

# Programming WPILib on a Romi

---

## Lesson 1: Clone-ing around

This repo (short for repository) is stored in the [FRCTeam3206 GitHub account](#). In order to work with it on your own computer, you will need to [clone](#) it. Cloning creates a local copy of the repository on your computer that is linked to the remote repository stored on GitHub. You will be able to edit this local repository in WPILib VSCode.

There are several ways to clone a repo from GitHub. First, open the [RomiTimed23](#) repo on GitHub. Above the code window, there is a green button that says `<>Code` on it. Click on this button to see a menu with several options to clone the repo. The simplest option, if you have [GitHub Desktop](#) installed is to select [Open in GitHub Desktop](#). This will create a clone linked to the remote repository. The clone will be located in the "GitHub" directory under "Documents".

Now that you have cloned this repo, open WPILib VSCode and then open the repository.

- On the menu bar, click on [File](#) then [Open Folder ...](#), you could also use the keyboard shortcut [Ctrl-K Ctrl-O](#).
- Navigate to [Documents\GitHub](#)
- Click on the folder with this repository
- Click on the [Select Folder](#) button (this still works if you navigate into the folder)

You should now see the contents of this repo on the left side of the VSCode window. Most of the folders and files that you see are there to support the build and deploy process and you won't need to make any changes to them. The robot code will all be stored in:

```
src\main\java\frc\robot\
```

Open this directory and you will see several `.java` files.

- [Main.java](#) - This is the "entry point" for the Java program. This file already contains the correct code and you won't need to edit it
- [Robot.java](#) - The top-level file that defines the behavior of the robot. This is where you'll place the code that controls your robot.
- [RomiDrivetrain.java](#) - The code that defines the drivetrain and how it operates.

Go ahead and open [Robot.java](#) and take a look at the code.

---

## Lesson 2: A bit of `git`

*NOTE: If you are using the shared CREATE\_XX computer, you are signed in to GitHub with the "RoyalT-Wrecks" account. Please don't sign in with your own account on these computers. If you are working from your own computer, please use your own GitHub account. If you have any trouble using git or GitHub, post a note on the team Discord.*

`git` is a popular version control system (VCS) for managing source code. A VCS provides a solution to the problem of managing code changes when multiple people are contributing to the same code base. It can also make it easier to keep track of changes and correct ones that cause bugs.

In a VCS, all code changes are tracked and can be reverted if they cause problems. `git` does this using the concept of **branches** and **commits**.

All repositories have at least one branch (called 'main' in this repo). When you want to modify the code, you create a new branch and edit the code in that branch. Once it has been tested, you then merge the new branch back into the main branch. `git` supports having multiple parallel branches, so it is possible to work on several different parts of the code base at the same time. A good branch is one that represents a single type of change to the code base, for example a branch to fix a specific bug or one to add a new feature.

Code changes are added to a branch with a commit. This allows you to specify exactly which changed files are added to the branch and document what has been changed in a given commit.

### Using `git`

There are several ways to access `git` commands. The command line is the most powerful option, but it can be tricky to remember the correct commands. VSCode has integrated support for `git` and is convenient since you can do things directly in the IDE (integrated development environment). Finally, there is GitHub desktop, which you probably used to clone the interface. The dedicated GUI is easy to use and allows you to do everything you'll need to contribute to the team code repository.

### Command Line Interface

If you don't already have `git`, you can install it from [git-scm.com](https://git-scm.com). The WPILib Documentation includes a [Git Version Control Introduction](#) that provides a good overview of using the `git` command line from the `git bash` shell.

### Git and GitHub in VSCode

VS Code has built-in support for managing git repositories, which allows you to conduct the common actions directly from the IDE. We also have the GitHub extension installed so that you can create pull requests in the IDE.

### GitHub Desktop

[GitHub Desktop](#) is a graphical user interface to the most common features of `git`. You can use this to clone repos, make commits, and push and pull code.

### Basic `git` operations

For the next several sections, we'll use the `git` command line interface for the examples. You can do the same things inside VSCode or with the GitHub Desktop by finding the right menus.

## Making a branch

A branch is an isolated copy of the code. You can make any changes you want within a branch without affecting the code in any other branches. You can have as many branches as you want, but having a lot can get hard to manage. Once you have made and tested your changes, you can `merge` the code from your branch back to the `main` branch to make it part of the robot code.

Always work on a branch since it isolates your changes and allows multiple contributors to work on the code simultaneously without overwriting one another's changes.

If you don't already have it open, load your local repo in WPILib VSCode. Open a terminal from the `Terminal` menu if one isn't already open. If you have installed `git` then you can type the following to create a new branch:

```
git checkout -b "your-branch-name"
```

This command both creates and checks-out (makes active) the new branch. Branch names should be short and descriptive. The name can't contain spaces, so it is common to use dashes between words for readability.

The branch you create will be based on whatever code is in your current working directory. If you make edits to the code, then create a new branch, your edits will be available in your new branch.

You can see which branch you are on by typing:

```
git branch
```

This will display a list of all local branches with an asterisk (\*) by the current branch.

If you want to change to a different branch, you can use:

```
git checkout "branch-to-checkout"
```

Make sure that you've committed any changes you want to keep before changing branches.

## committing code to a branch

Now that you are on your own branch go ahead and edit `Robot.java`. For example, add a comment with your name at the top of the file and save the file. To store this change in the repo, you need to `commit` it to the repo. There are two steps to making a commit, staging the files and then making the commit.

First you indicate which files you want to update by `adding` them to the stage (list of files that will be committed):

```
git add Robot.java
```

To add all files that have been changed, type:

```
git add .
```

Second, you **commit** them to the repo:

```
git commit -m "Short description of the change"
```

The **-m** tells **git commit** to add the text following it as the description of the commit. Good comments are short and help people understand what was changed. Instead of saying "fixed stuff", explain what you fixed, such as "corrected motor ids". You don't have to describe the exact code, since people can see how the code changed when they look at the commit, instead, try to describe why you made the change.

### Push code to the remote (GitHub)

Finally, we need to add this to the **GitHub** repository. The first time you **push** a branch to the upstream repository, you need to tell it that you want to add the branch to that repo.

```
git push -u origin "your-branch-name"
```

After that, you'll be able to update the branch any time you make a new commit by just typing:

```
git push
```

Whenever you start working on code, you should update the local repo from the remote repo. This will make sure that you are always working from the latest version of the code from GitHub. The opposite of pushing code to the repo is pulling it from the repo:

```
git pull
```

If there is a branch on GitHub that you want to work on, but it isn't on the computer you are currently on, you can create a local tracking branch with:

```
git fetch  
git checkout "name-of-remote-branch"
```

## Merging code from another branch

The primary way that you get code from one branch into another is with the `merge` command.

```
git checkout "destination-branch"  
git merge "source-branch"
```

`git` will try to integrate the new code into the current branch automatically, but that isn't always possible. If it can't figure out how to do the merge because there are conflicting edits, it will let you know that there has been a merge conflict. Managing and correcting merge conflicts is an advanced topic that we won't cover here.

### The takeaway

The more frequently you use git, the more benefit you will get from it. Pull from the remote when you start working on code, make branches for the work you are doing, create frequent commits to capture specific, granular changes. Finally, don't be afraid to ask for help.

---

## Lesson 3: Driving Lessons - Basic Teleop

The driver-controlled portion of the match is called **teleop**. During this part of the game, the driver is primarily in control of the robot and commands are relayed from controllers attached to the drive station to the robot through the FMS (field management system). Typically, the drive team consists of a driver, a mechanism operator, and a drive coach.

### Inputs

The primary way that the drive and mechanism operator control the robot is via a **joystick or controller**. Let's add a controller to the program so that we can operate the Romi.

#### **JAVA Concept: import**

Java and WPILib provide a lot of classes that you can use in your programs, but they aren't available by default in your code. In order to access them, you have to **import** them into the **namespace** of your program. You do this by using the **import** statement followed by the path to the class that you want to import. You must import classes into any file that needs them. Typically, **import** statements are located at the top of a **.java** file, just after the **package** statement.

**NOTE** Before making any changes to your code, create a new branch.

The first step is to **import** the class so that we can interact with the controller. The class that we're going to use is the **XboxController** class, which works with the Logitech controllers set to 'X' mode. It's located in **edu.wpi.first.wpilibj**. Add this line after the other imports at the top of the **Robot.java** file:

```
import edu.wpi.first.wpilibj.XboxController;
```

The next step is to create a new instance of the **XboxController** class. First, we have to define a variable that will hold the reference to the controller instance. We only want to have one reference to the joystick that we will re-use whenever we need to read values from the joystick. We'll do this inside the **Robot()** class before the constructor so that it is available to all the methods in the class.

Find the line that instantiates the **RomiDrivetrain()**:

```
private final RomiDrivetrain m_drivetrain = new RomiDrivetrain();
```

and add this line after it:

```
private XboxController m_controller = new XboxController(0);  
(1)---- (2)----- (3)----- (4)-----
```

This line defines a member variable and initializes it to a new instance of the **XboxController** class. The individual parts of this statement do the following:

1. `private` is an optional keyword that says this variable can only be accessed within this class. This prevents other classes from reading or altering our controller.
2. `XboxController` defines the `type` of the variable. Every variable must have a specific `type` in Java, which specifies what type of object it can hold.
3. `m_controller` is the name of the variable. This is how our code will access the instance.
4. `= new XboxController(0)` creates a new instance by calling the constructor of the `XboxController` class and assigns it to the variable. In this case, we pass the constructor the value 0, which indicates that the controller will be connected to channel 0 of the driver station.

Finally, we need to connect the values we read from the controller with the inputs to the drivetrain. This should be done in the `teleopPeriodic()` method of the `Robot` class. This method is automatically called once every 20 ms when the robot is in `teleop` mode. Any code you put in here will run 50 times/second, so this is the place for the code you will use to operate the robot in `teleop`.

Scroll down in the code until you see the definition of `teleopPeriodic()`. Currently this method does nothing. There is no code between the curly brackets `{` and `}`. Update the method to include the line between the curly brackets:

```
m_drivetrain.arcadeDrive(m_controller.getLeftY(), m_controller.getRightX());
```

Once you've done this, `teleopPeriodic()` should look like:

```
@Override
public void teleopPeriodic() {
    m_drivetrain.arcadeDrive(m_controller.getLeftY(), m_controller.getRightX());
}
```

## Test the code

Now you're ready to test the code.

First, power on the Romi and connect to its WiFi access point. You should also plug in an XboxController into your computer for controlling the Romi.

You start the simulation by clicking on the WPILib icon above the code window and selecting `WPILib: Simulate Robot Code`. This will compile your code and let you know if there are any errors. If the code compiles, then it will open a `dashboard` called `glass` that shows you information about the robot status.

## Dashboards

As mentioned above, when you start the code that operates the Romi, a `dashboard` will open on your screen that displays data about your robot. The dashboard that we use with the Romi is called `glass`, but there are also others such as SmartDashboard and Shuffleboard. The choice of dashboard depends on what the drivers and programmers prefer for their particular robot.

The information displayed is sent from the robot to the dashboard via a protocol called `Network Tables`. This protocol supports sharing information between the robot and the computer and can be used to both read

values from the robot and send values from the driver station to the robot. When you look at `glass`, you'll see that some information about the robot is already displayed. You can add other information that you want to display from within your program.

Open `RobotDrivetrain.java` and look through it. As you scroll through the code you may notice that, in addition to the motors, there are also encoders on each wheel. Encoders allow you to keep track of how much a wheel or other shaft has rotated. By applying a conversion factor to our Romi's encoders, we can convert the number of wheel rotations to a distance traveled by the wheel. The `RomiDrivetrain` class provides three methods to let us work with the on-board encoders:

```
public void resetEncoders() {
    m_leftEncoder.reset();
    m_rightEncoder.reset();
}
```

This does what it says and resets the encoders, which means that it causes them to reset to read 0. This can be useful when you want to measure how far the Romi has driven from a particular starting point.

```
public double getLeftDistanceInch() {
    return m_leftEncoder.getDistance();
}

public double getRightDistanceInch() {
    return m_rightEncoder.getDistance();
}
```

These two methods read the value from the left and right encoders respectively. The value is scaled to report the linear distance traveled by the wheel.

Let's display the distance driven by the Romi on our dashboard. Open your `Robot.java` file. We'll use the `SmartDashboard` class to send the data to our dashboard. This class is already imported, so we don't need to add an import statement for it.

Information that is about your robot and is relevant during every can be written to the dashboard via the `SmartDashboard.put...()` methods.

```
SmartDashboard.putNumber("Left Distance (in)",
m_drivetrain.getLeftDistanceInch());
SmartDashboard.putNumber("Right Distance (in)",
m_drivetrain.getRightDistanceInch());
```

```
stateDiagram-v2
    [*] --> s1
    s1 --> [*]
```



---

This is a note

**WIP**

outputs: motor configuration, drivetrains

inputs: joystick, buttons

driver station & dashboard

telemetry: sending data to dashboards (don't use print!)

<!--

## Lesson 4: Roaming Romi - Basic Auton

timed driving encoder driving

## Lesson 5: State-us Quo - Advanced Auton

(States and Catapults?) state machines complex auton sequences

## Lesson 6: Command and Conquer - Intro to Command Based

principles of Command Based event driven scheduler Subsystems and Commands composition, chaining, lambdas

## Lesson 7: Sense and Sensibility - Sensors and Feedback

-->