

Intro to WPILibJ

Tim Magoun and Aravind Koneru

Compiled on Saturday 11th June, 2016 at 13:03

1 Introduction

The WPI Library is a Java API that FIRST developed with the help of WPI (Worcester Polytechnic Institute) to help students write readable code for the robot. Despite being very high level code, the WPI Library allows users to have nearly full control over every aspect of the robot. The purpose of this document is to introduce you to the big ideas from WPI Lib and explain important concepts required for programming the robot. Rather than being a document detailing exactly what to type, this document outlines how the WPI Lib actually works, what type of code goes where, and the coding styles/methodologies required by the MidKnight Inventors.

2 Command Based Code

There are various ‘ways’ to write code for the robot. The standard for the MidKnight Inventors is to use the **Command Based** coding style. The main reason that we choose to write our code in a command based way is because it keeps the code modular, clean, and easy to debug. Also, command based code is the most flexible and allows users to easily create additionally functionality that isn’t already a part of the WPI Lib. That being said, the command based style of writing code is the most conceptually complex and requires a thorough understanding of what the code is doing at exactly every point.

3 The Master Classes

When you first make a new command based robot program, there are three default classes present in the `org.usfirst.teamXXXX.robot`. I refer to these three classes as the **master classes**

since they directly control what the robot is doing. All the other code ‘plugs’ into one of the three master classes.

3.1 Robot.java

Robot.java is both the simplest and most of the three master classes. Robot.java contains instructions for what the robot should be doing during every state that it could be in. Since we are using a command based system, all of our subsystems will be initialized in this class. You can think of the entire robot being in a constant state machine controlled by the **Field Management System** (FMS). Robot.java contains methods for all the different states that the robot could be in.

Let’s break down Robot.java to understand what it’s doing at every point. Here’s the code that you would see in Robot.java if you made a new project:

```
package org.usfirst.frc.team1923.robot;
...

public class Robot extends IterativeRobot {

    public static final ExampleSubsystem exampleSubsystem = new ExampleSubsystem();
    public static OI oi;

    Command autonomousCommand;
    SendableChooser chooser;

    public void robotInit() {
        oi = new OI();
        chooser = new SendableChooser();
        chooser.addDefault("Default Auto", new ExampleCommand());
        //  chooser.addObject("My Auto", new MyAutoCommand());
        SmartDashboard.putData("Auto mode", chooser);
    }

    public void disabledInit(){

    }

    public void disabledPeriodic() {
        Scheduler.getInstance().run();
    }
}
```

```

}

public void autonomousInit() {
    autonomousCommand = (Command) chooser.getSelected();

    /* String autoSelected = SmartDashboard.getString("Auto Selector", "Default");
    switch(autoSelected) {
        case "My Auto":
            autonomousCommand = new MyAutoCommand();
            break;
        case "Default Auto":
        default:
            autonomousCommand = new ExampleCommand();
            break;
    } */

    // schedule the autonomous command (example)
    if (autonomousCommand != null) autonomousCommand.start();
}

public void autonomousPeriodic() {
    Scheduler.getInstance().run();
}

public void teleopInit() {
}

public void teleopPeriodic() {
    Scheduler.getInstance().run();
}

public void testPeriodic() {
    LiveWindow.run();
}
}

```

Let's start by breaking down each method to see what it does.

3.1.1

```
public void robotInit()
```

When the robot is first turned on and enabled, it will run this set of instructions. `robotInit ()` is where you should declare all of your defaults and initialize and logging. The most important thing to remember is that `robotInit()` contains anything that you want the robot to do every time it's placed on the field. Let's say you wanted an intake to always start in the down position, then you would call that code here in `robotInit()`.

3.1.2

```
public void disabledInit()
```

Whenever the robot is disabled, this method is called. Although it's not used as often as `robotInit ()`, `disabledInit()` can have some important use cases. Occasionally, when the robot is disabled, a command may be in the middle of running or there might be an over-arching command that runs asynchronously that needs to be stopped. Stopping such a command would be done in `disabledInit()`. Additionally, when the robot is disabled there are sometimes changes that we want to make to the robot without actually pressing any buttons, this can be in the code fairly easily. Examples of such functionality would be making sure an arm is retracted all the way so that the drive team can easily move the robot from the field to the cart.

3.1.3

```
public void disabledPeriodic()
```

As the method name suggests, this code will be called periodically while the robot is disabled. This means that the code present in this method will run every few hundred milliseconds. While this method is rarely used by teams because it presents a safety issue, it can be used for novelty actions. If there are rgb leds on the robot, they can be controlled even while the robot is disabled by using this method. Once again however, this **method should not be used** without the **explicit permission** from the programming co-captain.

3.1.4

```
public void autonomousInit()
```

`autonomousInit()` is called at the beginning of **autonomous mode** (commonly referred to as auton). This function receives user input to determine which auton to run, and then runs said auton. It is important to make sure that this part of the code has fall-backs because if an auton choose or the desired input method doesn't work, then you will be left dead on the field for auton (which usually turns out to be a bad thing). As such, we require that there are fail safes in place to ensure that *an auton* will run even if it isn't the best one.

3.1.5

```
public void autonoumousPeriodic()
```

Similar to `disabledPeriodic()`, this method runs every hundred or so milliseconds during auton. Unlike `disabledPeriodic()`, this method can be used in any way that you please since it doesn't pose a safety problem.

3.1.6

```
public void teleopInit()
```

Whenever tele-operated (teleOp) mode is enabled, the FMS will call this method. You would use this method to default to a position after auton and use `robotInit()` to default to a position before auton. If there are sensors that need to be zeroed or solenoids that need to be activated before the start of teleOp but after auton, then this is where that code would.

3.1.7

```
public void teleopPeriodic()
```

When teleOp mode is enabled, this code is called periodically (every 20ms or so). If you need want to get back sensor readings, you would do those things in this method. Being able to see the status of current actions such as the position of an arm or the current readout of encoders is very important to the drive team since they won't always be able to see the robot from behind the glass. Also, running some sort of logging is useful while debugging problems so that you can see what exactly is going wrong.

3.1.8 `public void testPeriodic()`

This method isn't really important and usually remains empty. On the driver's station, there is an option for running the robot in a 'test mode' which just removes the timer. We don't really use this functionality of the driver's station on 1923 and we just tend to enable teleOp instead for testing purposes.

3.2 `OI.java`

`OI.java` is where you would declare and initialize all of your controllers and map specific actions of the robot to different functions. This class defaults to showing up empty and there are no special functions that are called periodically like in `Robot.java`. Let's go through an example mapping.

Disclaimer: If you can't get a general idea of what the code is doing from reading it (ie. creating a non-basic variable, constructors), you need to brush up on your basic java. The rest of the document is heavily focused on code and doesn't have as many conceptual ideas as above.

```
package org.usfirst.frc.team1923.robot;

import edu.wpi.first.wpilibj.Joystick;
import edu.wpi.first.wpilibj.buttons.JoystickButton;

import org.usfirst.frc.team1923.robot.commands.*;
import org.usfirst.frc.team1923.robot.utils.*;

/**
 * This class is the glue that binds the controls on the physical operator
 * interface to the commands and command groups that allow control of the robot.
 */
public class OI {

    public Joystick leftStick, rightStick;
    public JoystickButton leftTrigger, rightTrigger;

    public OI() {
        //Point 1
        leftStick = new Joystick(1);
        rightStick = new Joystick(2);
    }
}
```

```

    //Point 2
    leftTrigger = new JoystickButton(leftStick, 1);
    rightTrigger = new JoystickButton(rightStick, 1);

    //Point 3
    leftTrigger.whenPressed(new GearShiftCommand(true));
    rightTrigger.whenPressed(new GearShiftCommand(false));
}
}

```

3.2.1 Point 1

We declared two Joystick objects (`leftStick` and `rightStick`) earlier in `OI.java`, but we didn't initialize them. At this point in the code we are finally initializing our two Joystick objects and assigning them to usb ports.

```
leftStick = new Joystick(1);
```

This means that we are assigning our '*leftStick*' to usb port 1. This translates to mean that the joystick that will control the left side of the robot (assuming that we are using a drive system known as tank drive), must be plugged into port 1. Similarly, the joystick that will controlling the right side of the robot must be plugged into port 2. It's always a good idea to have these mappings written down somewhere on the driver's station or to test to make that the joysticks are plugged into their proper port before each match/practice.

3.2.2 Point 2

Once again, we are initializing two objects that we declared earlier in `OI.java`. This part of the code is pretty self-explanatory so if you don't understand what the parameters are or what is being initialized, please refer to the [WPILibJ Documentation on JoystickButtons](#).

3.2.3 Point 3

Until this point, we've just been initializing and declaring joysticks, controllers, and buttons in the code. Point 3 is where the actual work is being done to translate pressing a button into causing the robot to perform an action.

```
leftTrigger.whenPressed(new GearShiftCommand(true));
```

In english, this line is stating that when the leftTrigger button is pressed, call the GearShift-Command. In addition to **whenPressed**, there are a variety of options such as **whileHeld** or **whenReleased** to accommodate all types of control schemes. Thinking of the big picture, lines of code like this actually map actions to buttons.

In a real robot, there are going to be many more commands and many more button mappings, but the general idea remains the same so it is important to remember to name your commands and buttons in a manner that allows you to quickly make changes if needed.

4 Commands

5 Subsystem