

COMMAND BASED PROGRAMMING

How to organize your robot code effectively



D-Bug #3316

OVERVIEW

- What Is command based programming?
- Subsystems
- Commands
- Command groups
- The command scheduler
- Structuring a command based robot project
- Joysticks and HumanIO



WHAT IS COMMAND BASED PROGRAMMING?

The basics



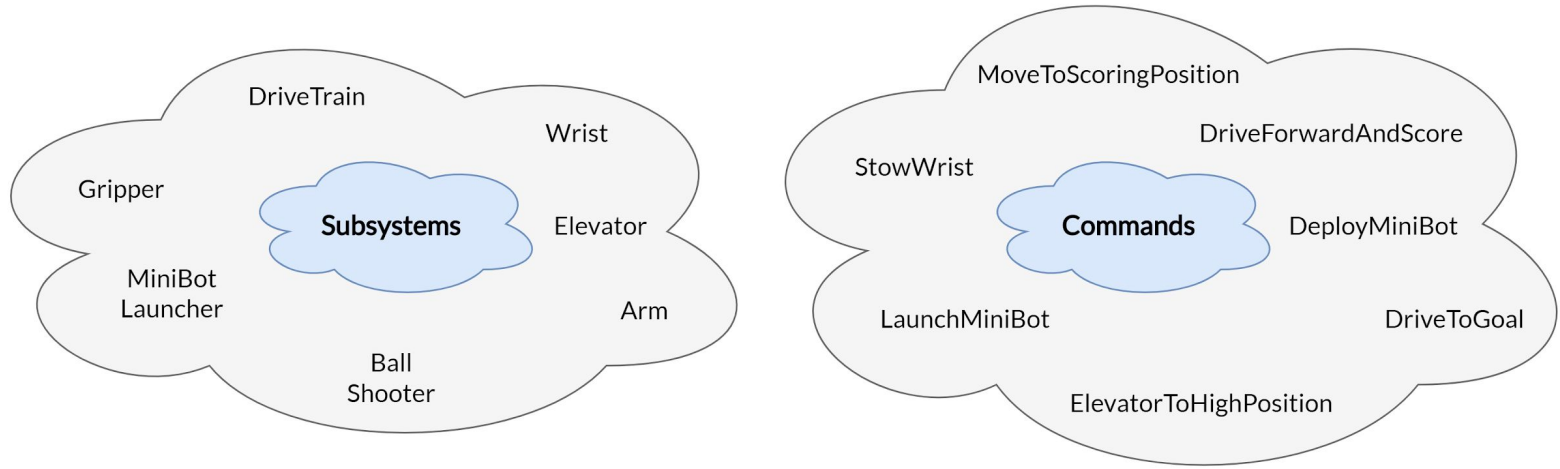
COMMAND BASED PROGRAMMING



Command based programming is a design pattern, a general way to design the structure of your robot code.

There are 2 core concepts in command based programming: **Subsystems** and **Commands**.





Each command is linked to at least one subsystem.

Each subsystem can (usually) run only one command at a time.



NEW PROJECT

Get your VS Code ready

Open VS Code and create a new **command based** project.



Welcome to WPILib New Project Creator

template java Command Robot

Select a folder to place the new project into.

C:\Users\User\Documents

Select a new project folder

Create new folder? Highly recommended to be checked

Enter a project name

command-based-programming-yourname

Enter a team number

3316

This is needed for simulation and unit testing support, however there are some cases where this will do some unexpected things during build. In addition, not all vendor libraries support desktop. This option can be set with the command "WPILib: Set Desktop Support" at any time.

Generate Project

SUBSYSTEMS

Our first building block



SUBSYSTEMS



In a nutshell, subsystems are the software equivalent of hardware mechanisms on the robot.

They encapsulate all the actuators and sensors which make up a mechanism into a single unit which can be easily accessed.

Here are some examples of subsystems we used for Mercury in 2019:

- Drivetrain
- Elevator
- Panel Mechanism
- Cargo Intake
- Cargo Ejector



SUBSYSTEM STRUCTURE



```
public class ExampleSubsystem extends SubsystemBase {  
    public ExampleSubsystem() {  
        // Constructor stuff goes here  
    }  
  
    @Override  
    public void periodic() {  
        // This method will be called once per scheduler run  
    }  
}
```



ADDING SOME HARDWARE

Private as we want to encapsulate the hardware

Initialized in the constructor

```
public class ExampleSubsystem extends SubsystemBase {
    private DBugVictor _victor;

    public ExampleSubsystem() {
        this._victor = new DBugVictor(0);
    }

    @Override
    public void periodic() {
        // This method will be called once per scheduler run
    }
}
```



PUBLIC METHODS

Public methods allow us to control the subsystem's components from the outside and to receive data about its current status.

```
public double getOutput() {
    return this._victor.getMotorOutputPercent();
}

public void setOutput(double percent) {
    this._victor.set(ControlMode.PercentOutput, percent);
}
```



CODING TIME

Making a basic subsystem

Create an elevator subsystem

- The elevator has 3 states: **TOP**, **INTERMEDIATE** and **BOTTOM**. *tip: use an enum*
- Define the following sensors and actuators:

Object	Type	PORT
VictorSP	PWM	0
Encoder	DIO	0, 1

- The subsystem has 3 public methods:
 - `setState` - sets a state and sets the victor to the correct output
 - `zeroMotor` - sets victor output to zero
 - `getState` - returns the current state of the elevator (assume that 0 ticks from the encoder = **BOTTOM**, and 10 tick = **TOP**)

COMMANDS

Building upon our subsystems



COMMANDS



At its core, a command is an action performed by a subsystem.

The command uses our subsystem's public methods to perform these actions.

Here are some examples of commands we wrote in 2019 for the Cargo Intake mechanism:

- CargoIntakeOpen
- CargoIntakeClose
- CargoIntakeSetRollers




COMMAND STRUCTURE



We start our command with a constructor.

After Which we have 4 default methods:

- Init
 - Execute
 - Is Finished
 - Fin
- 

```
public class SetArmState extends CommandBase {
    private ArmState _wantedState;
    private final ArmSubsystem _armSubsystem;
    public SetArmState(ArmSubsystem armSubsystem, ArmState wantedState) {
        // This line will be explained later
        this._armSubsystem = armSubsystem;
        addRequirements(this._armSubsystem);

        this._wantedState = wantedState;
    }

    // Default methods go here...
}
```

1/4 INIT

The `init()` method is called once when the command is first scheduled

```
@Override
public void init() {
    this._armSubsystem.setArmState(this._wantedState);
}
```



2/4 EXECUTE

The `execute()` method is called repeatedly while the command is running.

```
@Override  
public void execute() {  
    // Empty in this example  
}
```



3/4 IS FINISHED

The `isFinished()` method determines whether our command should continue running or not.

The method is called repeatedly while the command is running similar to the execute method.

When the method returns **true** fin is called and the command ends, but if **false** is returned the command will continue as usual.

```
@Override
public boolean isFinished() {
    return this._armSubsystem.getArmState() == this._wantedState;
}
```



4/4 END

The `end()` method is called once when the command ends.

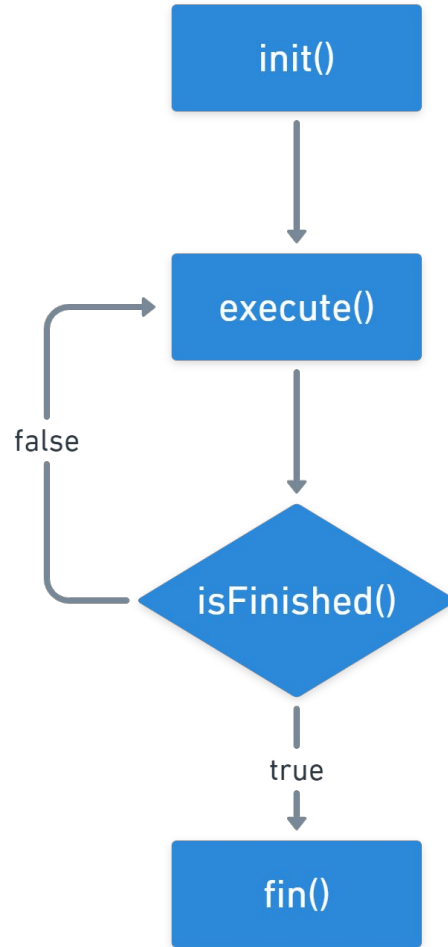
The value of `interrupted` depends on whether the command was interrupted, either by another command or by being explicitly canceled.

We can use this value to react differently when the command was interrupted.

```
@Override
public void end(boolean interrupted) {
    if (!interrupted) System.out.println("All good");
    else System.out.println("INTERRUPTED!");
}
```



COMMAND FLOW



ADVANCED COMMAND FEATURES

Requirements and Default Commands



REQUIREMENTS



Usually you'd want only one command to run on each subsystem at a time.

For example to prevent two commands from setting the same motor to different outputs at the same time.

For that reason we have requirements, when several commands require the same subsystem only one of them can run at a time.

Commands can also require more than one subsystem (see command groups section).



ADDING REQUIREMENTS

```
public class ExampleCommand extends DBugCommand {  
    private final ExampleSubsystem _exampleSubsystem;  
    public ExampleCommand(ExampleSubsystem exampleSubsystem) {  
        this._exampleSubsystem = _exampleSubsystem;  
        addRequirements(this._exampleSubsystem);  
    }  
}
```

We can pass `addRequirements` more than one subsystem if needed

We define our subsystems in `RobotContainer.java` and pass them as arguments like so



DEFAULT COMMANDS



Default commands run automatically whenever a subsystem is not being used by another command.

A good example is the drivetrain subsystem, where if no command is running we'd like to return control of the drivetrain to the human driver by calling the TankDrive command.

Setting default commands is done like so:

```
exampleSubsystem.setDefaultCommand(exampleCommand);
```



CODING TIME

Time to command

Create the SetElevatorState command. Make sure it:

- Requires the elevator subsystems
- Receives a wanted state in its constructor
- Sets the elevator to the wanted state
- Finishes only when the desired state is reached

Next add some prints like “*SETTING ELEVATOR FROM X TO Y*”.

Add a public get method for the command in RobotContainer. Then, schedule the command in teleopInit and run the simulation.

Note: to schedule a command call its schedule method

```
m_robotContainer.getSetElevatorCommand(state).schedule();
```

COMMAND GROUPS

The true power of commands



COMMAND GROUPS



A command groups is made up of two or more commands in a certain order.

Command groups allow us to run multiple commands simultaneously or one after the other.

Examples:

- Collect
- Shoot
- Eject

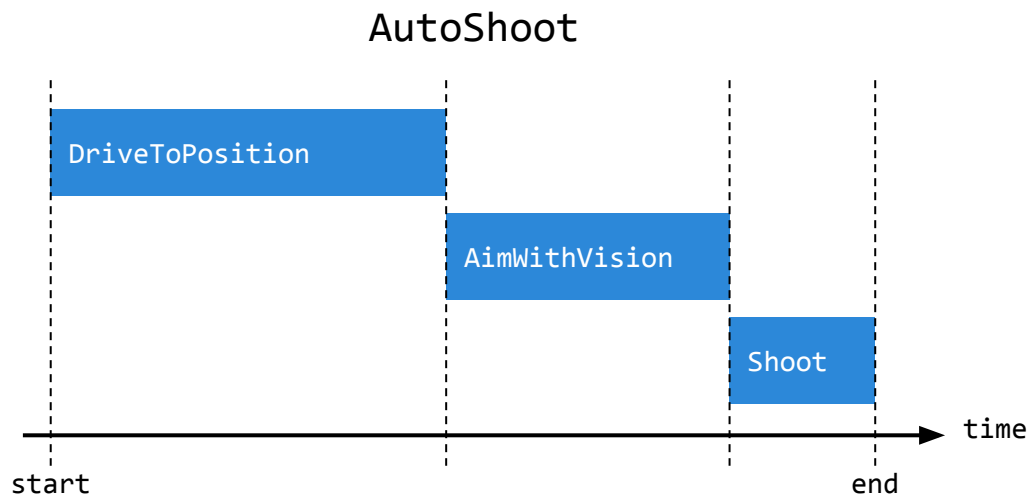
WPILIB offers 4 types of command groups.



1/4 SEQUENTIAL GROUP

A sequential command group runs a list of commands in sequence.

It ends after the last command in the sequence finishes.



CODE EXAMPLE

```
public class ExampleSequential extends SequentialCommandGroup {  
    public ExampleSequential() {  
        addCommands(  
            new RunsFirst(),  
            new RunsSecond(),  
            new RunsThird()  
        );  
    }  
}
```

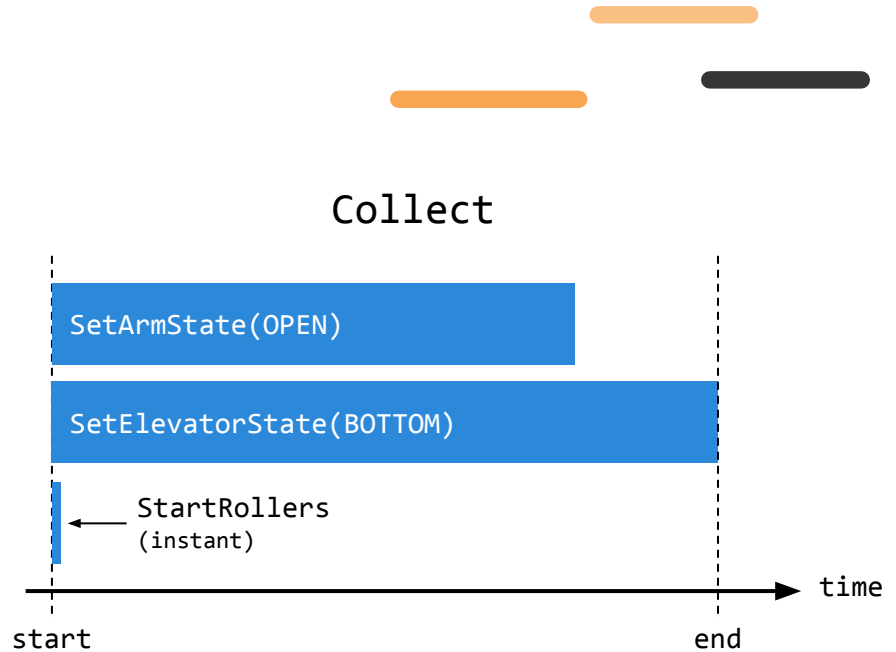
```
// Schedule the command group like you do with a command  
new ExampleSequential().schedule();
```



2/4 PARALLEL GROUP

A parallel command group runs a set of commands at the same time.

It ends when all commands have finished.



CODE EXAMPLE

```
public class ExampleParallel extends ParallelCommandGroup {
    public ExampleParallel() {
        addCommands(
            new TheOrderDoesntMatter(),
            new AllCommandsHere(),
            new RunSimultaneously()
        );
    }
}
```



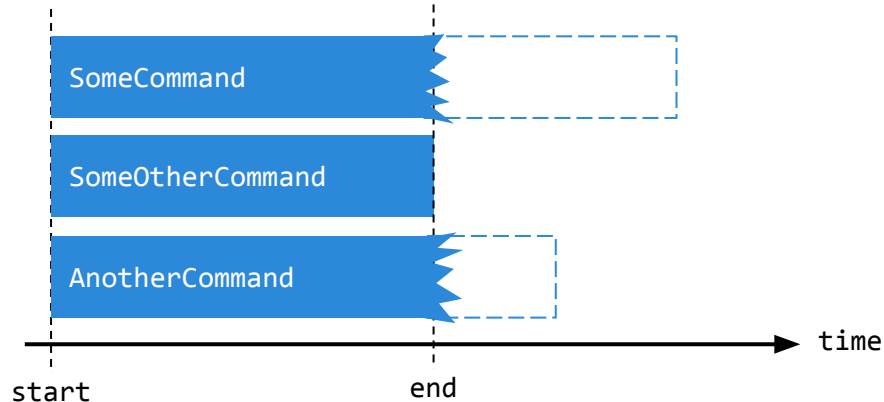
3/4 PARALLEL RACE GROUP



The parallel race group is much like a parallel command group.

However, the race group ends as soon as any command in the group ends, all other commands are then interrupted.

Parallel Race Group Example



CODE EXAMPLE

```
public class ExampleParallelRace extends ParallelRaceGroup {
    public ExampleParallelRace() {
        addCommands(
            new AllCommandsRunSimultaneously(),
            new WhenACommandEnds(),
            new AllOtherEndAsWell()
        );
    }
}
```

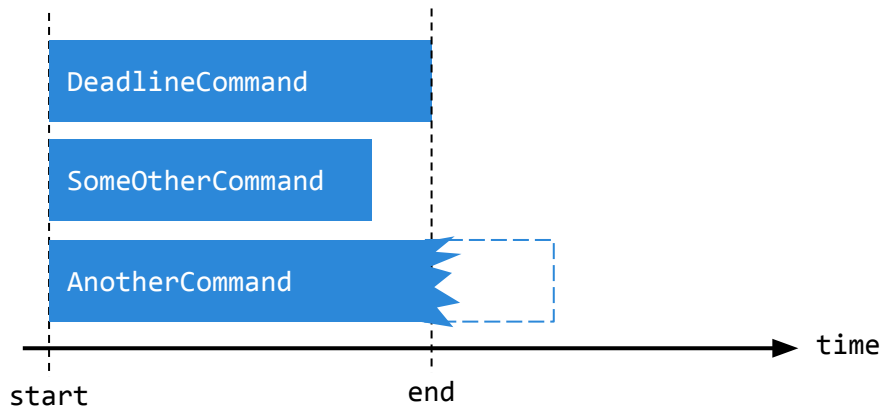


4/4 PARALLEL DEADLINE GROUP

The parallel deadline group is also similar to the parallel command group.

However, the deadline group ends when a *specific* command (the “deadline”) ends, all other commands are then interrupted.

Parallel Deadline Group Example



CODE EXAMPLE



First command is always
deadline command



```
public class ExampleParallelDeadline extends ParallelDeadlineGroup {  
    public ExampleParallelDeadline() {  
        super(  
            new DeadlineCommandFirst(),  
            new AllOtherCommands(),  
            new ComeAfterwards()  
        );  
    }  
}
```



NESTED COMMAND GROUPS

Command groups are also commands. As such we can put one in place of a regular command to make nested command groups.

Does the same and
easier to read

```
public class ExampleNestedGroup extends SequentialCommandGroup {
    public ExampleNestedGroup() {
        addCommands(
            new RunsFirst(),
            parallel(
                new BothRun(),
                new Together()
            ),
            new RunsThird()
        );
    }
}
```

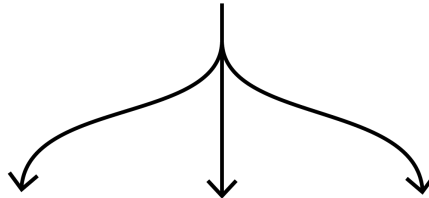


THE FULL FLOW

Human player presses shooting button



Shooting command group called



Rotate
Pizza

Spin
Shooter

Rotate
Extractor



Pizza
Subsystem

Shooter
Subsystem

Extractor
Subsystem

- Command Group
- Commands
- Subsystems



CODING TIME

Grouping commands

Object	Type	PORT
Elevator VictorSP	PWM	0
Intake Arm VictorSP	PWM	1
Intake Roller VictorSP	PWM	2
Encoder	DIO	0, 1
Intake Arm Open MS	DIO	2
Intake Arm Closed MS	DIO	3

Create 2 new subsystems: IntakeArm and IntakeRollers.

IntakeArm:

- Has 3 states: **OPEN**, **CLOSED** and **INTERMEDIATE**
- Has one victor and two micro switches: one to detect the OPEN state and one for CLOSED state
- Its public methods are: setState, getState and zeroMotor

IntakeRollers:

- Has 3 states: **IN**, **OUT** and **OFF**
- Has one victor
- Its public methods are getState and setState

YOUR VERY OWN COMMAND GROUP



Next create two commands: **SetIntakeArmState** and **SetIntakeRollersState**. Their structure is like **SetElevatorState**. Remember to add prints!

Finally we can build the command group. Create a new Sequential command group called **Collect** which does the following:

1. Sets elevator state to BOTTOM
2. Sets arm state to OPEN and rollers state to IN simultaneously
3. Print “Command Group Finished!”

Run the simulation and see the results.



THE COMMAND SCHEDULER

Taking a look under the hood



THE COMMAND SCHEDULER



The Command Scheduler is the class responsible for actually running commands. Some of its tasks include:

- Adding newly scheduled commands
- Running already scheduled commands
- Removing finished or interrupted commands
- Running subsystem periodic() methods
- Check if a button has been pressed (more on that later)



USING THE COMMAND SCHEDULER



The command scheduler is a singleton, we can access it like so: `CommandScheduler.getInstance()`.

With the scheduler object we can cancel commands, get the time passed since they were scheduled and more, but more importantly we can control the scheduler itself.

For example, in robot periodic we call `CommandScheduler.getInstance().run()` to run the scheduler.

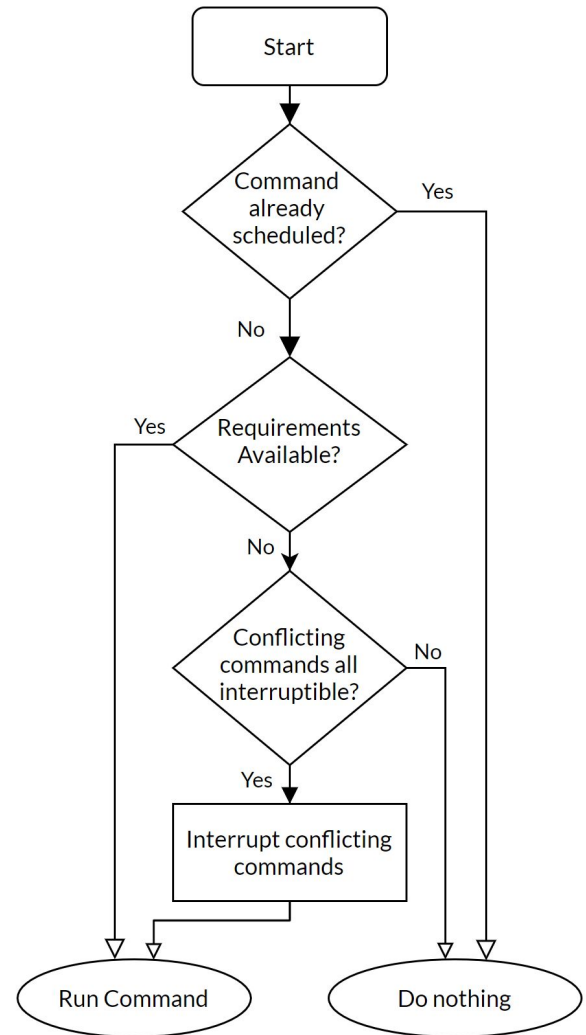


THE SCHEDULE METHOD

In order to schedule commands we use it's schedule method like so:

```
new ExampleCommand().schedule();
```

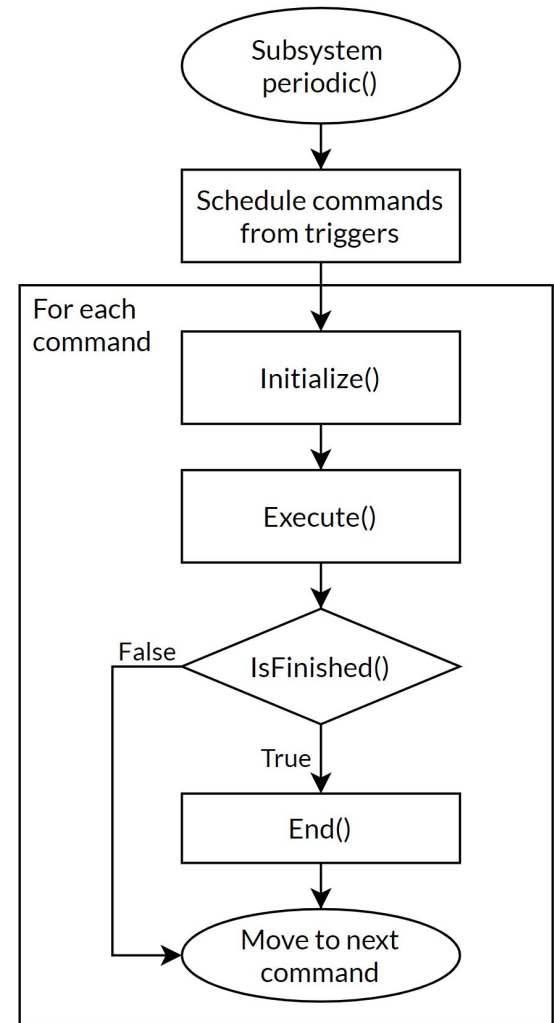
On the right we see what actually happens inside the scheduler object when we schedule a new command.



THE SCHEDULER RUN SEQUENCE

Each iteration the scheduler performs the following sequence:

1. Run each subsystem's periodic method
2. Schedule commands from triggers (more on that later)
3. Run one iteration of each command's flow
4. Schedule default commands



PROJECT STRUCTURE

Structuring a command based robot project



FOLDER STRUCTURE

- We keep subsystems, command and command groups each in their respective folder
 - Joysticks related stuff go to Joysticks.java
 - We define and call init from Robot.java
 - Constants (like port numbers or fixed values) go to Constants.java
-

```

  ▾ java
    ▾ frc \ robot
      ▾ commandGroups
        ● ExampleCommandGroup.java
      ▾ commands
        ● ExampleCommand.java
      ▾ HumanIO
        ● Joysticks.java
      ▾ subsystems
        ● ExampleSubsystem.java
        ● Constants.java
        ● Main.java
        ● Robot.java
    > kit

```



JOYSTICKS.JAVA

```
public class Joysticks {
    private Joystick _leftJoystick, _rightJoystick;
    private XboxController _operatorJoystick;

    public Joysticks () {
        this._leftJoystick = new Joystick(Constants.Joysticks.LeftJoystick.port);
        this._rightJoystick = new Joystick(Constants.Joysticks.RightJoystick.port);
        this._operatorJoystick = new XboxController(Constants.Joysticks.OperatorJoystick.port);
    }

    exampleButton.whenPressed(new ExampleCommand());
}

public JoystickButton getOperatorButton(Button button) {
    Return new JoystickButton(this._operatorJoystick, button.value)
}

// Useful public methods go here (getLeftStickY() for example)
}
```



ROBOTCONTAINER.JAVA

- Declare subsystems and joysticks
- Define subsystems
- Define and init joysticks

```
public class RobotContainer {  
    // Subsystems  
    private ExampleSubsystem _exampleSubsystem;  
  
    // HumanIO  
    private Joysticks joysticks;  
  
    public RobotContainer() {  
        // Define subsystems  
        _exampleSubsystem = new ExampleSubsystem();  
  
        // Define and init joysticks  
        joysticks = new Joysticks();  
        configureButtonBindings();  
    }  
}
```



JOYSTICKS & HUMAN I/O

Triggering some commands



BINDING COMMANDS TO TRIGGERS

We bind commands to joystick buttons in order to activate them.



```
// Step 1. Define the joystick
XboxController operatorJoystick = new XboxController(0);

// Step 2. Define the button
JoystickButton xButton = new JoystickButton(operatorJoystick, XboxController.Button.kX.value);

// Step 3. Bind button
xButton.whenPressed(new InstantCommand(() -> System.out.println("X BUTTON PRESSED!")));
```

Joystick port

Button number

Binding method



BUTTON BINDING METHODS

```
button.whenPressed(command)
```

Called once when button is first pressed

```
button.whenReleased(command)
```

Called once when button is released

```
button.whileHeld(command)
```

Called repeatedly while button is held

```
button.toggleWhenPressed(command)
```

Called once when button is first pressed and gets interrupted after the second press

[Full list of bindings can be found here](#)



CONVENIENCE FEATURES

Some extra stuff to ease your life



THE DOUBLE COLON OPERATOR ::

WPILIB provides some prebuilt tools that can make commands easier to write and understand.

To use them we'll first learn about the **double colon operator (::)** in java.

We use it to pass methods as parameters.

That way they can be later called by other parts of the code when needed.

```
command.withInterrupt(limitSwitch::get);
```

Object containing method

The method



() -> LAMBDA EXPRESSIONS

For some cases we have to use “::”, but for simpler cases it’s easier to use lambda expressions.

```
// Needlessly long way
void sayHi() {
    System.out.println("Hi!");
}
new InstantCommand(this::sayHi);
```

```
// Short and easy way
new InstantCommand(() -> System.out.println("Hi!"));
```



INCLUDED COMMAND TYPES

WPILIB includes pre-written commands to help with common use cases. For example:

- `InstantCommand` - executes a single action on initialization, and then ends immediately
- `PerpetualCommand` - runs a given command with its end condition removed, so that it runs forever (unless externally interrupted)

And some which are useful in command groups:

- `WaitCommand` - does nothing, and ends after a specified period of time elapses
- `WaitUntilCommand` - does nothing, and ends once a specified condition becomes true

[Full list of included commands here](#)



COMMAND DECORATOR METHODS

We can add additional functionality to commands by using some of their included methods.

```
Command justTheCommand = new ExampleCommand();
Command commandWithTimeout = justTheCommand.withTimeout(5);

// Will be interrupted 5 seconds after being scheduled
commandWithTimeout.schedule();
```

[Full list of command decorators here](#)

Some notable decorators include:

- `withInterrupt` - adds a condition on which the command will be interrupted
- `andThen` - adds a method or command to be executed after the command ends
- `alongWith` - returns a parallel command group containing the command, along with all the other commands passed in as arguments



FURTHER READING

Knowledge is power



FURTHER READING

- [Command-Based Programming — FIRST Robotics Competition documentation](#)
- [Coding exercise github repo](#)

