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

Function References, Lambda Functions, and more

Function References

The normal way to write functions in Java is to put them in a class. Functions in a class are called "methods" and must be called either on an instance object or (for `static` methods) on the class itself.

Sometimes you want to be able to use a function as an argument to another function. Such an argument is often referred to as a "callback function", and it allows other code to call parts of your code later in a very flexible way. An API that accepts a function reference can call your code without having to know anything about how it is written or what your methods are called.

Callback functions typically do one of three things: They supply information; they consume information; or they do something.

For a function to be used as an argument, it has to be a "function reference". In WPILIB, many methods related to triggers and commands will accept function references. There are two ways to create function references: lambda functions (`->`) and the method reference operator (`::`).

Lambda Expressions 🔗

The most common way to create such function references is using a lambda expression. This is a special expression that creates a function (reference) on-the-fly. Lambda expressions use the special operator `->`. (Compare with the `lambda` keyword in languages like Python.)

```
// Lambda expression for a function that takes three arguments
// and calculates their sum
(x, y, z) -> x + y + z

// Same thing, but with a statement block
(x, y, z) -> {
    return x + y + z;
}
```

On the left of the `->` operator is the parameter list. This has zero or more parameters in parentheses. The parentheses are optional when there is exactly one parameter. You may specify types of the parameters, but this is almost always optional because it can be deduced from the context.

On the right of the `->` operator is either an expression or a statement block in braces. A simple expression is used as the return value of the lambda expression. A statement block is executed like a normal function body. If there is a `return` statement, then this defines the value returned by the function; otherwise the function has `void` return.

Remember that the lambda expression only defines the function reference. Just like in a normal method, the function body is not evaluated until the function is actually invoked.

Lambda expressions have access to local and instance variables. This means that you can simply use these variables in the function body. This allows your lambda function to be configured in a way that is invisible to the code that calls it. It also allows it to have access to controls and information that the calling code does not. Your lambda function can do anything that you can do, but it is packaged up as something that other code can invoke without any explicit dependency on you.

Method Reference Operator

It is also possible to turn any method into a function reference using the `::` method reference operator.

```
class MySubsystem ... {
    ...
    boolean hasGamePiece() { ... }
}

...

private final MySubsystem subsystem = new MySubsystem();

// Get an anonoymous function reference with the same arguments
// and return type.  Use this as a boolean supplier without
// having to know about the subsystem.
subsystem::hasGamePiece
```

On the left of the `::` operator is a reference to some object. In the example, `subsystem` is a reference to a `MySubsystem` object. It is also common to see the `this` keyword used here, which allows you to reference methods on the current object.

On the right of the `::` is the name of a (public) method defined on the object. In the example, `hasGamePiece` is a method defined in `MySubsystem`. It is also possible to use the `new` keyword here, which allows you to reference the constructor method.

Putting the object reference and the method name together with the `::` operator yields a function reference that has the same arguments and return type, but which can be invoked as a function reference without reference to the instance object.

The function reference in the example above takes no arguments and returns a `boolean`, so matches the pattern required for a `BooleanSupplier`.

Interfaces

If you look up the documentation for WPILIB functions, you'll see that they don't actually take function references explicitly. Instead they accept an instance of some interface like `Supplier`, `Consumer`, `Runnable`, or `Callable`.

These are all "functional interfaces" which means you can just use a function reference instead and Java will automatically convert it for you.

Type	Arguments	Return	Methods	Notes
`Supplier` `<X>Supplier`	None	thing supplied, e.g. `boolean`	`get` or `getAs<X>`, e.g. `getAsBoolean`	Supplies information, e.g. `BooleanSupplier`
`Runnable`	None	None	`run`	Does something, e.g. runs command
`Consumer`	thing consumed, eg. `Pose2d`	None	`accept`	Accepts inputs of some type
`Callable`	None	result of some type	`call`	Also supplies things, but has special uses

Suppliers (especially `BooleanSupplier`s) and runnables are often encountered with WPILIB, whereas consumers and callables are not.

Suppliers

A supplier is a class that supplies values of some specific type when you call the appropriate `getAsX` or `get` method. It can be created from an anonymous function that takes no argument and returns a value of that type. Suppliers that return objects are not required to return a fresh object every time (so would not be used for an anonymous command factory).

```
() -> subsystem.hasGamePiece() // BooleanSupplier
() -> joystick.getX()           // DoubleSupplier
() -> joystick.getX() > 0.0     // BooleanSupplier
```

These suppliers can then be used later to fetch the value:

```
class ArcadeDrive extends Command {
    private DriveSubsystem m_subsystem;
    // Keep suppliers in instance variables
    private DoubleSupplier m_speed;
    private DoubleSupplier m_turn;

    public ArcadeDrive(DriveSubsystem subsystem,
        DoubleSupplier speed, DoubleSupplier turn) {
        m_subsystem = subsystem;
        // Store these suppliers for later use
        m_speed = speed;
        m_turn = turn;

        addRequirements(subsystem);
    }

    @Override
    void execute() {
        // Get values from the suppliers
        double drive = m_drive.getAsDouble();
        double turn = m_turn.getAsDouble();
        m_subsystem.drive(drive+turn, drive-turn);
    }
}

...

// In RobotContainer we connect the suppliers to the joystick axes
m_drive.setDefaultCommand(new ArcadeDrive(m_drive),
    () -> adjustJoystick(-m_joystick.getY()),
    () -> adjustJoystick(-m_joystick.getX()));
```

See also the similar but different example at Default Commands.

Unboxed types (non-objects) like `boolean` and `double` have special supplier types like `BooleanSupplier` and `DoubleSupplier` with methods like `getAsBoolean()` and `getAsDouble()` to get the value. Boxed types (objects) are treated differently: For example, a supplier for `Pose2d` objects would be `Supplier<Pose2d>`, and the accessor is simply `get()`.

Suppliers are a good way to isolate dependencies. In the code above, `speed` and `turn` come from a joystick, but this code doesn't need to know anything about joysticks. This means that you can change to a different type of joystick or even bring in semi-autonomous "driver assist".

In WPILIB, `Trigger`s implement the `BooleanSupplier` interface and often accept it.

Runnables

A `Runnable` is a class that has a `void run()` method. It can be created from an anonymous function that takes no arguments (and any return value is ignored). In WPILIB, it can be used to create commands on-the-fly using the classes `InstantCommand`, `RunCommand`, `StartEndCommand`, or `FunctionalCommand` or certain `Subsystem` methods `. It can also be passed to many trigger methods (along with a required subsystem).

```
// This expression can be used as a Runnable
() -> { /* do stuff here */ }
```

Because `Runnable`s have no parameters and no return, they are executed solely for their side-effects.

Consumers

A `Consumer` is similar to a supplier but in reverse. With a `Supplier`, the receiver gets to decide when and how often it is called. With a `Consumer`, the sender makes those decisions. Instead of having a `get` or `getAs<TYPE>` method, a `Consumer` has an `accept` method.

`Consumer`s are not much used in FRC programming, but they might be useful in a case where it's important that each value be processed exactly once. An example of this might be camera frames, or information derived from that such as robot location, or game piece locations.

```
// Define a record type to consume
public record VisionMeasurement(Pose2d pose,
    double timestamp, Matrix<N3,N1> stddevs) {}

// Expose a consumer that processes the records
public final Consumer<VisionMeasurement> visionMeasurementConsumer = (vm)
-> {
    m_poseEstimator.addVisionMeasurement(vm.pose(),
        vm.timestamp(), vm.stddevs());
};

// ...

// Somewhere else, pass new records to the consumer
m_visionMeasurementConsumer.accept(
    new VisionMeasurement(pose, timestamp, stddevs));
```

Callables

A `Callable` is like a cross between `Runnable` and `Supplier`. It has a `call()` method that returns a value of some type. `Callable`s may throw exceptions.

`Callable`s are used when expecting a new result each time (like an anonymous factory), when the work involves things that might throw (like file or network input/output), or when passing function references between threads. These are not much used in FRC programming.

See also

- Functions as Data