OpenSensorHub

Development Training
Building the Sensor Output

Lab 6 – 90 minutes
Requirements

• Java Programming Language – Entry Level Experience

• Lab 5 Complete!
Describing and Defining the Output

Using API
API: AbstractSensorOutput

- Class providing default implementation of common output API methods. This can be used as the base for most sensor outputs implementations as it aids in generating the following:
  - Record description
    - Output description
      - Including name, label, description and structure
  - Record encoding
    - The default encoding to use when publishing the sensors observations
    - Encoding examples:
      - Binary, CSV (Comma Separated Values), Text encoding
Output extends AbstractSensorOutput

```java
package com.sample.impl.sensor.simulated;

import net.opengis.swe.v20.*;
import org.sensorhub.api.data.DataEvent;
import org.sensorhub.impl.sensor.AbstractSensorOutput;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.vast.swe.helper.GeoPosHelper;

import java.io.PrintWriter;
import java.io.StringWriter;
import java.lang.Boolean;

/*
 * Output specification and provider for ...
 *
 * @author Nick Garay
 * @since Feb. 6, 2020
 */
public class Output extends AbstractSensorOutput<
Sensor> implements Runnable {
```
Creating the Output Description

• When creating the output description you are actually performing two integral operations at the same time

  • The first is that you are building the SensorML description of the output

  • The second is that you are defining the data structure that will be instantiated and used to populate the observations for publication

  • It will be important to note the order in which fields are added to the output description’s DataRecord. This will be the same order in which you populate the output data structure created from the description.
Navigate to Output(.java) and Open
Update Properties

• Constants are provided so that it is easy to change if necessary
• The following constants relate to the user friendly descriptive elements of the output
  • SENSOR_OUTPUT_NAME
  • SENSOR_OUTPUT_LABEL
  • SENSOR_OUTPUT_DESCRIPTION
Change the Value of Properties as Necessary

```java
private static final String SENSOR_OUTPUT_NAME = "Weather";
no usages

private static final String SENSOR_OUTPUT_LABEL = "Weather";
no usages

private static final String SENSOR_OUTPUT_DESCRIPTION = "Weather measurements";
```
DataRecord and DataEncoding Properties

- The following properties are provided and will be defined in the `doInit` method of the class:
  - **DataRecord** – used to describe and define the structure of the output
  - **DataEncoding** – used to provide the default encoding method for the particular output
Other Properties

• These properties are defined by the output template to
  • allow for multithreaded execution
    • stopProcessing - boolean
    • processingLock – Object for synchronization
    • Worker – Handle to a thread that will run our output loop
  • allow timing to be computed for the average sample rate
    • setCount, timingHistogram, histogramLock, and MAX_NUM_TIMING_SAMPLES constant
Brief Words on Average Sampling Time

• The template code includes the constructs necessary for the average sampling time of an output to be computed.

• Average sampling time may be provided by the physical or virtual data source in which case the average sampling time can just be returned from the `getAverageSamplingPeriod` method with no need to calculate it.

• If not provided by the source, these properties are used to maintain a history of the delay between samples and computes an average after 10 samples are received.
Constructor

- Creates an instance of the output, important to invoke the super class’s (AbstractSensorOutput) constructor
- Logging is used here (optional) to write a message to the console notifying the constructor has been called
Output Initialization

- **doInit** method is called by the owning/parent Sensor class when initialization takes place.

- You will notice the template makes use of a Helper class – GeoPosHelper. This class is used to help build the DataRecord describing the output and its structure.

- DataEncoding is also specified here, in this case using a text encoding of CSV elements.

- Finally, a Thread object is allocated that will be used to perform the work of gathering and publishing observations for this output.
Creating the Data Record Description

• We will be replacing with the correct API calls to build the `DataRecord`.

• We will be making use of the Helper class, but be replacing it with a more generic version from `GeoPosHelper` to `SWEHelper`.

• We will be describing the record and each of its fields. Again, this will build the SensorML description as well as the data structure for the output observations.

• Note: The data structure is defined as per SWE Common Data Model, you don’t need to know this specification yet but it is useful to understand. It is recommended you familiarize yourself with it:
  
  [SWE Common Data Model Encoding Standard - Open Geospatial Consortium (ogc.org)](https://ogc.org)

```cpp
// TODO: Create data record description
```
Begin by Changing the Helper

```java
// Get an instance of SWE Factory suitable to build components
GeoPosHelper sweFactory = new GeoPosHelper();

// Get an instance of SWE Factory suitable to build components
SWEHelper sweFactory = new SWEHelper();
```
Create the Data Record Description and Data Structure

1. Begin creating the record assigning it a name, definition, and description

2. Add the sampling time – all data records must have a sampling time as the first field added
Add Quantity Fields

- Depending on the type of fields being added certain properties must be specified through the Helper APIs
- Here we add two fields, temperature (3) and pressure (4)
- Each of these are quantities, have an ontological definition (CF Standard Names), a label, and units of measure – temperature is given in Celsius, and pressure in hectopascals
Add Quantity Fields...

- The last two fields to add are Wind Speed and Wind Direction.
- Each of these are quantities and have m/s and deg as their units of measure respectively.
- Notice also that Wind Direction has a coordinate reference frame given as North East Down with an axis id of “z”.
  - This axis refers to the axis perpendicular to the surface of the earth at the location of the sensor.
Invoke **build** to Complete Record Construction
Controlling Outputs Execution
Start and Stop

**doStart** – called by parent Sensor to start the output process

**doStop** – called by parent Sensor to stop the output process

For this exercise neither of these require extra startup or shutdown procedures – so it is safe to remove the TODO comments
Reporting if Output isAlive

- The `isAlive` method of the output reports the status of the output to the parent/owner sensor.

- In this particular case we are checking to see if the worker thread is still running to determine if the output is still working or if it has been shutdown or failed.

```java
/**
 * Check to validate data processing is still running
 */

public boolean isAlive() {
    return worker.isAlive();
}
```

Already added for you by the template!
Record Description, Encoding, and Average Sampling Time Reports

• `getRecordDescription` uses the DataRecord dataStruct to provide the description and data structure for the output to OpenSensorHub

• `getRecommendedEncoding` reports the default encoding for the output

• `getAverageSamplingPeriod` computes the average from the historical time between 10 consecutive samples

Already added for you by the template!
Publishing Observations
Background on Threads and Runnables

• Recall that this class is an **Output** class extending **AbstractSensorOutput** but also you may have noticed that it implements **Runnable**

• What is Runnable?
  • It is an interface specification, a contract, that this class upholds to allow it to be used within a Thread
  • A thread runs as a separate process within the context of the parent application so it can be scheduled concurrently with other processes without blocking the parent process that spawned the thread
  • Don’t worry about the details of multithreading, simply, for now, know that it exists and allows for parallel workers to exist.
Understanding our Worker Thread

1. Created the thread in `doInit` specifying `this` as the instance invoked to do the work. When the thread runs an instance of this class’s object’s `run` method will be invoked.

2. Starts the thread’s execution.

3. A collapsed view of the `run` method which satisfies the Runnable contract (interface specification).
Decomposing the *run* Method

- `processSets`: a Boolean flag used to control the loop of execution
- `lastSetTimeMillis`: a value containing the CPU clock, initialized to current time in milliseconds
- `try – catch – finally`: a construct used to execute a code block in which an exception may arise, is gracefully dealt with, and cleans-up resources if necessary
  - If an exception occurs we log a trace of the execution stack as an error as well
  - Finally always gets executed whether an exception occurs or the code block within the try completes successfully

```java
boolean processSets = true;
long lastSetTimeMillis = System.currentTimeMillis();

try {...} catch (Exception e) {
    StringWriter stringWriter = new StringWriter();
    e.printStackTrace(new PrintWriter(stringWriter));
    logger.error("Error in worker thread: {} due to exception: {}", Thread.currentThread().getName(), stringWriter.toString());
} finally {
    logger.debug("Terminating worker thread: {}", this.name);
}
```
Looping in the Body of the **try** Block

- The body of the try block contains a loop controlled by the Boolean value *processSets*
  - if *processSets* evaluates to true the body of the loop is executed, otherwise the loop terminates and in effect the thread terminates
- The loop is responsible for acquiring the data structure for population, updating the histogram for average sample time calculations, populating and publishing the data structure as an observation
- Notice the template code takes care of most of the work for you, you just need to complete the population of the data block
- **Before we do so let's break down what is happening here**
Creating or Allocating the Data Structure

• This segment of the code declares a DataBlock variable that will be used to populate the data structure

• Notice the DataBlock is created from the DataRecord instance held in dataStruct

• Remember this object holds the description and structure of the data to be published

• The latestRecord property if defined allows us to reuse its memory, otherwise a new instance is created

Already added for you by the template!
Updating the Histogram for Average Sampling Time

- The synchronized key word allows for multiple threads to operate on the **timingHistogram** by limiting access when it is being read or written to by distinct threads.

- Here we simply calculate the time passed since the last sample was generated and store it in the histogram.

- Finally increment the count of samples.

```java
synchronized (histogramLock) {
    int setIndex = setCount % MAX_NUM_TIMING_SAMPLES;
    // Get a sampling time for latest set based on previous set sampling time.
    timingHistogram[setIndex] = System.currentTimeMillis() - lastSetTimeMillis;
    // Set latest sampling time to now.
    lastSetTimeMillis = timingHistogram[setIndex];
    ++setCount;
}
```
Publish the Observation

- First we store the **dataBlock** in **latestRecord**
- Get the current system time in milliseconds to stamp the publishing time of the observation
- Publish the observation as a new **DataEvent**
- Finally, check if **doStop** has been invoked. Again this is guarded for multithreaded read/write by the synchronized key work on the **processingLock** object

```java
latestRecord = dataBlock;
latestRecordTime = System.currentTimeMillis();
eventChannel.publish(new DataEvent(latestRecordTime, dataBlock));
synchronized (processingLock) {
    processSets |= stopProcessing;
}
```

Already added for you by the template!
But what about the actual observation values?

- We skipped this line on purpose.
- Replacing this line with the logic or maybe a method call to populate the data structure is necessary.
- In the next few slides will present and explain the process of populating the data structure with appropriate values.
- Remember this is a simulated sensor so, we will need to generate samples at regular intervals; to do so add the following.

```java
221 222 223 224
```

```
221
222
223
224
```

```java
221 222 223 224
```

```
// TODO: Populate data block
```

```
eventHandler.publish(new DataEvent(latestRecordTime, dataInterface Output.this, dataBlock));
```

```
Thread.sleep(milliseconds); // milliseconds
```

- This will set the interval to 10 seconds (10000 milliseconds) by causing this thread to sleep for that time between samples published, you are welcome to change this value but keep it to no less than 1sec (1000 milliseconds).
Create a Random Number Generator

• Insert the following member variables and method into the Output class just above the run method definition

• The variation method will allow us to generate the values we need for the simulated sensor

```java
// reference values around which actual values vary
double tempRef = 20.0;

double pressRef = 1013.0;

double windSpeedRef = 5.0;

double directionRef = 0.0;

// initialize then keep new values for each measurement
double temp = tempRef;

double press = pressRef;

double windSpeed = windSpeedRef;

double windDir = directionRef;

private Random rand = new Random();

private double variation(double val, double ref, double dampingCoeff, double noiseSigma)
{
    return -dampingCoeff*(val - ref) + noiseSigma * rand.nextGaussian();
}

@override
public void run()
{
}
```
Add Logic to Generate and Populate Values

```java
// Add logic to generate and populate values

double time = System.currentTimeMillis() / 1000.;

// Temperature; value will increase or decrease by less than 0.1 deg
double temp = variation(temp, tempRef, dampingCoeff: 0.001, noiseSigma: 0.1);

// Pressure; value will increase or decrease by less than 20 kPa
double press = variation(press, pressRef, dampingCoeff: 0.001, noiseSigma: 0.1);

// Wind speed; keep positive
// Vary value between -10 m/s and 10 m/s
double windSpeed = variate(windSpeed, windSpeedRef, dampingCoeff: 0.001, noiseSigma: 0.1);
windSpeed = Math.max(0, windSpeed);

// Wind direction; keep between 0 and 360 degrees
double windDir = Math.random() * 360;
windDir = windDir % 360;

parentSensor.getLogger().info("temp=\{\}, press=\{\}, wind speed=\{\}, wind dir=\{\}", temp, press, windSpeed, windDir);

// Build and publish data block
dataBlock.setDoubleValue(index: 0, value: time);
dataBlock.setDoubleValue(index: 1, value: temp);
dataBlock.setDoubleValue(index: 2, value: press);
dataBlock.setDoubleValue(index: 3, value: windSpeed);
dataBlock.setDoubleValue(index: 4, value: windDir);
```
Populate the Data According to Field Order and Value Type

• Remember the order of fields added to the record description?
• Populating the values must follow the same order and must also use the appropriate setter based on the data type, in this case all double

```java
.dataField(name="time", sObject.createTime()
// as SamplingTime.isUT())
.dataField(name="temperature", sObject.createQuantity
//.definition(ccHelper.getCFURI("ppName":"air_temperature")))
//.label("Air Temperature")
//.uomCode("C")
.dataField(name="pressure", sObject.createQuantity
//.definition(SHEMHelper.getCFURI("ppName":"air_pressure")))
//.label("Atmospheric Pressure")
//.uomCode("hPa")
.dataField(name="windSpeed", sObject.createQuantity
//.definition(SHEMHelper.getCFURI("ppName":"wind_speed")))
//.label("Wind Speed")
//.uomCode("m/s")
.dataField(name="windDirection", sObject.createQuantity
//.definition(SHEMHelper.getCFURI("ppName":"wind_from_direction")))
//.label("Wind Direction")
//.uomCode("deg")
//.refFrame(SWEConstants.REF_FRAME_AER, wind: "z")
```

```java
// build and publish dataBlock
dataBlock.setDoubleValue(index: 0, time);
dataBlock.setDoubleValue(index: 1, temp);
dataBlock.setDoubleValue(index: 2, press);
dataBlock.setDoubleValue(index: 3, windSpeed);
dataBlock.setDoubleValue(index: 4, windDir);
```
Build, Deploy, and Test

Rebuild and Deploy
Follow the Next Few Slides to Configure and Run your Module
Hit the Refresh Button

Output will change at the interval set for the output thread to sleep.
Closing Remarks

• At this point you have
  • Described and Defined the Output
  • Populated the Output data structure according to definition
  • Deployed and started the simulated sensor

• Try
  • Change the sleep time for the thread to increase the rate at which observations are generated

• Next Steps
  • Edit Configuration to GeoLocate the Simulated Sensor
  • Refactor the Sensor to have a more meaningful name and description
  • Familiarize Self with APIs for Requesting Data from OpenSensorHub