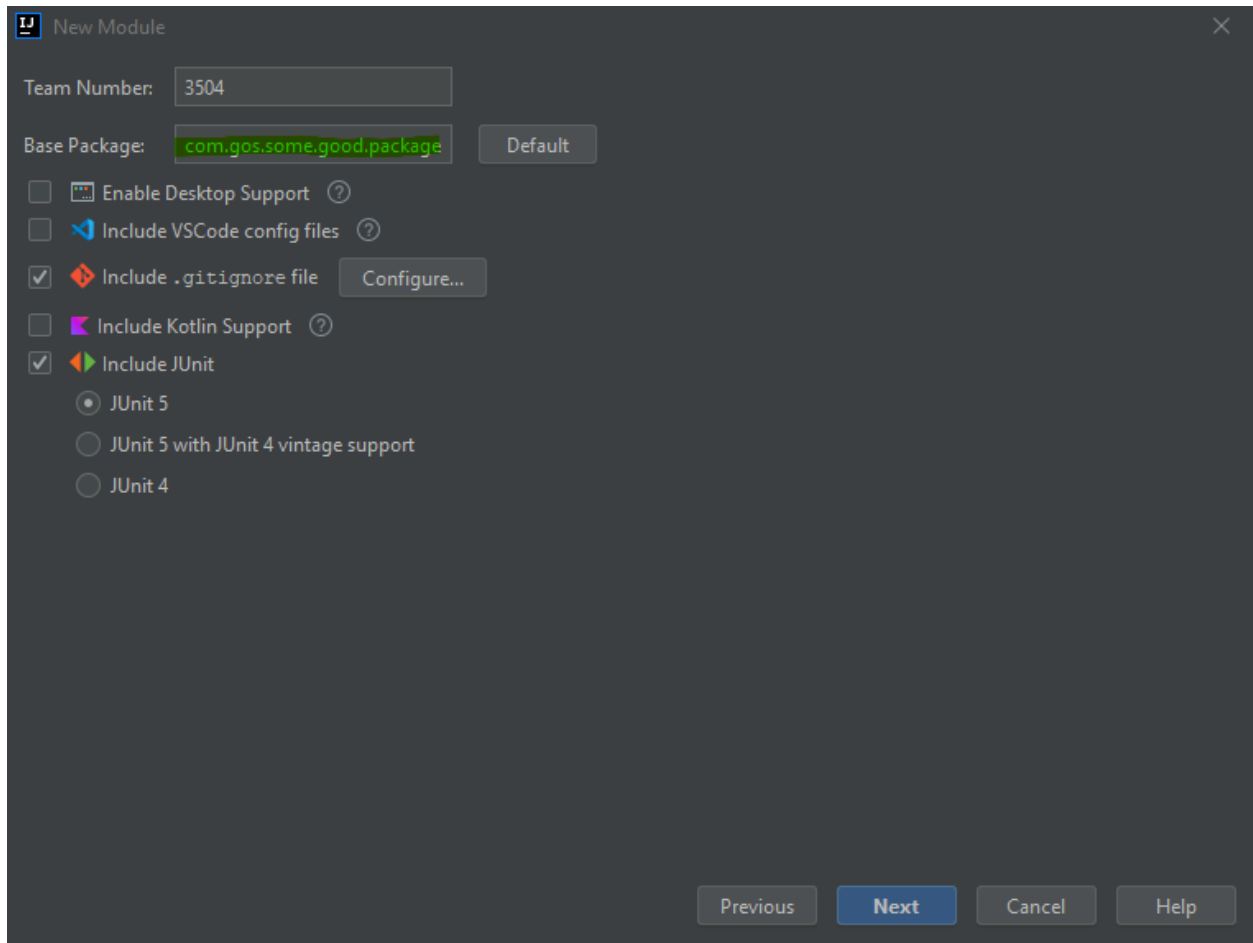


20.1.3 3. Select a template to start with. We use the Timed Robot template



20.1.4 4. Set the team number and packge name

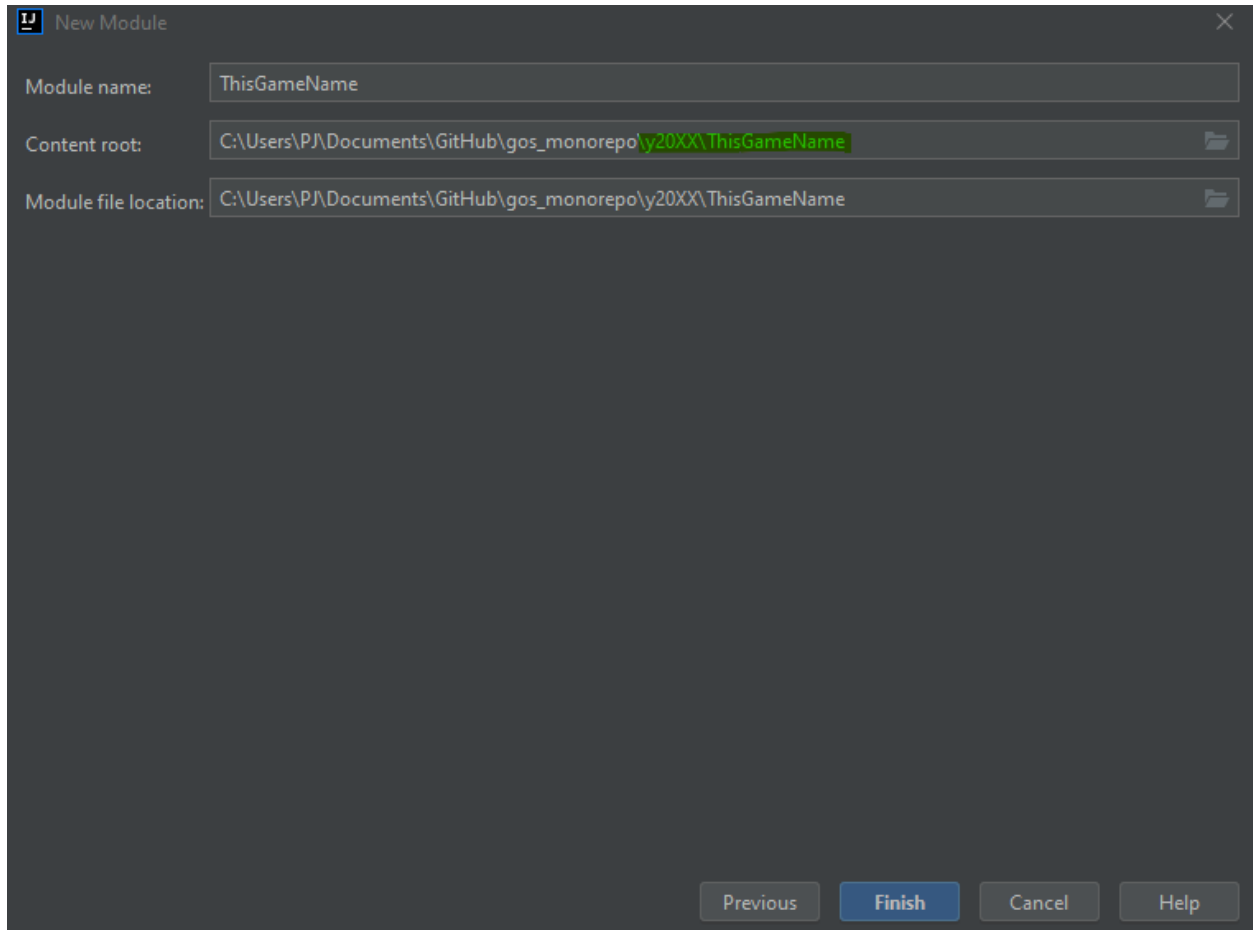
We want all of the projects in the monorepo to have a unique package name so we can differentiate them. All of our packages should start with `com.gos`. For a robot project, the next chunk should be the game name, in snake_case. i.e. `infinite_recharge`, `power_up`, etc



20.1.5 5. Select where the project should live

We group our projects by year, so the first sub-directory in the repo should match y20XX. The last chunk should be what we want this project to be called. It is common for us to name the robot after the game in UpperCamelCse, i.e. “InfiniteRecharge”, “PowerUp”, etc.

NOTE: It is easiest if you manually type these things into the “ContentRoot” section. Typing in there will automagically fill in the Module Name and Module file location.



We now we have our barebones robot generated.

20.1.6 6. Add libraries.

We will very likely need third party libraries to write our code, so we can start by adding all of them, and trim them later if they aren't actually needed. Doing this "Ask for forgiveness" approach should allow us to get up and running quicker, and avoid situations where three different groups all have to add a RevLib dependency.

a. Copy vendor deps

VendorDeps are the way that gradlerio pulls in third party FIRST libraries, like CTRE and REV. They are json files that live in a `vendordeps` folder in the robot project. The easiest way to grab them all is to literally copy and paste them from an existing project, like `y2020/2020InfiniteRecharge/vendordeps`

b. Add internal library dependencies

In addition to the external vendordep libraries, we also have a suite of libraries built up over time to make our lives easier. Similar to the vendor deps, it is easiest to add them all.

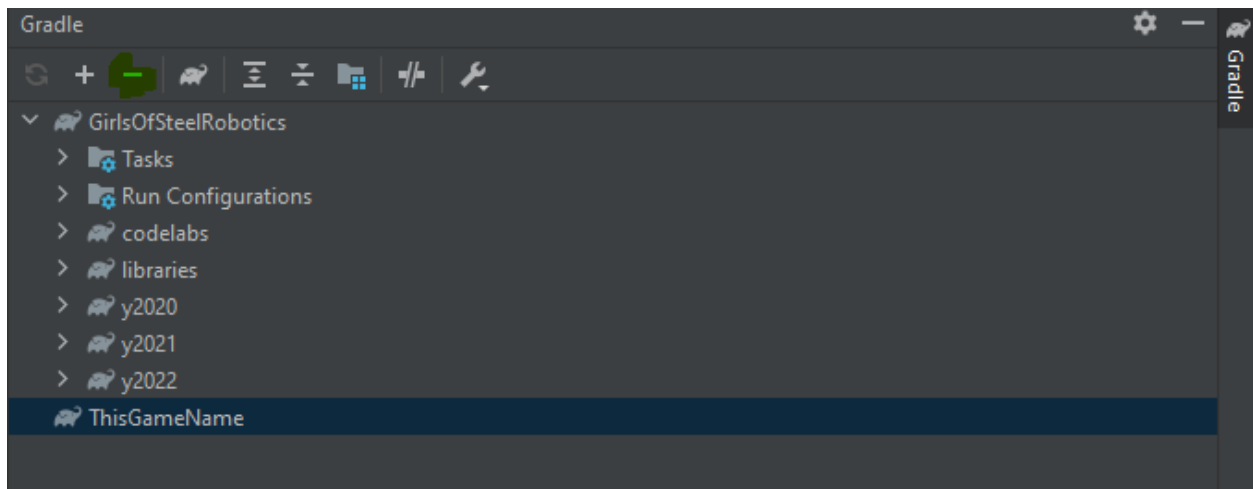
Add this to the dependency block in the robots `build.gradle` file

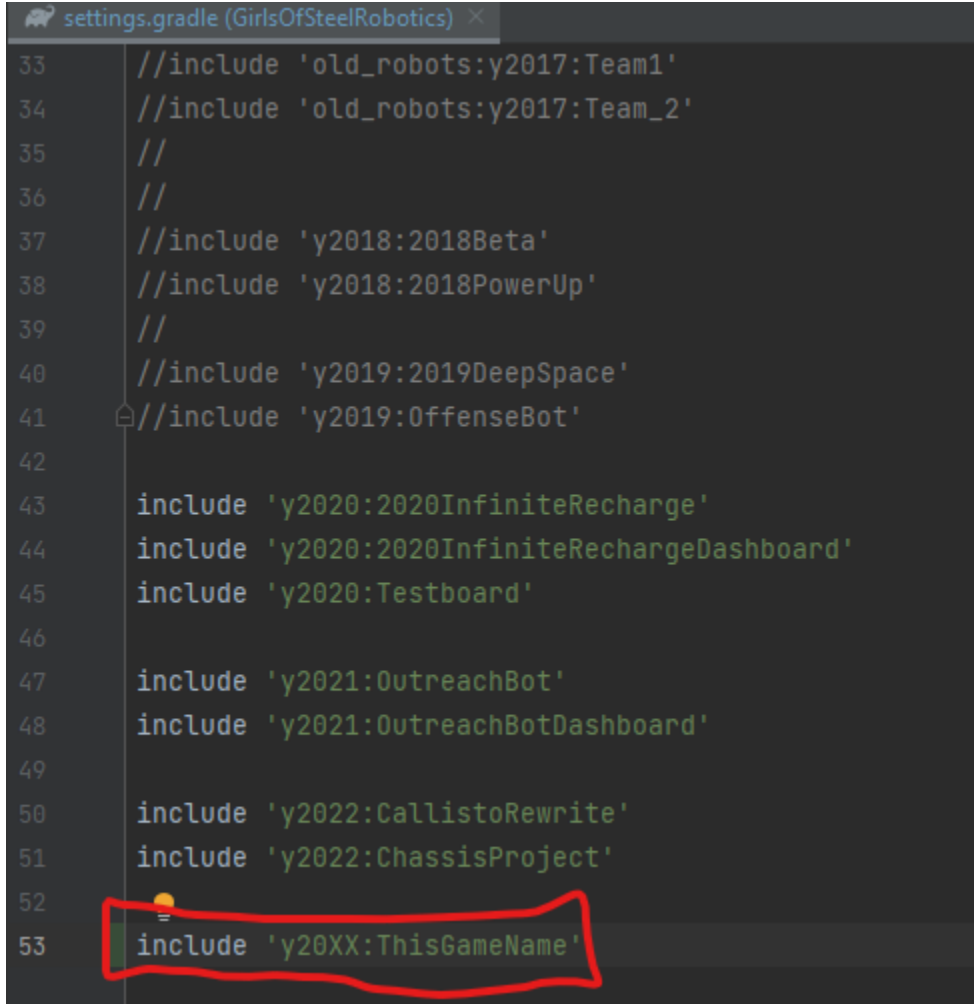
```
implementation project(":libraries:GirlsOfSteelLib")
implementation project(":libraries:GirlsOfSteelLibCtre")
implementation project(":libraries:GirlsOfSteelLibNavx")
implementation project(":libraries:GirlsOfSteelLibRev")
```

20.1.7 7. Cleanup

It is very likely that IntelliJ / FRC plugin did not add the project in the way that we wanted, so we might need to do these manual steps

a. Remove the project from IntelliJ's gradle tab



b. Add the new robot project to the overall project

```
33 //include 'old_robots:y2017:Team1'
34 //include 'old_robots:y2017:Team_2'
35 //
36 //
37 //include 'y2018:2018Beta'
38 //include 'y2018:2018PowerUp'
39 //
40 //include 'y2019:2019DeepSpace'
41 //include 'y2019:OffenseBot'
42
43 include 'y2020:2020InfiniteRecharge'
44 include 'y2020:2020InfiniteRechargeDashboard'
45 include 'y2020:Testboard'
46
47 include 'y2021:OutreachBot'
48 include 'y2021:OutreachBotDashboard'
49
50 include 'y2022:CallistoRewrite'
51 include 'y2022:ChassisProject'
52 include 'y20XX:ThisGameName'
53
```

20.1.8 8. Build the project

It is very likely that the default templates generated code that does not meet our coding standards. Running a build will notify us to the breakages that must be fixed before we can make a PR.

20.1.9 9. Make a PR and land the new robot**20.2 Create New Shuffleboard Plugin**

CHAPTER 21

Indices and tables

- `genindex`
- `modindex`
- `search`