Understanding the WPI Library

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 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:

```
k+knpackage n+nnorg.usfirst.frc.team1923.roboto;
o...

k+kdpublic k+kdclass n+ncRobot k+kdextends nIterativeRobot o

        k+kdpublic k+kdstatic k+kdfinal nExampleSubsystem nexampleSubsystem o= knew nExampleSu
        k+kdpublic k+kdstatic nOI noio;

    nCommand nautonomousCommando;
    nSendableChooser nchoosero;

    k+kdpublic k+ktvoid n+nfrobotInito() o
                noi o= knew nOIo();
        nchooser o= knew nSendableChoosero();
        nchoosero.n+naaddDefaulto(l+sDefault Autoo, knew nExampleCommando());
c+c1//        chooser.addObject(My Auto, new MyAutoCommand());
        nSmartDashboardo.n+naputDatao(l+sAuto modeo, nchoosero);
    o

    k+kdpublic k+ktvoid n+nfdisabledInito()

    o

    k+kdpublic k+ktvoid n+nfdisabledPeriodico() o
                nSchedulero.n+nagetInstanceo().n+naruno();
    o


    k+kdpublic k+ktvoid n+nfautonomousInito() o
        nautonomousCommand o= o(nCommando) nchoosero.n+nagetSelectedo();

                c+cm/* String autoSelected = SmartDashboard.getString(Auto Selector, Default);
c+cm                switch(autoSelected)
c+cm                case My Auto:
```

```
c+cm                        autonomousCommand = new MyAutoCommand();
c+cm                        break;
c+cm                case Default Auto:
c+cm                default:
c+cm                        autonomousCommand = new ExampleCommand();
c+cm                        break;
c+cm                 */

            c+c1// schedule the autonomous command (example)
        kif o(nautonomousCommand o!= k+kcnullo) nautonomousCommando.n+nastarto();
    o

    k+kdpublic k+ktvoid n+nfautonomousPeriodico() o
        nSchedulero.n+nagetInstanceo().n+naruno();
    o

    k+kdpublic k+ktvoid n+nfteleopInito() o

    o


    k+kdpublic k+ktvoid n+nfteleopPeriodico() o
        nSchedulero.n+nagetInstanceo().n+naruno();
    o

    k+kdpublic k+ktvoid n+nftestPeriodico() o
        nLiveWindowo.n+naruno();
    o
o
```

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

3.1.1

```
k+kdpublic k+ktvoid n+nfrobotInito()
```

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

```
k+kdpublic k+ktvoid n+nfdisabledInito()
```

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

```
k+kdpublic k+ktvoid n+nfdisabledPeriodico()
```

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

```
k+kdpublic k+ktvoid n+nfautonomousInito()
```

`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

```
k+kdpublic k+ktvoid n+nfautonoumousPeriodico()
```

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

```
k+kdpublic k+ktvoid n+nfteleopInito()
```

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 senors 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

```
k+kdpublic k+ktvoid n+nfteleopPeriodico()
```