Multithreading for FTC

Background: Program, Process, Thread

Process Model

- A program is a static entity made up of program statements; an example is pycharm64.exe.
- When a program runs on a particular operating system it does so as a process. The operating system manages all running processes.
- Process components:
  - The code to be executed
  - The internal data that belongs to the process
  - Private or shared resources that the process can access
- Two or more processes may execute the same program, each with its own data and resources. For example, you could run two instances of the Windows Calculator.

Threads

- A ***thread* of *execution*** runs within a process and is defined as a sequence of programmed instructions that can be scheduled independently.
  - Several threads may run concurrently within a single process
  - Threads may share data and resources
  - Threads may communicate with each other
- The sharing of data and the coordination of behavior between threads can be complex and error-prone.

Threading in FTC

- Fortunately, multi-threading in FTC can be simplified. Let's look at a typical use-case from the autonomous portion of the Freight Frenzy game.
- Sequential; the main thread --
  - Drives the robot into position at the Shipping Hub
  - Raises the delivery arm to the correct level
  - Delivers the freight
- Multi-threaded; the main thread --
  - Launches a thread that starts the robot moving in the direction of the Shipping Hub
  - Launches another thread that starts raising the delivery arm
  - Waits for both threads to complete, then delivers the freight

Threading in FTC

- In our example there is:
  - No shared data between threads
  - No communication between threads
- The main thread only needs to launch two threads and wait for both to complete

Threading in Java 8

- There is a long history of threading in Java but we will start with the most recent improvements that came in Java 8.
- To start a thread in Java 8 we use the class CompletableFuture<T> where <T> is the generic return type.
- The next slide shows an example of a CompletableFuture that returns a String.

Threading in Java 8

```
CompletableFuture<String> future =
 CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Result of the asynchronous
 computation";
});
```

- We'll go through this unfamiliar syntax after the next slide.

Threading in Java 8

- Now the thread is running so you can do any number of other things, including starting another thread.
- When you're ready to get the results of the thread you call "get" to block until the thread is done, at which point the return value is available in the variable "result" OR any exceptions that occurred in the thread are thrown.

```
String result = future.get();
System.out.println(result);
```

Multithreading

- There is actually quite a lot going on in this code.
- The return type of the CompletableFuture is generic in line with Java's generic collection classes and functional interfaces.
- CompletableFuture.supplyAsync is a method which, in our example, takes a Java lambda as its argument. The code that will be executed asynchronously is declared directly in the method call. Java lambda syntax allows for the passing of code as data.
- But what if the code in the lambda were 200 lines long or what if you wanted to use the same code in another context? Answer: you can package the code inside a Java 8 functional interface – Callable<T>.

Threading in Java 8

```
 // Keep monitoring the touch sensor until the operator presses
// the button pMaxTouchCount times.
private Callable<Void> makeTouchSensorCallable(int pMaxTouchCount) {
    return () -> {
        int currentTouchCount = 0;
        boolean waitForRelease = false;
        while (opModeIsActive() && currentTouchCount < pMaxTouchCount)
         {
            if (!waitForRelease && touchSensor.isPressed()) {
                waitForRelease = true;
                telemetry.addData("Touch sensor count", "%2d",
                  ++currentTouchCount);
                telemetry.update();
            }

              if (waitForRelease && !touchSensor.isPressed())
                waitForRelease = false;
         }

         return null;
    };
}
```

Multithreading

- The method makeTouchSensorCallable also uses lambda syntax to return a Callable.
- But note that the code inside the lambda is not executed at this point. The code is packaged inside the Callable so that it can be executed later.
- Also: a Callable<T> in itself says nothing about whether the code inside will be used in a thread.
- But in our case, we will use the Callable<T> in a thread, as the next slide shows.

Threading in Java 8

```
// Call a local method to create the Callable
// in advance.
Callable<Void> touchSensorCallable =
makeTouchSensorCallable(5);

// Run the touch sensor *asynchronously* until
// the user presses the button 5 times.
CompletableFuture<Void> touchSensorFuture =
Threading.launchAsync(touchSensorCallable);

// The touch sensor is active. Other code can go
// here.
// …
```

Threading in Java 8

```
RobotLog.d("Wait for the touch sensor thread to
complete");
Threading.getFutureCompletion(touchSensorFuture);
RobotLog.d("Touch sensor thread done");
```

Threading in Java 8

- **Now we can look at the actual code in the demo.**