Lesson 4: Team Standards Java Edition

Updated January 2025

Table of Contents

Introduction

This document outlines the general standard practices for programming that Big MO has developed over the last several years. It covers everything from how code should be organized to how names should be structured. These standards are in place to streamline the process and make it as consistently readable as possible, without compromising functionality, no matter who wrote it. This document assumes an understanding of all concepts included and will not be explaining how they work, only how we as a team use them.

Any examples of code within this document are written and colored as closely as they can be to how it will appear in VS Code, the program we use for writing the actual code. Sections surrounded by angle brackets such as <type> are required placeholders while sections surrounded by square brackets such as [argument] are optional placeholders, typically to be replaced with references to other parts of code like specific classes, variable names, or direct values.

Section 1 - Formatting and Structure

1.1 Packages

One method to organize your files is through packages. Make all packages with short but concise names. All of our classes are broken into various packages and there are several we will use every year such as subsystem for any mechanical subsystems, or period for all of the control periods during the game.

1.2 Classes

Almost every class will contain the functions: init(), initDashboard(), updateDashboard(), and periodic(). This is standard to initiate anything at startup and to continuously update the class. Typically, every class will contain its enumerations and other constants first, followed by all of its dashboard elements, private variables, the Constructor function (often private and empty as most classes will be accessed statically and will handle any initialization in the init() function instead), the init(), initDashboard(), and updateDashboard() functions, any private or protected functions, all public functions, and finally the periodic() function. Try to group related functions to make it easier to navigate and read your code.

1.2a Robot

If done correctly, the Robot class should be very bare-bones. Typically, all it should contain is the creation of the MO Data, Control Periods, and Subsystems NetworkTables, the control period and subsystem classes' init(), intiDashboard(), updateDashboard(), and periodic() functions, Camera creation/configuration, and the ButtonScheduler updateValues() function. All other tasks should be constrained to their respective sections; Teleoperated should handle all included subsystems' periodic() functions, Chassis should handle all motor configuration within its init() function, and so on.

1.3 Functions

A function definition must include its visibility (public, private, or protected), a return type, an appropriate name and any applicable parameters.

```
public void setDrivePower(double powerLeft, double powerRight) {
     mtrDrive_L1.set(powerLeft);
     mtrDrive_R1.set(powerRight);
}
```

Note the space between the arguments and the opening brace, and how the following lines are tabbed over. No matter the contents, all functions should be formatted this way. The only exception is when the body contains a single line. In some cases where this happens, it is acceptable to write the whole function on the same line.

```
public int getCount() { return mCount; }
```

Note the continued use of spaces to separate the brackets from the nearby code, especially when it's so short. This makes it much easier to read when referencing this code later.

1.4 Class Variables

Every variable belonging to a class must include its visibility (public, private, or protected), its type, an appropriate name, and a starting value. There are only a few exceptions where the variable will not have a starting value assigned to it, typically when a value will be passed through the constructor.

```
public int count = 0;
```

1.5 Enumerations

1.5a Simple Enumerations

Enumerations are used in two ways. One is very simple and is just a list of possible values essentially. In this case they will be just a list and nothing more. Typically, the entire enumeration can be listed on a single line. If the list is rather long however, it is recommended to list them each individually.

```
enum Speed { HIGH, MEDIUM, LOW }
enum Colors {
     RED,
     ORANGE,
     YELLOW,
     GREEN,
     BLUE,
     PURPLE,
     BLACK,
     GRAY,
     WHITE,
     PINK,
     BROWN
}
```

1.5b Complex Enumerations

In rare cases, enumerations will need additional data to be stored in them. One such instance in the past was created to store different angles of an arm. In this case, the enumeration is also treated like a standard class. These are used when you want to force the user to use an enumeration to select different options without directly dealing with the data stored within them.

At the beginning, the list is much the same except each element in it starts with the values to be retrieved later. Following the list, we create the public but final variables to store this data and create a constructor to pass these values into the variable

```
enum ArmAngles {
     REVERSE_FLOOR(-10.0),
     REVERSE_LOW(-30.0),
     REVERSE_HIGH(-45.0),
     CENTER(0.0),
     FORWARD_HIGH(45.0),
     FORWARD_LOW(30.0),
     FORWARD_FLOOR(100.0);

     public final double angle;
     ArmAngles(double angle) { this.angle = angle; }
}
```

1.6 Driver Station Dashboard

1.6a SmartDashboard

Within the WPILibrary, SmartDashboard provides the ability to push and pull data from the driver station dashboard. In order to organize these however, the complete path name must be provided to access them each time you reference them. It is recommended that you avoid these unless absolutely necessary as the classes and functions provided in MOLib are much easier to use.

1.6b MOLib DashboardValue and DashboardSelector

Included with MOLib is an alternative way of connecting to Network Tables to communicate information to and from the Driver Station. In order to make use of it, a table must first be created. The original parent table should be named “MO Data” and subtables can then be created from them, and more from those, as many as necessary. Other common subtables are “Control Periods” and “Subsystems” under which the respective classes will make their own subtables. This ultimately makes it easier to find the specific information you are looking for in what will typically become a very long list. From there, Entries and Options can be made to actually store the data.

Section 2 - Naming Conventions

2.1 Package Names

Packages are used to break up and organize files. The names for them are very simple on purpose with a single word, all lowercase. The name should designate either a broad umbrella of other packages or a very specific group of files.

2.2 Class Names

Class names are written in Pascal Case, meaning that the first letter of every word is capitalized and there are no spaces. However, class names should typically be restricted to a single word if possible. Often, if more words are necessary to make it distinct, the similar classes should all be moved into a separate package. Chassis, Teleoperated, and Intake are all good examples.

2.3 Function Names

Names for functions are written in Camel Case, meaning that the first letter of each word is capitalized, except the first one which should be left lowercase. They should be short, but descriptive, and should be consistent. This means that similar functions throughout various classes and packages should all follow the same structure.

Common functions are getters and setters. These are simply named “get” or “set” followed by what information they are tied to, such as getCount() or setPower(). Other common functions are named for the specific action they perform such as raiseArm() or disableIntake(). In cases like this, it is best to be as consistent and descriptive as to what action is being performed as possible. Most commonly this can be one of the following: raise, lower, extend, retract, enable, disable, or reverse. These should be followed by the specific mechanism being manipulated unless being used for the entire subsystem. The exception to this rule is setters for configuration purposes. These will be prefixed with “config” such as configDeadzone() for a controller.

2.4 Variable Names

Depending on the context of the variable, the naming scheme will be slightly different. Variables of simple types (int, double, boolean, String, etc) are written in Camel Case, meaning the first letter of each word is capitalized, except for the first word. Private ones are prefixed with the letter “m” to indicate that they are private. This counts as the first word and all subsequent words will be capitalized. Constants of simple types are written entirely in capital letters and words are separated by underscores.

Complex Objects such as devices on the robot or driver controllers, whether final or not, have a special convention. All of them are prefixed with a three letter abbreviation indicating what kind of object it is. This is followed by a single word description of its use, one as specific as possible is best. A drivetrain motor would use Drive, a limit switch for the arm in a manipulator would only use Arm, and so on.

A complete list of object prefixes

Button	btn	LED Lights	led
Camera	cam	Limelight	lml
Compressor	cmp	Limit Switch	lim
Driver Controller	ctl	Motor Controller	mtr
Dial	dia	Photo Eye	pho
Digital Input	dgi	PID Controller	pid
Digital Output	dgo	Potentiometer	pot
Dashboard Entry	dsh	Solenoid	sol
Encoder	enc	Network Table	tbl
Gyro	gyr	Timer	tmr
Jumper	jmp		

If there are multiple objects with a similar purpose, they are designated with either a letter representing a position, a number representing order, or a combination thereof. This designation is separated from the previous part with an underscore. Suffixes with the same letter should never actually conflict as they will not be used in the same context.

A complete list of directional suffixes

Left	L	Top	T
Right	R	Bottom	B
Center	C	Upper	U
Front	F	Lower	L
Back	B		

The only exceptions are Dashboard Entries and PIDs. These objects should append their specific purpose as well, such as _Speed, _Distance, _Position, _Power, or _Enabled. Some examples of objects with varying degrees of complication are: mtrDrive_L1, limArm_U, pidDrive_L_Distance, dshArm_Speed.

2.5 Enumeration Names #EDIT

Enumerations are treated like standard classes containing a list of constants. The name of the enumeration is as short, but descriptive as possible and written in Pascal Case meaning that the first letter of each word is capitalized. The elements of the enumeration are considered constants and should also be as short but descriptive as possible, typically a single word or two. These are in all caps, words separated by underscores. For complex enumerations with variables and functions in them, the standard conventions above apply to these variables and functions as well.

Section 3 - Comments

3.1 Standard Comments

Comments are used to describe what is happening at specific points in code. Explain confusing sections in as much detail as possible because it might make sense in the moment, but two weeks later it will likely be a different story. Plus it will make it much easier to understand when anyone else looks it over for any reason. Single line comments are denoted by two backslashes. Everything following on that line is considered a comment and is no longer part of the code.

```
//Single line comments are good for short descriptions
```

Multi-line comments are denoted by starting with `/*` and ending with `*/`. Unlike single line comments, only what is between these characters is considered a comment. As such they can be inserted anywhere in code, but typically should be restricted to blocks in between sections of code.

```
/*
 * Multi-line or Block comments are used when a longer
 * description is necessary or when sections need to be
 * separated a bit more.
 */
```

Note the spacing used in the multi-line comment. Lining the asterisks up and spacing the text evenly away from them makes it a lot easier to read.

3.2 JavaDoc Comments

JavaDoc comments are special multi-line comments that are typically tied to specific functions. The only written distinction is the use of two asterisks at the start rather than the one used in standard multi-line comments. They are used by the editor to create a popup when these functions are used, providing extra detail and help with using them.

Within them, a detailed explanation of the functions' purpose goes first. Be as descriptive as possible with this part, explain it as you would to another person who has never read any of your code. After the description, use the various tags to describe the other elements of the function. The @param tag is used to describe any parameters passed into the function. Any parameters that should be within a certain range such as those for motor powers, should start with their minimum and maximum value in square brackets. Similarly, the @return tag is to explain what is returned from the function when applicable. Use @see when there is an outside, but related, element that the reader should look into such as a special return type or related class.

```
/**
 * A detailed description of what the following function does.
 * This part should be the longest and could take several lines.
 * Also supported are HTML tags such as <p> </p> to specifically
 * designate paragraphs as any new lines or other separations
 * are not reflected unless explicitly stated through the use of
 * these tags.
 * @param <parameter>    Brief description of the parameter.
 * @param <parameter>    [min, max] Followed by description.
 * @return               Brief description of what is returned.
 *
 * @see                  <reference>
 */
```

Note that these are setup almost identical to standard multi-line comments where everything is spaced and aligned vertically. Long sections of the main description are deliberately broken up into multiple lines to prevent them from running off the screen. Also note the use of tabs after tags as this further aligns the descriptions and makes it cleaner and easier to read. There should be a blank line before any @see tags that follow at the end to break it up from one large block of text. {@link <reference>} can be used to direct users to other sections of related code within the comment itself. Use of this vs the @see tag is up to the discretion of the author.

3.3 Miscellaneous Comments

There are a few keywords that you can throw into comments that can further help you understand what needs to be done. A TODO tag can be inserted anywhere in a comment and everything that follows on that line will be included in the Problems tab on VS Code. Similarly, you can use FIXME to indicate something that needs to be fixed in the future.

```
//TODO: Description of what needs to be done.
//FIXME: Description of what needs to be fixed.
```