This page is part of BoVLB's FRC Tips. Find this page online at https://bovlb.github.io/frc-tips/commands/

Commands

Although you can avoid it in some simple cases, doing anything complex with your FRC robot will involve creating commands. These respond to joysticks and buttons, run your autoonomous routines, and do other maintenance tasks.

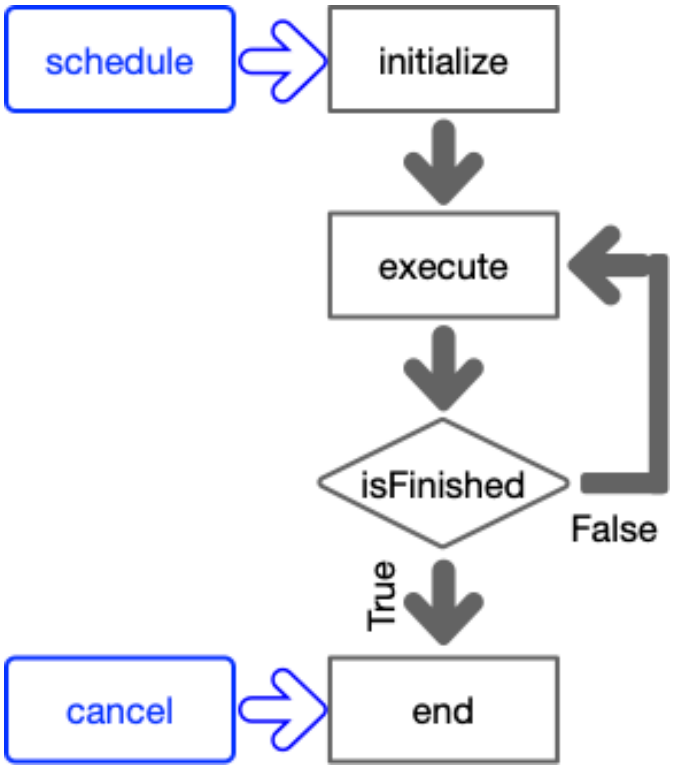
In addition to the usual constructor, commands have four lifecycle methods: `initialize`, `execute`, `isFinished`, and `end`. These methods are called by the `command scheduler` (and never by you). By overriding the implementation of these methods, you can change the behaviour of the command.

void initialize()

- Called once whenever a command is scheduled (including default commands).
- Use this to do anything your command needs to do once, such as running motors at constant speed, or initializing variables. It's a good idea to print a log message in this command.
- Default implementation does nothing.

void execute()

- Called every cycle for scheduled commands, in alternation with `isFinished`.
- Use this to do anything your command needs to do dynamically, like responding to joysticks or sensors, and updating internal state. You can also use this to update SmartDashboard (although that is usually better done in subsystem `periodic`)
- Default implementation does nothing.



The scheduler calls the four lifecycle methods of a command. This starts with `initialize` when the command is first scheduled, then `execute` and `isFinished` are called in alternation. Finally `end` is called either when `isFinished` returns trure, or when the command is interrupted.

boolean isFinished()

- Called every cycle for scheduled commands, in alternation with `execute`.
- Use this to tell the scheduler when your command is complete.
- Default implementation in `Command` returns `false`, so a command will run forever unless interrupted or canceled, but may be overridden (say in `InstantCommand`).

void end(boolean interrupted)

- Called when a command is descheduled, which means that `isFinished` has returned `true`, or that the command has been interrupted or cancelled. It's a good idea to print a log message in this command.

- Use this to tidy up after the command. The typical usage is to stop motors.
- Default implementation does nothing.

These methods (as well as subsystem `periodic` methods and any `Triggers` you have created) all run in a single shared thread, which is expected to run fifty times a second. This means that they all share a total time budget of 20ms. It is important that these commands execute quickly, so avoid using any long-running loops, sleeps, or timeouts. The scheduler will only run one command lifecycle method (`initialize`, `isFinished`, `execute`, `end`) or subsystem `periodic` at a time, so if you stay within this framework you don't have to worry about being thread-safe.

Generally commands exist in order to do something with a subsystem, like run motors. It's very important that you never have two commands trying to control the same motor. WPILIB's solution to this is called "subsystem requirements". Generally you will pass the subsystems into the command's constructor (along with any other configuration) to be stored for later use. In the constructor, you should also call `addRequirements(...)` with any subsystems used by the command. In some complex cases, you may use a subsystem without requiring it, say because you are only reading sensor data and setting any motor speeds.

A programmer needs to be familiar with the various command-related tricks available in WPILIB. I've divided them here into six groups:

- Command groups: Classes that take one or more commands and execute them all.
- Commands for use in groups: Commands that are useful when using command groups.
- Runnable wrappers: Classes that turn runnables into commands
- Subsystem wrapper method: Methods on `Subsystem` that turn runnables into commands
- Command decorators: Methods provided by all commands to connect or change them.
- Running commands: How to run a command
- Esoteric commands: Commands that are used only in specialized circumstances

These might seem a little complex and daunting, but the good news is that if you use them effectively your code will become simpler and easier to read. There are many subtle gotchas about combining commands, and these help you to navigate them safely.

Command groups

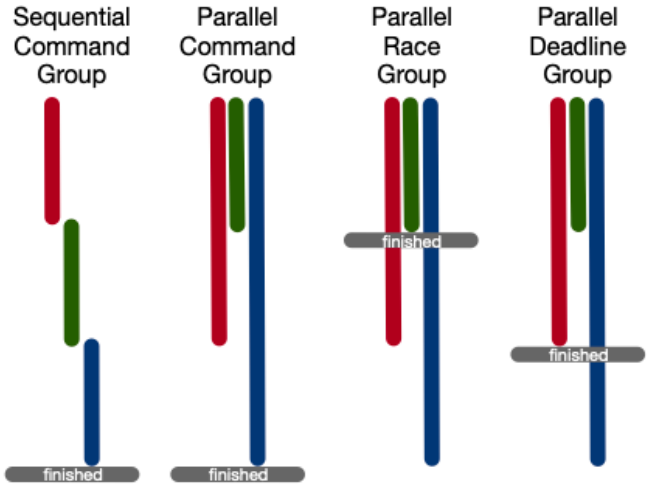
These classes group togather one or more commands and execute them all in some order. They inherit the subsystem requirements of all of their sub-commands. The sub-commands can be specified either in the constructor, or by subclassing and using `addCommands`.

SequentialCommandGroup

- Runs the sub-commands in sequence.
- See also `andThen` and `beforeStarting`.

ParallelCommandGroup

- Runs the sub-commands in parallel.



`SequentialCommandGroup` runs each command in turn until the last finishes. `ParallelCommandGroup` runs the commands in parallel, until they all finish. `ParallelRaceGroup` runs until the fastest command finishes. `ParallelDeadlineGroup` runs until the first command finishes.

- Finishes when the slowest sub-command is finished.
- See also the decorator `alongWith`.

ParallelRaceGroup

- Runs the sub-commands in parallel.
- Finishes when the fastest sub-command is finished.
- See also the decorator `raceWith`.

ParallelDeadlineGroup

- Runs the sub-commands in parallel.
- Finishes when the first command in the list is finished.
- See also the decorator `deadlineWith`.

Commands used in groups

The following commands are useful to build command groups. Some of them take commands as arguments, and their subsystem requirements are inherited.

- `ConditionalCommand`: Given a condition (evaluated in `initialize`), runs one of two sub-commands. See also the decorator `unless`.
- `SelectCommand`: Takes a mapping from keys to commands, and a key selector. At `initialize`, the key selector is executed and then one of the sub-commands is run.
- `ProxyCommand`: This behaves exactly like the underlying command except that subsystem requirements are not shared between the child and parent commands. See also the decorator `asProxy`. `Warning: ProxyCommand` works by scheduling the command independently and waiting for it to complete. A consequence of this is that any scheduled commands with overlapping requirements will be interrupted. If this includes the command group that is using `ProxyCommand`, then the proxy command will also be canceled.
- `RepeatCommand`: Run the sub-command until it is finished, and then start it running again. See also the decorator `repeatedly`.
- `WaitCommand`: Insert a delay for a specific time.
- `WaitUntilCommand`: Insert a delay until some condition is met.

Runnable wrappers

Here are some wrapper classes that turn runnables (e.g. lambda expressions) into commands. These can be used in command groups, but they are also used in `RobotContainer` to create command on-the-fly. When using these methods, please remember to add the subsystem(s) as the last parameter(s) to make subsystem requirements work correctly.

- `InstantCommand`: The given runnable is used as the `initialize` method, there is no `execute` or `end`, and `isFinished` returns `true`. You will also sometimes inherit from `InstantCommand` instead of `BaseCommand`.
- `RunCommand`: The given runnable is used as the `execute` method, there is no `initialize` or `end`, and `isFinished` returns `false`. Often used with a decorator that adds an `isFinished` condition.
- `StartEndCommand`: The given runnables are used as the `initialize` and `end` methods, there is no `execute`, and `isFinished` returns `false`. Commonly used for commands that start and stop motors.
- `FunctionalCommand`: Allows you you set all four life-cycle methods. Not used if one of the above will suffice.

Class	initialize	execute	end	isFinished
InstantCommand	arg 1			true

Class	initialize	execute	end	isFinished
RunCommand		arg 1		false
StartEndCommand	arg 1		arg 2	false
FunctionalCommand	arg 1	arg 2	arg 3	arg 4

Subsystem wrapper methods

The Subsystem class provides some useful methods that provide more or less the same functionality as the classes above, with the feature that they automatically get the subsystem as a requirement.

Method	initialize	execute	end	isFinished	Equivalent to
run		arg 1		false	RunCommand
runEnd		arg 1	arg 2	false	
runOnce	arg 1			true	InstantCommand
startEnd	arg 1		arg 2	false	StartEndCommand
startRun	arg 1	arg 2		false	

Subsystem also provides a method defer which is used to create DeferredCommand. This take a Command supplier, so the underlying command is not determined until initialize.

These methods are also available on Commands, but the Subsystem version is preferred because it makes it hard to forget the subsystem requirement.

Command decorators

These are methods that are provided by all Commands and allow you to create new commands that modify the underlying command in some way, or implicitly create command groups. These can be used as an alternative way to write command groups, but are also used when creating commands on-the-fly in RobotContainer.

- alongWith: Runs the base command and the sub-command(s) in parallel ending when they are all finished (cf ParallelCommandGroup)
- andThen: Runs the base command and then the sub-command(s) or runnable. See also the class SequentialCommandGroup.
- asProxy: Blocks inheritance of subsystem requirements. See also the class ProxyCommand.
- beforeStarting: Runs the sub-commands or runnable and then the base command. See also the class SequentialCommandGroup.
- deadlineWith: Runs the base command and sub-commands in parallel, ending when the base command is finished. See also the class ParallelDeadlineGroup.
- finallyDo: Adds an (additional) end method to a command.
- raceWith: Runs the base command and sub-commands in parallel, ending when any of them are finished. See also the class ParallelRaceGroup.
- repeatedly: Runs the base command repeatedly. See also the class RepeatCommand.
- unless: Runs the command only if the supplied BooleanSupplier is false. See also the class ConditionalCommand.
- until: Overrides the isFinished method with a BooleanSupplier.
- withTimeout: Adds a timer-based isFinished condition.

(I have omitted a few of the more esoteric decorators for brevity.)

Running commands

There are generally three ways to run a command:

- Bind it to a trigger, usually a joystick button
- Run it by default
- Run it in autonomous mode

Triggers

Triggers are objects that run some command when some event takes place, like a button being pressed. The easiest way to create a trigger is by using a `CommandJoystick` or `CommandXboxController`. Trigger objects don't need to be stored.

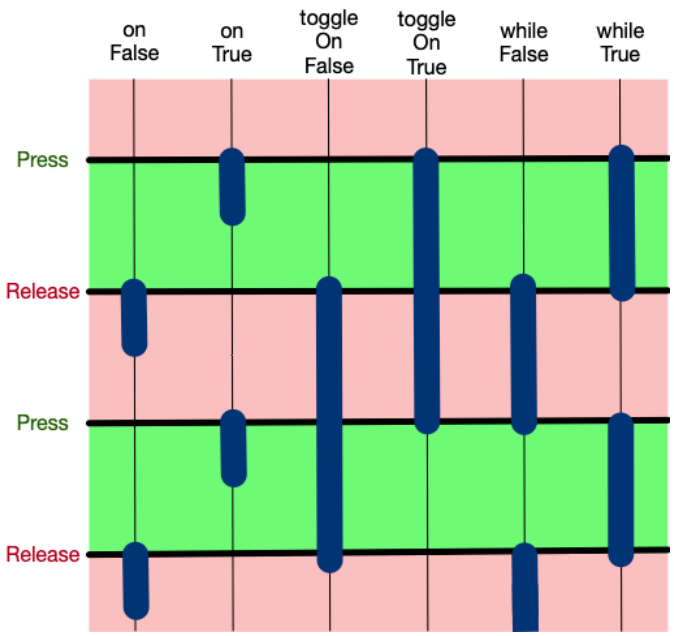
```
CommandJoystick joystick = new CommandJoystick(0);

joystick.button(1).toggleOnTrue(new MyCommand(...))
```

It is also possible to create triggers from any Boolean supplier:

```
new Trigger(() -> subsystem.getLimitSwitch()).whileTrue(...)
```

Some trigger methods should be passed a command to run:

- `onFalse`: Starts the command when the trigger becomes false, e.g. the button is released. Usually the command will have its own `isFinished` condition. Often used for instant commands.
- `onTrue`: Starts the command when the trigger becomes true, e.g. the button is pressed. Usually the command will have its own `isFinished` condition. Often used for instant commands.
- `toggleOnFalse`: Starts or stops the command when the trigger becomes false. Seldom used. Usually this command will otherwise run indefinitly (`isFinished` returns `false`).
- `toggleOnTrue`: Starts or stops the command when the trigger becomes true. For example, press a button and the intake starts running; it keeps running until the button is pressed a second time. Usually this command will otherwise run indefinitly (`isFinished` returns `false`).



`onTrue` starts when a button is pressed and usually ends on its own. `whileTrue` starts when the button is pressed and runs until it is released. `toggleOnTrue` turns on or off in alternation every time the button is pressed. `onFalse`, `whileFalse`, and `toggleOnFalse` do the same, but when the button is released.

- `whileFalse`: Starts the command when the trigger becomes false, and stops it when the trigger becomes true. Usually this command will otherwise run indefinitly (`isFinished` returns `false`).
- `whileTrue`: Starts the command when the trigger becomes true, and stops it when the trigger becomes true. For example, the robot feeds balls into the shooter while the button is pressed, and stops when it is released. Usually this command will otherwise run indefinitly (`isFinished` returns `false`).

Some trigger methods create new triggers:

- `and`: Combines the trigger with the parameter (often another trigger) to make a trigger than only activates when both triggers are true.
- `debounce`: Creates a new trigger than only activates when the underlying trigger has been true for some period of time. This is useful for physical sensors and buttons that may be jittery.
- `negate`: Creates a new trigger that is only true when the underlying trigger is false.
- `or`: Combines the trigger with the parameter (often another trigger) to make a trigger than only activates when either trigger is true.

Of these, you will probably use `onTrue` (for instant commmands), `whileTrue` (to run while pressed), `toggleOnTrue` (to turn on or off when pressed), and `debounce` (to smooth noisy signals) most often.

Default commands

Sometimes you want a command to run all the time on some subsystem, unless you have something more specific to run. This is the "default command" for that subsystem.

The most commonly encountered example of a default command is the "Arcade Drive" command, which connects a joystick to the drive subsystem. This will run all of the time, except when you engage some autonomous driving routine.

To set the default command for a subsystem, simply call `setDefaultCommand()`. Each subsystem can only have (at most) one default command. When using default commands, it is important that all commands using that subsystem have their requirements set correctly; this ensures that the scheduler will deschedule the default command when they are scheduled.

```
// in RobotContainer.java, in configureBindings()
m_driveSubsystem.setDefaultCommand(
    new ArcadeDriveCommand(m_driveSubsystem,
        () -> m_joystick1.getX(), // double supplier for turn
        () -> m_joystick1.getY()); // double supplier for speed
```

Autonomous commands

TODO

Esoteric commands

These commands are used only in very specific circumstances.

- `NotifierCommand`
- `RamseteCommand`
- `ScheduleCommand`
- `ProxyScheduleCommand`

- `WrapperCommand`
- `MecanumControllerCommand`
- `SwerveControllerCommand`
- `TrapezoidProfileCommand`

See also

- Binding Commands to Triggers
- Command Compositions
- Command interface